

# **Chapter 3**

## **Basic Input & Output**

## Basic Output in C++

*cout* is a predefined variable in C++ that indicates you are going to output a stream of characters to an output device. *cout* uses an operator called the *insertion operator* (<<). You will often see << referred to as “put to”. The insertion operator requires two operands. For our purposes the operand on the left will be the *cout* variable and the operand on the right will be an expression. Examples:

The following expression is a literal constant of type `cstring`.

```
cout << "Hello World!";
```

The following expression is a simple arithmetic expression.

```
cout << (num1 + num2) /2;
```

The following expression contains a literal constant of type `cstring` followed by a variable. Note the insertion operator must be used to separate the various items in the output stream..

```
cout << "The average is " << averageAge;
```

A ***general form*** specifies the proper syntax for a given statement. The general form for the output statement is shown below.

GENERAL FORM for `cout`:

```
cout << ExprOrString << ExprOrString ...;
```

Given:

```
const char SCHOOL[20] = "Saddleback";
int num1, num2;
```

Examine the following output statements and what they produce. Assume num1 contains 3 and num2 contains 7.

<u>Statemen</u>	<u>Output</u>
<code>cout &lt;&lt; num1;</code>	3
<code>cout &lt;&lt; num2;</code>	7
<code>cout &lt;&lt; num1 + num2;</code>	10
<code>cout &lt;&lt; "Code =" &lt;&lt; num1;</code>	Code =3
<code>cout &lt;&lt; num1 &lt;&lt; "+" &lt;&lt; num2;</code>	3+7
<code>cout &lt;&lt; num1 &lt;&lt; " + " &lt;&lt; num2;</code>	3 + 7
<code>cout &lt;&lt; "Number is" &lt;&lt; num2;</code>	Number is7
<code>cout &lt;&lt; "Number is" &lt;&lt; " " &lt;&lt; num2;</code>	Number is 7
<code>cout &lt;&lt; "My school is " &lt;&lt; SCHOOL;</code>	My school is Saddleback
<code>cout &lt;&lt; "My school is" &lt;&lt; endl &lt;&lt; SCHOOL;</code>	My school is Saddleback

NOTE: endl (end line) causes the cursor to move to the beginning of the next line. This is one way to get blank lines in your output.

```
cout << num1 << endl << endl << num2;
```

3

7

Why did we get only one blank line?

## Escape Sequences

Another way to cause cout to produce a new line is by inserting an *escape sequence* into a string. When you are dealing with strings this is an easier method. An escape sequence begins with a backslash ( \ ) and is followed by a *control character*. The "\ " and the control character are not printed. Control characters allow you to specify the appearance of your output.

There are many escape sequences – following are a list of a few that we will use in this class.

\n	Newline	Moves the cursor to the next line
\t	Horizontal tab	Moves the cursor to the next tab stop (be careful)
\a	Alarm	Causes the computer to beep
\\	Backslash	Causes a backslash to be printed
\'	Single quote	Causes a single quotation mark to print
\"	Double quote	Causes a double quotation mark to print

Examples:

```
cout << "Stop lights have three colors\n";  
cout << "Red\n";  
cout << "Yellow\n";  
cout << "Green\n";
```

```
cout << "Stop lights have three colors\n";  
cout << "Red\nYellow\nGreen\n";
```

```
cout << "Stop lights have three colors";  
cout << "\nRed\nYellow\nGreen\n";
```

Compare `\n` with `endl`

```
cout << "Stop lights have three colors" << endl;  
cout << "Red" << endl;  
cout << "Yellow" << endl;  
cout << "Green" << endl;
```

Rewrite the above output using a combination of `\n` and `endl`

Write the `cout` statements necessary to produce the following output.

1. Hello World

2. Hello  
World

3. Hello  
World

4. It's a beautiful day.

5. It's a "wacky" world!

## Using Manipulators to Format Output

We have been using the `endl` manipulator to insert a new line in a `cout` statement. We are going to examine the manipulators `setw()`, `setprecision()`, `fixed`, and `showpoint`.

### **setw(n)**

`setw(n)` is used to specify a field for output display where `n` is an integer expression representing the field width. The output is right justified in a field width of `n`. This manipulator may be used with `int`, `float`, and `cstring` data types. It is not valid with data type `char`.

```
int val;  
val = 25;  
cout << "The value is " << setw(5) << val;
```

The value is ---25

The dashes (-) are used to show a blank space in this example and will NOT appear on the actual output.

NOTE: `setw(n)` applies ONLY to the next item in the stream. It will revert to a default of 1 after each use.

### **Formatting floating point values:**

Often times we want to output floating point values in a particular format that is consistent throughout the program. Following are three manipulators to use in this process. To use the following manipulators, you must add the following preprocessor directive

```
#include <iomanip>
```

Use the following declarations for the discussion of `setprecision`, `fixed`, and `showpoint`.

```
float val = 52.1234;  
float val2 = 5;
```

### **setprecision(n)**

This manipulator allows us to control the number of digits displayed on output. **Precision refers to the total number of digits to the right and left of the decimal point.** The default precision on most systems is 6 digits. Argument `n` is used to indicate the number of digits displayed. Unlike `setw()`, once the precision is set, it remains at that value until changed in the program.

```
cout << val << ' ' << val2;           output 52.1234 5  
cout << setprecision(2) << val << ' ' << val2;   output 52 5  
cout << setprecision(4) << val << ' ' << val2;   output 52.12 5
```

### **fixed**

If you wish to ensure that all floating point values are displayed in fixed decimal format, include the following statement in your program. This causes the precision to refer to the digits to the right of the decimal point.

```
cout << fixed << val << ' ' << val2;           output 52.123400 5.000000  
cout << fixed << setprecision(2) << val << ' ' << val2;   output 52.12 5.00
```

The fixed format will remain until it is disabled with the statement

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

### **showpoint**

Showpoint is used when the decimal part of a number is 0. Many compilers will not display the decimal point and the fixed number of digits to the right unless you instruct it to do so. This is accomplished using

```
cout << showpoint << val << ' ' << val2;           output 52.1234 5.000000  
cout << showpoint << setprecision(2) << val << ' ' << val2; output 52. 5.0
```

The manipulators may be specified in the same statement.

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

Sample program:

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

void main(void)
{
    float n1 = 3;
    float n2 = 3.14159;
    float n3 = 12.75;

    cout << fixed << setprecision(2);
    cout << setw(10) << n1 << endl;
    cout << setw(8) << n2 << endl;
    cout << n3 << endl;
    cout << setw(10) << setprecision(5) << n3;
}
```

Output:

```
        3.00
       3.14
12.75
12.75000
```

## Basic Input in C++

*cin* is a predefined variable in C++ that indicates you are going to input a stream of characters from an input device (*cin* is associated with the standard input device which is the keyboard). *cin* uses an operator called the *extraction operator* (>>). You will often see >> referred to as “get from”. The extraction operator requires two operands. For our purposes the operand on the left will be the *cin* variable and the operand on the right will be a variable (char, int, float etc.). The general form for the input statement is shown below:

**GENERAL FORM:**

```
cin >> variable >> variable ...;
```

**NOTE:** Unlike the output statement which may contain constants, variables or more complicated expressions, the only items that may be specified in an input statement are the names of one or more variables. The purpose of an input statement is to fill a memory location. The only memory locations that may be modified while the program is running are those named by variables.

Examples:

```
cin >> length;  
cin >> width;
```

or

```
cin >> length >> width;
```

Although you may use a sequence of extraction operators in the same input statement, we will typically prompt for one item at a time using *cout* and read into variables one at a time using *cin*.

```
cout << "Enter the length: ";  
cin >> length;  
cout << "Enter the width: ";  
cin >> width;
```

It is necessary that the input entered at the keyboard match the data type of the variable specified in the input statement. Variables of type `int` require an integer literal constant. Variables of type `float` may receive either an integer or a floating point literal constant. If an integer is entered at the keyboard, the compiler will convert it to a float and store it appropriately. Variables of type `char` require a single printable character other than a blank. While the program will continue to run when invalid data is entered, the results produced may be erroneous.

When a sequence of extraction operators appear in the same input statement, the various data items may be delimited with *whitespace characters*. Whitespace characters include blanks and certain nonprintable characters such as the end-of-line character and the tab character.

It is very important that you carefully enter data from the keyboard. Entering data of a type not compatible with the input variable specified in the `cin` statement can cause very unexpected results.

Given the following declarations:

```
int n1, n2, n3;  
float aFloatNum;  
char theChar;
```

For each input statement and the indicated data from the keyboard, indicate the contents of the memory locations after the input statement has been executed.

<u>Input Statement</u>	<u>Data</u>
cin >> n1;	12
cin >> n1 >> n2;	8 16
cin >> theChar >> aFloatNum;	X 3.75
cin >> n1 >> n2 >> n3;	12 24 36
cin >> aFloatNum >> theChar >> n1;	5X3
cin >> n1 >> n2 >> n3	5 6

## cin and c-strings

Given the following:

```
char userString[5];  
  
cout << "Enter a string: ";  
cin >> userString;  
cout << "\n\n" << userString;
```

A run of the program would produce the following (user input is in italics)

Enter a string: *abcdefghij*

abcdefghij

**NOTE: In standard C++ the entire c-string is placed in memory and the extra characters will spill over into adjacent memory locations. ANYTHING STORED IN THESE LOCATIONS WOULD BE REPLACED/DESTROYED. Some new compilers however treat this as an error and halt the program. Test this on your compiler.**

There are formatting tools available to cin that may be used to control the number of characters read. Either the **setw** manipulator or the **cin.width** member function may be used. Both produce the same result.

```
char userString[5];

cout << "Enter a string: ";
cin >> setw(5) >> userString;
cout << "\n\n" << userString;
```

OR

```
cout << "Enter a string: ";
cin.width(5);
cin >> userString;
cout << "\n\n" << userString;
```

A run of the program would produce the following output (user input in italics)

Enter a string: *abcdefghij*

abcd

cstrings are terminated with a null terminator (\0) in C++ so the width specified in the declaration section must be one more than the total number of characters you wish to store.

Given the following:

```
char fullName[25];      // cstring to hold the name

cout << "Enter your full name: ";
cin >> fullName;
cout << "\n\nYour full name is " << fullName;
```

A run of the program would produce the following (user input is in italics)

Enter your full name: *Peggy Watkins*

Your full name is Peggy

**NOTE: cin stops reading input when it encounters a whitespace character. Whitespace characters include the Enter key, space, and tab.**

cin has a member function called **getline** that may be used to read a string containing blank spaces or tabs.

```
cin.getline(stringName, stringWidth);
```

This function has two arguments, the name of the string variable and the number of characters declared. Up to one less than the string width characters will be read into the array and the null terminator will automatically be placed at the end of the string.

```
cout << "Enter your full name: ";
cin .getline(fullName,25);
cout << "\n\nYour full name is " << fullName;
```

A run of the program would produce the following (user input is in italics)

Enter your full name: *Peggy Watkins*

Your full name is Peggy Watkins

If C++ ignores all leading *whitespace* when it is looking for a character, how do you make it possible to program features such as

Press enter to continue

It is not possible to use the >> operator to accept the Enter key as character input. cin has another member function called **get** that is useful here. This function reads a single character including whitespace characters.

```
{  
    char theCh;  
  
    cout << "Press enter to continue ";  
    cin.get(theCh);  
    cout << "Program continues" << endl;  
  
}
```

NOTE: BE VERY CAREFUL ABOUT MIXING cin >> AND cin.get IN THE SAME PROGRAM!! WHAT WILL HAPPEN HERE??

```
int num;  
char theCh;
```

```
cout << "Enter a number ";  
cin >> num;  
cout << "Enter a character ";  
cin.get(theCh);  
cout << "Thank you"
```

## "Flushing" the buffer

Often we need to be able to “flush” the buffer and skip (read and discard) characters in the input stream. This is done using `cin.ignore()`.

### **`cin.ignore(int expression, char value)`**

Example:

```
cin.ignore(100, '\n');
```

This statement would ignore the next 100 characters (or skip characters) until a newline is read – whichever comes first.

Example:

```
cin >> num;  
cin.ignore(1, '\n');  
cin.get(theCh);  
cout << "\n\nprogram over";
```

If the user enters 3 followed by a blank space followed by the enter key what would happen?

## Comparing c-strings

It is not possible to compare c-strings using relational operators. Relational operators may compare numeric and char values only. C++ stores a c-string in an array. When an array is accessed, you are actually working with the *address* of the beginning of the array. When you make the following comparison

```
if (stringOne == stringTwo)
```

you are really comparing two addresses and they will never be the same. String comparisons are done in C++ using a library function called

```
strcmp(string1, string2)
```

This function compares the contents of string1 with the contents of string2 and returns

- 0 if the strings are the same
- an integer < 0 if string1 is lower on the ASCII chart than string2
- an integer > 0 if string1 is higher on the ASCII chart than string2

```
char stringOne[10];  
char stringTwo[10];
```

```
cout << "Enter the first string: ";  
cin >> stringOne;  
cout << "Enter the second string: ";  
cin >> stringTwo;
```

```
if(strcmp(stringOne, stringTwo) == 0)  
{  
    cout << "The strings are the same";  
}
```

NOTE: Some compilers require you to include the header file **string.h** in your program if you wish to use the strcmp function.

## Review

When using *cin*, C++ reads data from the \_\_\_\_\_.  
The program waits for keyboard input only when \_\_\_\_\_  
\_\_\_\_\_

When reading data using *cin*, reading stops as soon as any \_\_\_\_\_  
character is encountered. These characters include \_\_\_\_\_,  
\_\_\_\_\_, and \_\_\_\_\_.

Describe what happens if the user enters more characters than the declared  
size of a c-string.

To limit the number of characters read and stored to a c-string variable, use  
\_\_\_\_\_ or \_\_\_\_\_.

To read a c-string that contains whitespace characters use \_\_\_\_\_

To read a char (including whitespace characters) use \_\_\_\_\_

To "flush" the buffer of unwanted characters use \_\_\_\_\_

The name of a c-string variable is really a(n) \_\_\_\_\_.  
For this reason, you may only compare two c-string variables using  
\_\_\_\_\_