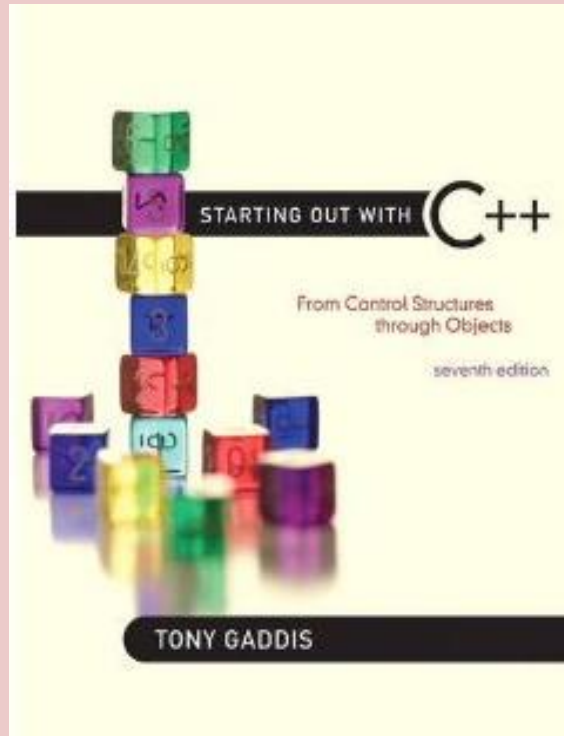


# Software Design & Programming I



*Starting Out with C++ (From Control Structures through Objects) 7th Edition*

*Written by: Tony Gaddis*

*Pearson - Addison Wesley*

*ISBN: 13-978-0-132-57625-3*

# Chapter 3

## Introduction to C++

# The *setprecision* Manipulator

Floating-point values may be rounded to a number of *significant digits*, or *precision*, which is the total number of digits that appear before and after the decimal point. You can control the number of significant digits with which floating-point values are displayed by using the *setprecision* manipulator.

# Combined Assignment Operators

```
12     cout << quotient << endl;
13     cout << setprecision(5) << quotient << endl;
14     cout << setprecision(4) << quotient << endl;
15     cout << setprecision(3) << quotient << endl;
16     cout << setprecision(2) << quotient << endl;
17     cout << setprecision(1) << quotient << endl;
18     return 0;
19 }
```

## Program Output

```
4.91877
4.9188
4.919
4.92
4.9
5
```

If the value of a number is expressed in fewer digits of precision than specified by `setprecision`, the manipulator will have no effect.

# The *setprecision* Manipulator

Floating-point values may be rounded to a number of *significant digits*, or *precision*, which is the total number of digits that appear before and after the decimal point. You can control the number of significant digits with which floating-point values are displayed by using the `setprecision` manipulator.

**Table 3-11** The `setprecision` Manipulator

Number	Manipulator	Value Displayed
28.92786	<code>setprecision(3)</code>	28.9
21.40	<code>setprecision(5)</code>	21.4
109.50	<code>setprecision(4)</code>	109.5
34.78596	<code>setprecision(2)</code>	35

# The fixed Manipulator

If a number is too large to print using the number of digits specified with `setprecision`, many systems print it in scientific notation.

```
Enter the sales for day 1: 145678.99[Enter]
Enter the sales for day 2: 205614.85[Enter]
Enter the sales for day 3: 198645.22[Enter]
```

```
Sales Figures
```

```
-----
```

```
Day 1: 1.4568e+005
```

```
Day 2: 2.0561e+005
```

```
Day 3: 1.9865e+005
```

```
Total: 5.4994e+005
```

To prevent this, you can use another stream manipulator, `fixed`, which indicates that floating-point output should be printed in *fixed-point*, or decimal, *notation*.

```
cout << fixed;
```

# The fixed Manipulator

**fixed** specifies the number of digits to be displayed after the decimal point of a floating-point number, rather than the total number of digits to be displayed.

```
Enter the sales for day 1: 321.57[Enter]
Enter the sales for day 2: 269.60[Enter]
Enter the sales for day 3: 307.00[Enter]

Sales Figures
-----
Day 1:    321.57
Day 2:    269.60
Day 3:    307.00
Total:    898.17
```

By using `fixed` and `setprecision` together, we get the desired output. Notice in this case, however, we set the precision to 2, the number of decimal places we wish to see, not to 5.

# The showpoint Manipulator

Another useful manipulator is showpoint, which indicates that a decimal point should be printed for a floating-point number, even if the value being displayed has no decimal digits.

## Program 3-19

```
1 // This program illustrates how the fixed, showpoint, and
2 // setprecision manipulators operate when used together.
3 #include <iostream>
4 #include <iomanip> // Needed to use stream manipulators
5 using namespace std;
6
7 int main()
8 {
9     double amount = 125.0;
10
11     cout << setw(6) << amount << endl;
12
13     cout << showpoint;
14     cout << setw(6) << amount << endl;
15
16     cout << fixed << showpoint << setprecision(2);
17     cout << setw(6) << amount << endl;
18     return 0;
19 }
```

## Program Output

```
125
125.000
125.00
```



# The showpoint Manipulator

Actually, when the fixed and setprecision manipulators are both used, it is not necessary to use the showpoint manipulator. For example,

```
cout << fixed << setprecision(2);
```

will automatically display a decimal point before the two decimal digits. However, many programmers prefer to use it anyway as shown here:

```
cout << fixed << showpoint << setprecision(2);
```

# The left and right Manipulators

Previously, we have seen, output that was right-justified. This means if the field it prints in is larger than the value being displayed, it is printed on the far right of the field, with leading blanks. There are times when you may wish to force a value to print on the left side of its field, padded by blanks on the right. To do this you can use the left manipulator. It remains in effect until you use a right manipulator to set it back. These manipulators can be used with any type of value, even a string.

# The left and right Manipulators

```
9     string month1 = "January",
10         month2 = "February",
11         month3 = "March";
12
13     int days1 = 31,
14         days2 = 28,
15         days3 = 31;
16
17     double high1 = 22.6,
18         high2 = 37.4,
19         high3 = 53.9;
20
21     cout << fixed << showpoint << setprecision(1);
22     cout << "Month      Days      High\n";
23
24     cout << left << setw(12) << month1
25         << right << setw(4) << days1 << setw(9) << high1 << endl;
26     cout << left << setw(12) << month2
27         << right << setw(4) << days2 << setw(9) << high2 << endl;
28     cout << left << setw(12) << month3
29         << right << setw(4) << days3 << setw(9) << high3 << endl;
30
31     return 0;
32 }
```

## Program Output

Month	Days	High
January	31	22.6
February	28	37.4
March	31	53.9

# The left and right Manipulators

**Table 3-12** Output Stream Manipulators

Stream Manipulator	Description
<code>setw(n)</code>	Sets the display field width to size <code>n</code> .
<code>fixed</code>	Displays floating-point numbers in fixed (i.e., decimal) form.
<code>showpoint</code>	Displays the decimal point and trailing zeroes for floating-point numbers even if there is no fractional part.
<code>setprecision(n)</code>	If the <code>fixed</code> manipulator is <i>not</i> set, <code>n</code> indicates the total number of significant digits to display. If the <code>fixed</code> manipulator is set, <code>n</code> indicates the number of decimal digits to display.
<code>left</code>	Causes subsequent output to be left-justified.
<code>right</code>	Causes subsequent output to be right-justified.

# Working with Characters and String Objects

Special functions exist for working with characters and string objects.

A char variable can hold only one character, whereas a variable defined as a string can hold a whole set of characters. The following variable definitions and initializations illustrate this.

```
char letter1 = 'A',  
      letter2 = 'B';  
string name1 = "Mark Twain",  
      name2 = "Samuel Clemens";
```

# Working with Characters and String Objects

As with numeric data types, characters and strings can be assigned values.

```
letter2 = letter1; // Now letter2's value is 'A'  
name2 = name1; // Now name2's value is  
// "Mark Twain"
```

They can also be displayed with the cout statement. The following line of code outputs a character variable, a string constant, and a string object.

```
cout << letter1 << ". " << name1 << endl;
```

The output produced is

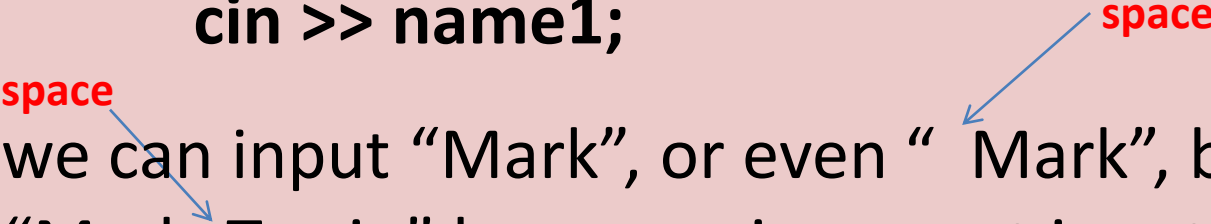
**A. Mark Twain**

# Inputting a String

Although it is possible to use `cin` with the `>>` operator to input strings, it can cause problems you need to be aware of. When `cin` reads data it passes over and ignores any leading whitespace characters (spaces, tabs, or line breaks). However, once it comes to the first nonblank character and starts reading, it stops reading when it gets to the next whitespace character. If we use the following statement

```
cin >> name1;
```

**space** we can input “Mark”, or even “ Mark”, but not “Mark Twain” because `cin` cannot input strings that contain embedded spaces. **space**



# Inputting a String

## Program 3-21 *(continued)*

```
7 int main()
8 {
9     string name;
10    string city;
11
12    cout << "Please enter your name: ";
13    cin >> name;
14    cout << "Enter the city you live in: ";
15    cin >> city;
16
17    cout << "Hello, " << name << endl;
18    cout << "You live in " << city << endl;
19    return 0;
20 }
```

First and last name entered, but the program accepted John for name and Doe for city.

### Program Output with Example Input Shown in Bold

```
Please enter your name. John Doe[Enter]
Enter the city you live in: Hello, John
You live in Doe
```



# Inputting a Character

## Program 3-23

```
1 // This program reads a single character into a char variable.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     char ch;
8
9     cout << "Type a character and press Enter: ";
10    cin  >> ch;
11    cout << "You entered " << ch << endl;
12    return 0;
13 }
```

### Program Output with Example Input Shown in Bold

```
Type a character and press Enter: A[Enter]
You entered A
```

Sometimes you want to read only a single character of input. Some programs display a menu of items for the user to choose from. The simplest way to read a single character is with `cin` and the `>>` operator.

# Using cin.get

As with string input, however, there are times when using `cin >>` to read a character does not do what we want. For example, because it passes over all leading whitespace, it is impossible to input just a blank or [Enter] with `cin >>`. The program will not continue past the `cin` statement until some character other than the spacebar, the tab key, or the [Enter] key has been pressed. (Once such a character is entered, the [Enter] key must still be pressed before the program can continue to the next statement.) Thus, programs that ask the user to "Press the enter key to continue." cannot use the `>>` operator to read only the pressing of the [Enter] key.

# Using cin.get

In those situations, a cin function called **get** becomes useful. The get function reads a single character, including any whitespace character. If the program needs to store the character being read, the get function can be called in either of the following ways. In both cases, ch is the name of the variable that the character is being read into.

```
cin.get(ch);
```

```
ch = cin.get();
```

If the program is using the **get** function simply to hold the screen until the [Enter] key is pressed and does not need to store the character, the function can also be called like this:

```
cin.get();
```

# Using cin.get

## Program 3-24

```
1 // This program demonstrates three ways
2 // to use cin.get() to pause a program.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     char ch;
9
10    cout << "This program has paused. Press Enter to continue.";
11    cin.get(ch);
12    cout << "It has paused a second time. Please press Enter again.";
13    ch = cin.get();
14    cout << "It has paused a third time. Please press Enter again.";
15    cin.get();
16    cout << "Thank you!";
17    return 0;
18 }
```

### Program Output with Example Input Shown in Bold

```
This program has paused. Press Enter to continue.[Enter]
It has paused a second time. Please press Enter again.[Enter]
It has paused a third time. Please press Enter again.[Enter]
Thank you!
```

# Mixing cin >> and cin.get

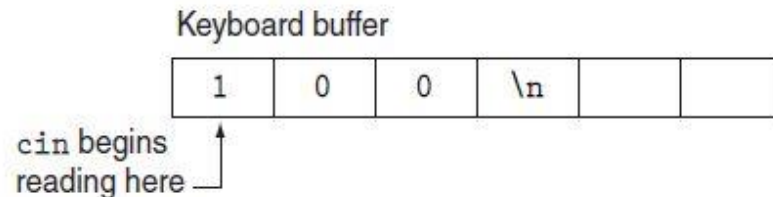
Mixing cin >> with cin.get can cause an annoying and hard-to-find problem. For example, look at the following code segment. The lines are numbered for reference.

```
1  char ch;           // Define a character variable
2  int number;       // Define an integer variable
3  cout << "Enter a number: ";
4  cin >> number;    // Read an integer
3  cout << "Enter a character: ";
6  ch = cin.get();   // Read a character
7  cout << "Thank You!\n";
```

# Mixing `cin >>` and `cin.get`

These statements allow the user to enter a number, but not a character. It will appear that the `cin.get` statement on line 6 has been skipped. This happens because both `cin >>` and `cin.get` read the user's keystrokes from the keyboard buffer. After entering a number in response to the first prompt, the user presses the [Enter] key. Pressing this key causes a newline character ('\n') to be stored in the keyboard buffer.

Figure 3-3



When the `cin >>` statement reads data from the keyboard buffer, it stops reading at the newline character. In our example, 100 is read in and stored in the number variable. The newline character is left in the keyboard buffer. However, `cin.get` always reads the next character in the buffer, no matter what it is, without skipping over whitespace.

# Using cin.ignore

**cin >>** only waits for the user to input a value if the keyboard buffer is empty. When **cin.get** finds the newline character in the buffer, it uses it and does not wait for the user to input another value. You can remedy this situation by using the **cin.ignore** function, described in the following section.

**Using cin.ignore To Solve This Problem.** The **cin.ignore** function can be used. This function tells the **cin** object to skip characters in the keyboard buffer.

# Using cin.ignore

Here is its general form:

```
cin.ignore(n, c);
```

The arguments shown in the parentheses are optional. If they are used, n is an integer and c is a character. They tell cin to skip n number of characters, or until the character c is encountered. For example, the following statement causes cin to skip the next 20 characters or until a newline is encountered, whichever comes first:

```
cin.ignore(20, '\n');
```

If no arguments are used, cin will only skip the very next character. Here's an example:

```
cin.ignore();
```



# Using cin.ignore

The statements that mix `cin >>` and `cin.get` can be repaired by inserting a `cin.ignore` statement after the `cin >>` statement:

```
cout << "Enter a number: ";  
cin >> number;  
cin.ignore();           // Skip the newline character  
cout << "Enter a character: ";  
cin.get(ch);  
cout << "Thank You!" << endl;
```

# Useful String Functions and Operators

The C++ string class provides a number of functions, called member functions, for working with strings. One that is particularly useful is the length function, which tells you how many characters there are in a string. Here is an example of how to use it.

```
string state = "New Jersey";  
int size = state.length();
```

The size variable now holds the value 10. Notice that a blank space is a character and is counted just like any other character. On the other hand, notice that the '\0' null character.

# Useful String Functions and Operators

The string class also has special operators for working with strings. One of them is the **+** operator.

You have already encountered the + operator to add two numeric quantities. Because strings cannot be added, when this operator is used with string operands it concatenates them, or joins them together. Assume we have the following definitions and initializations in a program.

```
string greeting1 = "Hello ",  
    greeting2;  
string name1 = "World";  
string name2 = "People";
```

# Useful String Functions and Operators

The following statements illustrate how string concatenation works.

```
greeting2 = greeting1 + name1; // greeting2 now holds "Hello World"
```

```
greeting1 = greeting1 + name2; // greeting1 now holds "Hello People"
```

Notice that the string stored in `greeting1` has a blank as its last character. If the blank were not there, `greeting2` would have been assigned the string "HelloWorld".

The last statement could also have been written using the `+=` combined assignment operator.

```
greeting1 += name2;
```

# Useful String Functions and Operators

The following statements illustrate how string concatenation works.

```
greeting2 = greeting1 + name1; // greeting2 now holds "Hello World"
```

```
greeting1 = greeting1 + name2; // greeting1 now holds "Hello People"
```

Notice that the string stored in `greeting1` has a blank as its last character. If the blank were not there, `greeting2` would have been assigned the string "HelloWorld".

The last statement could also have been written using the `+=` combined assignment operator.

```
greeting1 += name2;
```

# Using C-Strings

In C, and in C++ prior to the introduction of the string class, strings were stored as a set of individual characters. A group of contiguous 1-byte memory cells was set up to hold them, with each cell holding just one character of the string. A group of memory cells like this is called an array.

Because this was the way to create a string variable in C, a string defined in this manner is called a C-string.

```
char word[10] = "Hello";
```

Defines **word** to be an array of characters that will hold a C-string and initializes it to "Hello".

**word can only hold a string of nine characters because of the null character**

H	e	l	l	o	\n				
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

# Using C-Strings

## Program 3-25

```
1 // This program uses cin >> to read a word into a C-string.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     const int SIZE = 12;
8     char name[SIZE];          // name is a set of 12 memory cells
9
10    cout << "Please enter your first name." << endl;
11    cin  >> name;
12    cout << "Hello, " << name << endl;
13    return 0;
14 }
```

### Program Output with Example Input Shown in Bold

```
Please enter your first name.
Sebastian[Enter]
Hello, Sebastian
```

# Assigning a Value to a C-String

The first way in which using a C-string differs from using a string object is that, except for initializing it at the time of its definition, it cannot be assigned a value using the assignment operator. In Program 3-25 we could not, for example, replace the cin statement with the following line of code.

```
name = "Sebastian";           // Wrong!
```

Instead, to assign a value to a C-string, we must use a function called strcpy (pronounced string copy) to copy the contents of one string into another. Example:

```
strcpy(Cstring, value);
```



# Using C-Strings

## Program 3-26

```
1 // This program uses the strcpy function to copy one C-string to another.
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     const int SIZE = 12;
8     char name1[SIZE],
9         name2[SIZE];
10
11     strcpy(name1, "Sebastian");
12     cout << "name1 now holds the string " << name1 << endl;
13
14     strcpy(name2, name1);
15     cout << "name2 now also holds the string " << name2 << endl;
16
17     return 0;
18 }
```

## Program Output

```
name1 now holds the string Sebastian
name2 now also holds the string Sebastian
```

# Keeping Track of a How Much a C-String Can Hold

Another crucial way in which using a C-string differs from using a string object involves the memory allocated for it. With a string object, you do not have to worry about there being too little memory to hold a string you wish to place in it. If the storage space allocated to the string object is too small, the string class functions will make sure more memory is allocated to it. With C-strings this is not the case. The number of memory cells set aside to hold a C-string remains whatever size you originally set it to in the definition statement.

Characters that don't fit will spill over into the following memory cells, overwriting whatever was previously stored there. This type of error, known as a buffer overrun, can lead to serious problems.

# Ways of preventing a Buffer Overrun

One way to prevent this from happening is to use the `setw` stream manipulator. This manipulator, which we used earlier in this chapter to format output, can also be used to control the number of characters that `cin >>` inputs on its next read, as illustrated here:

```
char word[5];  
cin >> setw(5) >> word;
```

Another way to do the same thing is by using the `cin width` function.

```
char word[5];  
cin.width(5);  
cin >> word;
```

**In both cases the field width specified is 5 and `cin` will read, at most, one character less than this, leaving room for the null character at the end.**

# Ways of preventing a Buffer Overrun

## Program 3-28

*(continued)*

```
10
11     cout << "Enter a word: ";
12     cin.width(SIZE);
13     cin >> word;
14     cout << "You entered " << word << endl;
15
16     return 0;
17 }
```

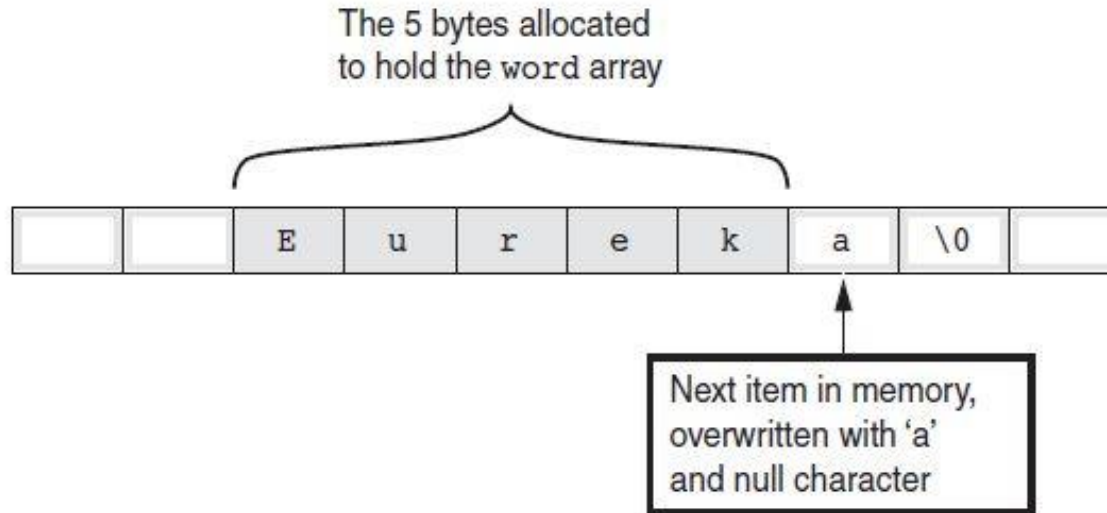
### Program Output for Programs 3-27 and 3-28 with Example Input Shown in Bold

Enter a word: **Eureka**[Enter]

You entered Eure

# Ways of preventing a Buffer Overrun

Figure 3-5



There are three important points to remember about the way `cin` handles field widths:

- The field width only pertains to the very next item entered by the user.
- To leave space for the `'\0'` character, the maximum number of characters read and stored will be one less than the size specified.
- If `cin` comes to a whitespace character before reading the specified number of characters, it will stop reading.

# Reading a Line of Input

Still another way in which using C-strings differs from using string objects is that you must use a different set of functions when working with them. To read a line of input, for example, you must use `cin.getline` rather than `getline`. These two names look a lot alike, but they are two different functions and are not interchangeable. Like `getline`, `cin.getline` allows you to read in a string containing spaces. It will continue reading until it has read the maximum specified number of characters, or until the [Enter] key is pressed. Here is an example of how it is used:

```
cin.getline(sentence, 20);
```

The `getline` function takes two arguments separated by a comma. (**name of the array, size of the array**)

# Reading a Line of Input

## Program 3-29

```
1 // This program demonstrates cin's getline function
2 // to read a line of text into a C-string.
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     const int SIZE = 81;
9     char sentence[SIZE];
10
11     cout << "Enter a sentence: ";
12     cin.getline(sentence, SIZE);
13     cout << "You entered " << sentence << endl;
14     return 0;
15 }
```

### Program Output with Example Input Shown in Bold

Enter a sentence: **To be, or not to be, that is the question.**[Enter]  
You entered **To be, or not to be, that is the question.**

# Random Numbers

Some programs need to use randomly generated numbers. The C++ library has a function called `rand()` for this purpose. To use the `rand()` function, you must include the `cstdlib` header file in your program. The number returned by the function is an `int`. Here is an example of how it is used.

```
randomNum = rand();
```

However, the numbers returned by the function are really pseudorandom. This means they have the appearance and properties of random numbers, but in reality are not random.

They are actually generated with an algorithm. The algorithm needs a starting value, called a seed, to generate the numbers. If it is not given one, it will produce the same stream of numbers each time it is run.



# Random Numbers

## Program 3-32

```
1 // This program demonstrates random numbers, providing
2 // a "seed" for the random number generator.
3 #include <iostream>
4 #include <cstdlib>
5 using namespace std;
6
7 int main()
8 {
9     int num1, num2, num3; // These hold the 3 random numbers
10    unsigned seed;       // Random generator seed
11
12    // Get a "seed" value from the user
13    cout << "Enter a seed value: ";
14    cin  >> seed;
15
16    // Set the random generator seed before calling rand()
17    srand(seed);
18
19    // Now generate and print three random numbers
20    num1 = rand();
21    num2 = rand();
22    num3 = rand();
23    cout << num1 << "      " << num2 << "      " << num3 << endl;
24    return 0;
25 }
```

### Program Output with Example Input Shown in Bold

#### Run 1:

```
Enter a seed value: 19[Enter]
100      15331      209
```

#### Run 2:

```
Enter a seed value: 171[Enter]
597      10689      28587
```

# Introduction to Files

Program input can be read from a file and program output can be written to a file.

If a program is to retain data between the times it runs, it must have a way of saving it. Data is saved in a file, which is usually stored on a computer's disk. Once the data is saved by writing it into a file, it will remain there after the program stops running. The data can then be retrieved and used at a later time.

# Introduction to Files

There are five steps that must be taken when a file is used by a program:

1. Include the header file needed to perform file input/output.
2. Define a file stream object.
3. Open the file.
4. Use the file.
5. Close the file.

## Step 1: Include the header file needed to perform file input/output.

Just as cin and cout require the iostream file to be included in the program, C++ file access requires another header file. The file fstream contains all the definitions necessary for file operations. It is included with the following statement:

```
#include <fstream>
```

## Step 2: Define a file stream object.

The next step in setting up a program to perform file I/O is to define one or more file stream objects. They are called stream objects because a file can be thought of as a stream of data.

The `fstream` header file contains definitions for the data types **`ofstream`**, **`ifstream`**, and **`fstream`**. Before a C++ program can work with a file, it must define an object of one of these data types. The object will be associated with an actual file stored on some secondary storage medium, and the operations that may be performed on that file depend on which of these three data types you pick for the file stream object.

# Step 2: Define a file stream object.

**Table 3-14** File Stream Data Types

File Stream Data Type	Description
<code>ofstream</code>	Output file stream. This data type can be used to open <i>output</i> files and write data to them. If the file does not yet exist, the <code>open</code> operation will automatically create it. If the file already exists, the <code>open</code> operation will destroy it and create a new, empty file of the same name in its place. With the <code>ofstream</code> data type, data may only be copied from variables to the file, but not vice versa.
<code>ifstream</code>	Input file stream. This data type can be used to open existing <i>input</i> files and read data from them into memory. With the <code>ifstream</code> data type, data may only be copied from the file into variables, not but vice versa.
<code>fstream</code>	File stream. This data type can be used to open files, write data to them, and read data from them. With the <code>fstream</code> data type, data may be copied from variables into a file, or from a file into variables.

Here are example statements that define `ofstream` and `ifstream` objects:

```
ofstream outputFile;  
ifstream inputFile;
```

**outputFile and inputFile, could have been named using any legal C++ identifier names.**



## Step 3: Open the file.

Before data can be written to or read from a file, the file must be opened. Outside of the C++ program, a file is identified by its name. Inside a C++ program, however, a file is identified by a stream object. The object and the file name are linked when the file is opened.

Files can be opened through the open function that exists for file stream objects. Assume inputFile is an ifstream object, defined as

```
ifstream inputFile;
```

The following statement uses inputFile to open a file named customer.dat:

```
inputFile.open("customer.dat");           //Open an input file
```

## Step 3: Open the file.

The argument to the open function in this statement is the name of the file. This links the file customer.dat with the stream object inputFile. Until inputFile is associated with another file, any operations performed with it will be carried out on the file customer.dat.

It is also possible to define a file stream object and open a file all in one statement. Here is an example:

```
ifstream inputFile("customer.dat");
```

In our example open statement, the customer.dat file was specified as a simple file name, with no path given.



## Step 3: Open the file.

If the file you want to open is not in the default directory, you will need to specify its location as well as its name.

For example, on a Windows system the following statement opens file C:\data\inventory.dat and associates it with outputFile.

```
outputFile.open("C:\\data\\inventory.dat");
```

Notice the use of the double back slashes in the file's path. This is because, as mentioned earlier, two back slashes are needed to represent one backslash in a string.

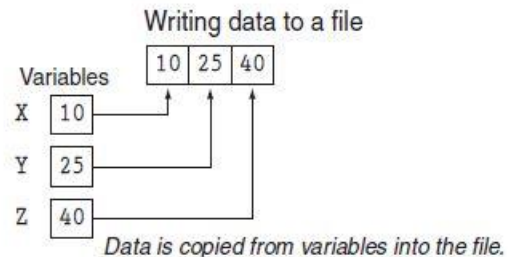
**NOTE: Some systems cannot handle file names that contain spaces. In this case, the entire pathname should be enclosed in an additional set of quotation marks.**

```
outputFile.open("\"C:\\data\\Truck Inventory.dat\"");
```

# Step 4: Use the file.

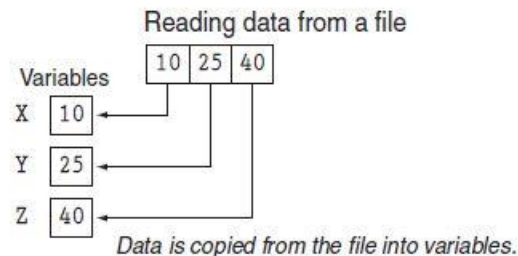
Now that the file is open and can be accessed through a file stream object, you are ready to use it. When a program is actively working with data, the data is located in random-access memory, usually in variables. When data is written into a file, it is copied from variables into the file.

Figure 3-6



When data is read from a file, it is copied from the file into variables. Figure 3-7 illustrates this.

Figure 3-7



# Writing Information to a file

You already know how to use the stream insertion operator (<<) with the cout object to write information to the screen. It can also be used with file stream objects to write information to a file. Assuming outputFile is a file stream object, the following statement demonstrates using the << operator to write a string to a file:

```
outputFile << "I love C++ programming";
```

As you can see, the statement looks like a cout statement, except the file stream object name replaces cout. Here is a statement that writes both a string and the contents of a variable to a file:

```
outputFile << "Price: " << Price;
```

# Opening, Writing to and Closing a File

## Program 3-33

```
1 // This program uses the << operator to write information to a file.
2 #include <iostream>
3 #include <fstream> // Needed to use files
4 using namespace std;
5
6 int main()
7 {
8     ofstream outputFile;
9     outputFile.open("demofile.txt");
10
11     cout << "Now writing information to the file.\n";
12     // Write 3 great names to the file
13     outputFile << "Bach\n";
14     outputFile << "Beethoven\n";
15     outputFile << "Mozart\n";
16
17     // Close the file
18     outputFile.close();
19     cout << "Done.\n";
20     return 0;
21 }
```

1. Header file

2. Object stream

3. Open file

4. Use file

5. Close file

## Program Screen Output

```
Now writing information to the file.
Done.
```

## Output to File demofile.txt

```
Bach
Beethoven
Mozart
```

# Reading Information from a file

The >> operator not only reads user input from the cin object, but it can also be used to read data from a file. Assuming inFile is a file stream object, the following statement shows the >> operator reading data from the file into the variable name:

```
inFile >> name;
```

# Opening, Reading From and Closing a File

## Program 3-34

```
1 // This program uses the >> operator to read information from a file.
2 #include <iostream>
3 #include <fstream> // Needed to use files
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     ifstream inFile; // 2. Object stream
10    string name;
11
12    inFile.open("demofile.txt"); // 3. Open file
13    cout << "Reading information from the file.\n\n";
14
15    inFile >> name; // Read name 1 from the file
16    cout << name << endl; // Display name 1
17
18    inFile >> name; // Read name 2 from the file
19    cout << name << endl; // Display name 2
20
21    inFile >> name; // Read name 3 from the file
22    cout << name << endl; // Display name 3
23
24    inFile.close(); // Close the file
25    cout << "\nDone.\n";
26    return 0;
27 }
```

1. Header file

2. Object stream

3. Open file

4. Use file

5. Close file

## Program Screen Output

```
Reading information from the file.
```

```
Bach
Beethoven
Mozart
```

```
Done.
```

## Step 5: Close the file.

The opposite of opening a file is closing it. Although a program's files are automatically closed when the program shuts down, it is a good programming practice to write statements that explicitly close them. Here are two reasons a program should close files when it is finished using them:

- Most operating systems temporarily store information in a file buffer before it is written to a file. A file buffer is a small holding section of memory that file-bound information is first written to. When the buffer is filled, all the information stored there is written to the file. This technique improves the system's performance. Closing a file causes any unsaved information that may still be held in a buffer to be saved to its file. This means the information will be in the file if you need to read it later in the same program.
- Some operating systems limit the number of files that may be open at one time. When a program keeps open files that are no longer being used, it uses more of the operating system's resources than necessary.

Calling the file stream object's close function closes a file. Here is an example:

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