Chapter 10
The C++ I/O System Basics

- Formatting of data and overload the << (insertion) and >> (extraction) I/O operators to be used with classes.
- Creation of I/O functions called manipulators that can make our program more efficient.

Old vs. modern C++ I/O:

- There are currently two versions of the C++ object-oriented I/O library in use: older and newer.

Older	Newer
Based upon original specifications of C++.	Defined by standard C++
Old I/O library is supported by header file <iostream.h>	New I/O library is supported by the header <iostream>. It is an improved and updated version of old one. New I/O library contains a few additional features and defines some new data types. It is superset of the old one.
I/O library was in global namespace.	The new style library is in the std namespace.

C++ streams:

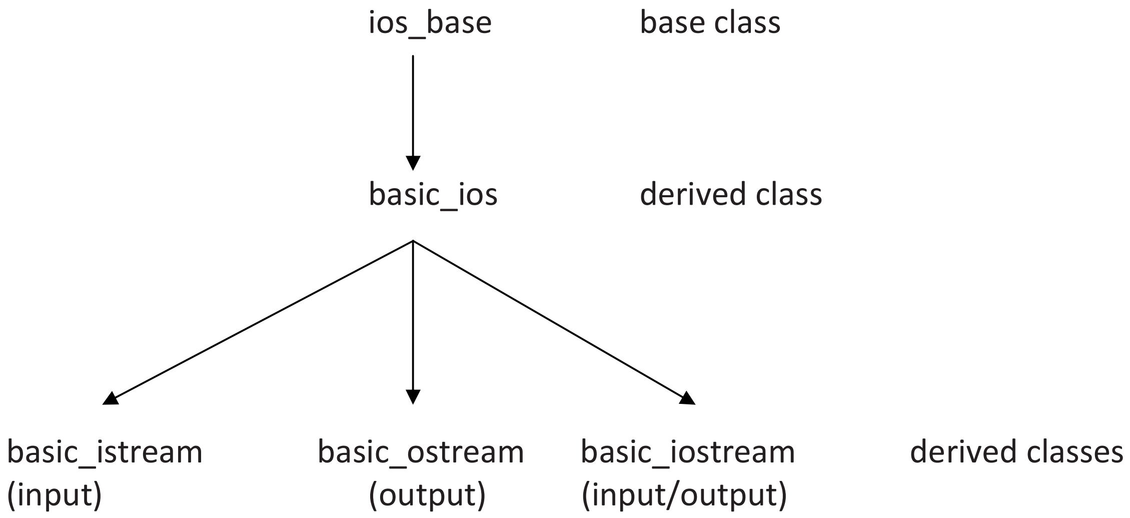
- Stream is a sequence of bytes. It acts as either a source from which the input data can be obtained or as a destination to which the output data can be sent.
- A stream is a logical device that either produces or consumes information and is linked to a physical device by the I/O system.
- All streams behave in the same way even though the actual physical devices they are connected to may differ substantially.

The C++ stream classes:

- The I/O classes begin with a system of template classes.
- Standard C++ creates two specializations of the I/O template classes:
  - One for 8-bit characters.
  - Another for wide characters.
- The C++ I/O system is built upon two related but different template class hierarchies.
  1. First is derived from the low-level I/O class called **basic_streambuf**.

    - This class supplies the basic, low-level input and output operations, and provides the underlying support for the entire C++ I/O system.

ii. The class hierarchy that will most commonly be working with is derived from **basic_ios**.
    - This is high-level I/O class that provides formatting, error checking and status information related to stream I/O.



- These classes are used to create streams.

Template class	Character based class	Wide character based class
basic_streambuf	streambuf	wstreambuf
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream
basic_fstream	fstream	wfstream
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream

- The ios class contains many member functions and variables that control or monitor the fundamental operation of a stream.
- The inclusion of <iostream> in the program, then we have access to the important class (ios).

C++ predefined streams:

- When a C++ program begins execution, four built-in streams are automatically opened. They are:

Stream	Meaning	Default device
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error output	Screen
clog	Buffered version of cerr	Screen

C++ Streams	--------	cin	cout	cerr
corresponds to C's	--------	stdin	stdout	stderr

- By default, the standard streams are used to communicate with the console.
- Wide characters are of type wchar_t and are generally 16-bit quantities.
- Wide characters are used to hold the large character sets associated with some human languages.

Formatted I/O:

- The C++ I/O system allows to format I/O operations.
- For example, set a field width, specify a number base, or determine how many digits after the decimal point will be displayed.
- There are two related but conceptually different ways to format data.
    i. Directly access members of the **ios** class. Specifically, set various format status flags defined inside the **ios** class or call various **ios** member functions.
    ii. Use special functions called *manipulators* that can be included as part of an I/O expression.

Formatting using ios members:

- Each stream has associated with it a set of format flags that control the way information is formatted.
- The **ios** class declares a bitmask enumeration called **fmtflags** in which the following values are defined.

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

Related flags:

flag value	equivalent to
adjustfield (adjustment)	left | right | internal
base field (numerical base)	dec | oct | hex
floatfield (float format)	scientific | fixed

- When the **left** flag is set, output is left justified. When the **right** flag is set, output is right justified. When the **internal** flag is set, a numerical value is padded to fill a field by inserting spaces between any sign or base character. If none of these flags are set, output is right justified by default.

- By default, numerical values are output in decimal. It is possible to change number base. Setting **oct** flag causes output to be displayed in octal. Setting the **hex** flag causes output to be displayed in hexadecimal. To return output to decimal, set the **dec** flag.

- By setting the **scientific** flag, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation.

Independent flags:

- When **boolalpha** is set, booleans can be input or output using the keywords **true** and **false**.
- Setting **showbase** causes the base of numeric values to be shown. For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F.
- Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output.
- Setting **showpos** causes a leading plus sign to be displayed before positive values.
- When the **skipws** flag is set, leading whitespace characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When skipws is cleared, white-space characters are not discarded.
- When **unitbuf** is set, the buffer is flushed after each insertion operation.
- By default, when scientific notation is displayed, the **e** is in lowercase. Also, when a hexadecimal value is displayed, the **x** is in lowercase. When **uppercase** is set, these characters are displayed in uppercase.

Setting the format flags:

- To set a flag, use the setf() function. This function is a member f **ios**. Its most common form is:

  fmtflags setf(fmtflags *flags*);

- This function returns the previous settings of the format flags and turns on those flags specified by *flags*. For example, to turn on the **showpos** flag, you can use this statement:

  stream.setf(ios::showpos);

- Here, *stream* is the stream you wish to affect. Notice the use of **ios::** to qualify **showpos**. Since **showpos** is an enumerated constant defined by the **ios** class, it must be qualified by **ios** when it is used.
- The following program displays the value 100 with the **showpos** and **showpoint** flags turned on.

  Example program:

```
#include<iostream>
using namespace std;
main()
{
        cout.setf(ios::showpoint);
        cout.setf(ios::showpos);
        cout<<100.000<<endl;
}
Output:
+100.000
```

- Instead of making multiple calls to setf(), simply OR together the values of the flags.

  Example program:

```
#include<iostream>
using namespace std;
main()
{
        cout.setf(ios::showpoint|ios::showpos);
        cout<<100.000<<endl;
}
Output:
+100.000
```

Clearing Format Flags:

- The complement of **setf( )** is **unsetf( )**. This member function of **ios** is used to clear one or more format flags. Its general form is

  void unsetf(fmtflags *flags*);

- The flags specified by *flags* are cleared.
- The following program illustrates **unsetf()**. It first sets both the **uppercase** and **scientific** flags. It then outputs 100.12 in scientific notation. In this case, the "E" used in the scientific notation is in uppercase.
- Next, it clears the **uppercase** flag and again outputs 100.12 in scientific notation, using a lowercase "e."

  Example program:

```
#include <iostream>
#include <iomanip>
using namespace std;
main ()
{
   float n;
   cout<<"Enter any float value\n";
   cin>>n;
   cout.setf(ios::scientific|ios::uppercase);
   cout<<n<<endl;
   cout.unsetf(ios::uppercase);
   cout<<n<< endl;
}
Output:
Enter any float value
100.12
1.001200E+02
1.001200e+02
```

An Overloaded Form of setf():

- There is an overloaded form of **setf()** that takes this general form:

fmtflags setf(fmtflags *flags1*, fmtflags *flags2*);

- In this version, only the flags specified by *flags2* are affected. They are first cleared and then set according to the flags specified by *flags1*. Note that even if *flags1* contains other flags, only those specified by *flags2* will be affected. The previous flags setting is returned.

  Example program:

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

main ()
{
        cout.setf(ios::showbase|ios::uppercase);
        cout.setf(ios::hex,ios::basefield);
        cout<<100<<endl;
}
Output:
0X64
```

- The most common use of the two-parameter form of **setf( )** is when setting the number base, justification, and format flags.
- References to the **oct**, **dec**, and **hex** fields can collectively be referred to as **basefield**.
- Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**.
- Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.
- Since the flags that comprise these groupings are mutually exclusive, you may need to turn off one flag when setting another. For example,
- To output in hexadecimal, some implementations require that the other number base flags be turned off in addition to turning on the **hex** flag. This is most easily accomplished using the two-parameter form of **setf( )**.

Examining the Formatting Flags:

- There will be times when you only want to know the current format settings but not alter any.
- To accomplish this goal, **ios** includes the member function **flags( )**, which simply returns the current setting of each format flag. Its prototype is:

  fmtflags flags( );

Example program:

```
#include<iostream>
using namespace std;
void showflags()
{
        ios::fmtflags f;
        long i;
        f=cout.flags();
        for(i=0x4000;i;i=i>>1)
        if(i&f)cout<<" 1 ";
        else
        cout<<" 0 ";
        cout<<endl;

}
main()
{
        showflags();
        cout.setf(ios::right|ios::showpoint|ios::fixed);
        showflags();
}
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0
0 0 1 0 1 0 0 1 0 0 0 0 1 1 0
```

Setting All Flags:

- The **flags()** function has a second form that allows you to set all format flags associated with a stream. The prototype for this version of **flags( )** is shown here:

  fmtflags flags(fmtflags *f*);

- When you use this version, the bit pattern found in *f* is used to set the format flags associated with the stream. Thus, all format flags are affected. The function returns the previous settings.

Using width(), precision(), and fill():

- In addition to the formatting flags, there are three member functions defined by **ios** that set these format parameters: the field width, the precision, and the fill character.
- The functions that do these things are **width()**, **precision()**, and **fill()**, respectively.

width():

- By default, when a value is output, it occupies only as much space as the number of characters it takes to display it.
- However, you can specify a minimum field width by using the **width()** function. Its prototype is:

  streamsize width(streamsize *w*);

- Here, *w* becomes the field width, and the previous field width is returned. In some implementations, the field width must be set before each output.
- If it is not, the default field width is used. The **streamsize** type is defined as some form of integer by the compiler.
- After you set a minimum field width, when a value uses less than the specified width, the field will be padded with the current fill character (space, by default) to reach the field width.
- If the size of the value exceeds the minimum field width, the field will be overrun. No values are truncated.

precision():

- When outputting floating-point values, you can determine the number of digits of precision by using the **precision()** function. Its prototype is:
  streamsize precision(streamsize *p*);

- Here, the precision is set to *p*, and the old value is returned. The default precision is 6. In some implementations, the precision must be set before each floating-point output. If it is not, then the default precision will be used.

fill():

- By default, when a field needs to be filled, it is filled with spaces. You can specify the fill character by using the **fill()** function. Its prototype is
  char fill(char *ch*);

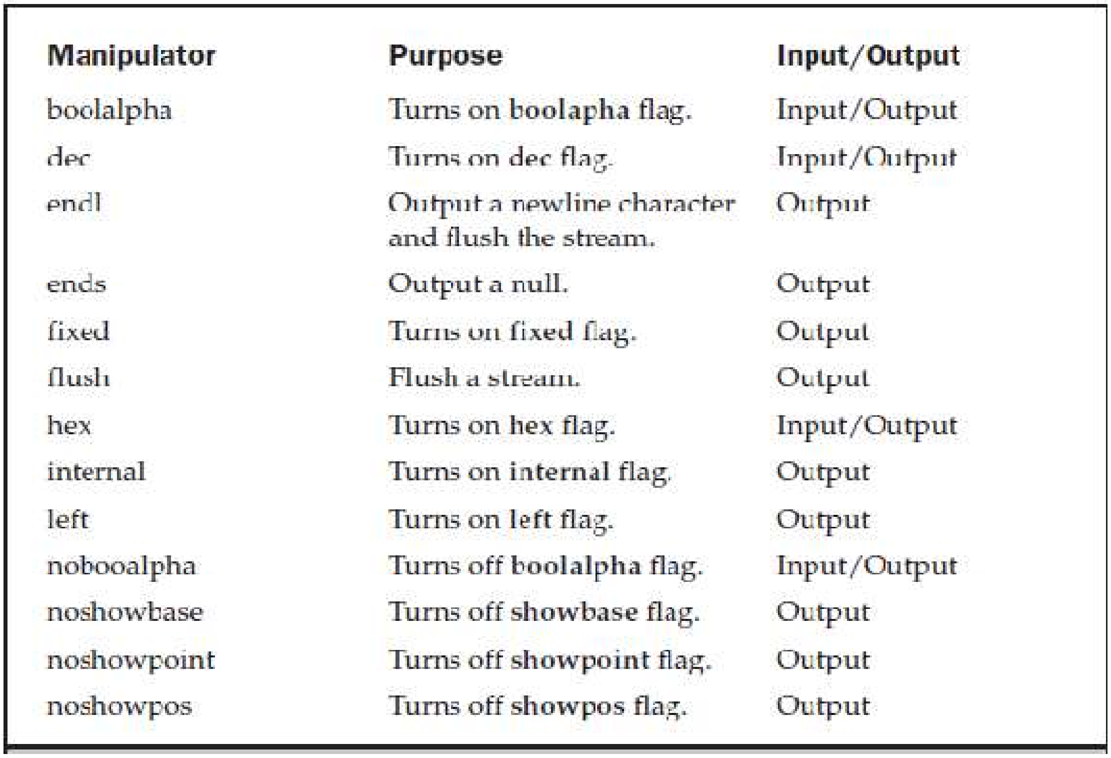
- After a call to **fill( )**, *ch* becomes the new fill character, and the old one is returned.

  Example program:

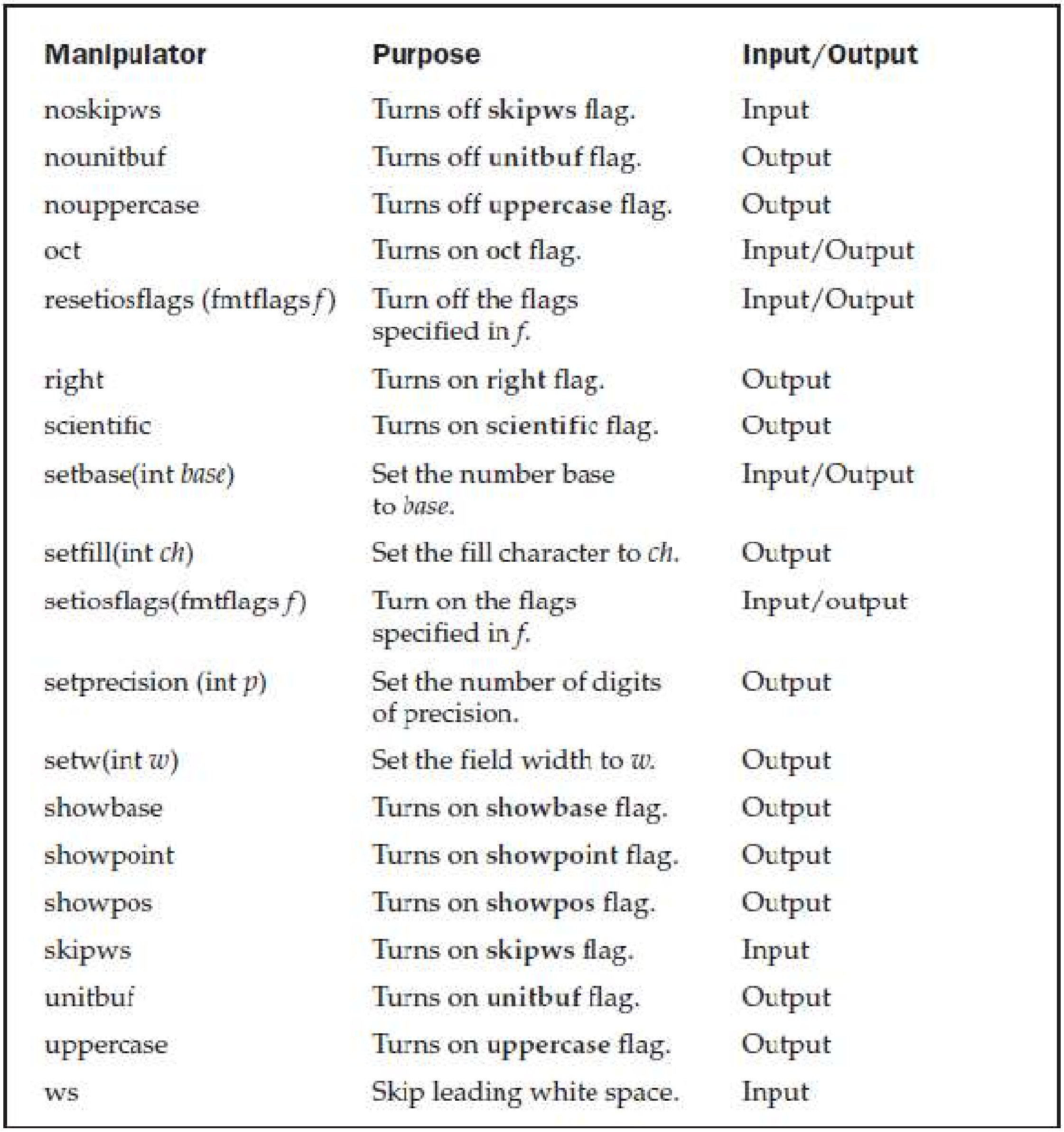
```
#include <iostream>
using namespace std;
main()
{
    cout.precision(4) ;
    cout.width(10);
    cout<<10.12345 << "\n";
    cout.fill('*');
    cout.width(10);
    cout<<10.12345 << "\n";
    cout.width(10);
    cout <<"C++"<<"\n";
    cout.width(10);
    cout.setf(ios::left);
    cout<<10.12345<<endl;
}
Output:
     10.12
*****10.12
*******C++
10.12*****
```

Using Manipulators to Format I/O:

- The second way you can alter the format parameters of a stream is through the use of special functions called *manipulators* that can be included in an I/O expression. The standard manipulators are shown in the below table:



Manipulator	Purpose	Input/Output
boolalpha	Turns on boolapha flag.	Input/Output
dec	Turns on dec flag.	Input/Output
endl	Output a newline character and flush the stream.	Output
ends	Output a null.	Output
fixed	Turns on fixed flag.	Output
flush	Flush a stream.	Output
hex	Turns on hex flag.	Input/Output
internal	Turns on internal flag.	Output
left	Turns on left flag.	Output
nobooalpha	Turns off boolalpha flag.	Input/Output
noshowbase	Turns off showbase flag.	Output
noshowpoint	Turns off showpoint flag.	Output
noshowpos	Turns off showpos flag.	Output



Manipulator	Purpose	Input/Output
noskipws	Turns off skipws flag.	Input
nounitbuf	Turns off unitbuf flag.	Output
nouppercase	Turns off uppercase flag.	Output
oct	Turns on oct flag.	Input/Output
resetiosflags (fmtflags *f*)	Turn off the flags specified in *f*.	Input/Output
right	Turns on right flag.	Output
scientific	Turns on scientific flag.	Output
setbase(int *base*)	Set the number base to *base*.	Input/Output
setfill(int *ch*)	Set the fill character to *ch*.	Output
setiosflags(fmtflags *f*)	Turn on the flags specified in *f*.	Input/output
setprecision (int *p*)	Set the number of digits of precision.	Output
setw(int *w*)	Set the field width to *w*.	Output
showbase	Turns on showbase flag.	Output
showpoint	Turns on showpoint flag.	Output
showpos	Turns on showpos flag.	Output
skipws	Turns on skipws flag.	Input
unitbuf	Turns on unitbuf flag.	Output
uppercase	Turns on uppercase flag.	Output
ws	Skip leading white space.	Input

- To access manipulators that take parameters (such as **setw( )**), you must include **<iomanip>** in your program.

```
//boolalpha
#include<iostream>
using namespace std;
main()
{
        bool b;
        cout<<"enter boolean value(true/false)";
        cin>>boolalpha>>b;
        cout<<boolalpha<<b<<endl;
}
```

Output:

```
enter boolean value(true/false)true
true
```

```
//base field : hex,dec,oct
#include<iostream>
using namespace std;
main()
{
        int n;
        cout<<"enter n\n";
        cin>>n;
        cout<<hex<<showbase<<uppercase<<n<<endl;
        cout<<oct<<n<<endl;
        cout<<dec<<n<<endl;
}
```

Output:

```
enter n
100
0X64
0144
100
```

```
//setw, setfill
#include<iostream>
#include<iomanip>
using namespace std;
main()
{
        char str[20];
        cout<<"enter str\n";
        cin>>str;
        cout<<setw(20)<< str<<endl;
        cout<<setw(20)<<setfill('*')<< str<<endl;
}
```

Output:

```
enter str
Anuradha
            Anuradha
************Anuradha
```

```
//setiosflag, resetiosflags
#include<iostream>
#include<iomanip>
```

```
using namespace std;
main()
{
        float f;
        cout<<"enter float value\n";
        cin>>f;
        cout<<f<<endl;
        cout<<setiosflags(ios::showpoint)<<f<<endl;
        cout<<resetiosflags(ios::showpoint)<<f<<endl;
}
```

Output:

```
enter float value
100
100
100.000
100
```

```
//fixed, scientific, setprecision
#include<iostream>
#include<iomanip>
using namespace std;
main()
{
        float f;
        cout<<"enter float value\n";
        cin>>f;
        cout<<f<<endl;
        cout<<scientific<<f<<endl;
        cout<<fixed<<setprecision(2)<<f<<endl;
}
```

Output:

```
enter float value
100.12
100.12
1.001200e+02
100.12
```

```
//setbase 16,10,8
#include<iostream>
#include<iomanip>
using namespace std;
main()
{
        int n;
        cout<<"enter octal num\n";
        cin>>setbase(8)>>n;
        cout<<n<<endl;
        cin>>setbase(16)>>n;
        cout<<n<<endl;
        cin>>setbase(10)>>n;
        cout<<n<<endl;
}
```

Output:

```
enter octal num
144
100
64
100
100
100
```

```
//showbase
#include<iostream>
#include<iomanip>
using namespace std;
main()
{
        int n=100;
        cout<<showbase<<hex<<uppercase<<n<<endl;
}
```

Output:

```
0X64
```

```
//showpos, noshowpos
#include<iostream>
#include<iomanip>
using namespace std;
main()
{
        int n;
        cout<<"Enter n value\n";
        cin>>n;
        cout<<showpos<<n<<endl;
        cout<<noshowpos<<n<<endl;
}
```

Output:

```
Enter n value
89
+89
89
```

Creating Your Own Manipulator Functions:

- Customization of C++'s I/O system by creating your own manipulator functions.
- Custom manipulators are important for two main reasons.
- First, you can consolidate a sequence of several separate I/O operations into one manipulator.
- For example, you might use a manipulator to send control codes to a special type of printer or to an optical recognition system. Custom manipulators are a feature of C++ that supports OOP, but also can benefit programs that are not object oriented.
- As you know, there are two basic types of manipulators: those that **operate on input streams** and those that **operate on output streams**.
- The creation of parameter less manipulators is straightforward and the same for all compilers.

  All parameter less **manipulator output functions** have this skeleton:

```
ostream &manip-name(ostream &stream)
{
        // your code here
        return stream;
}
```

Example program:

```
#include<iostream>
using namespace std;
ostream& setoct(ostream &s)
{
        s.setf(ios::showbase);
        s.setf(ios::oct,ios::basefield);
        return s;
}
main()
{
        int n;
        cout<<"enter n\n";
        cin>>n;
        cout<<"decimal n: "<<n<<setoct<<"\nOctal n: "<<n<<endl;
}
```

```
Output:
enter n
100
decimal n: 100
Octal n: 0144
```

Another example program:

```
#include <iostream>
#include <iomanip>
using namespace std;
ostream &sethex(ostream &s)
{
        s.setf(ios::showbase);
        s.setf(ios::hex, ios::basefield);
        return s;
}
main()
{
        cout<<100<<" "<<sethex<<100<<endl;
}
```

```
Output:
100 0x64
```

- Using an output manipulator is particularly useful for sending special codes to a device.
- For example, a printer may be able to accept various codes that change the type size or font, or that position the print head in a special location. If these adjustments are going to be made frequently, they are perfect candidates for a manipulator.

All parameterless **input manipulator functions** have this skeleton:

```
istream &manip-name(istream &stream)
{
        // your code here
        return stream;
}
```

- An **input manipulator receives a reference to the stream for which it was invoked**. This stream must be returned by the manipulator.

Example program:

```
#include<iostream>
using namespace std;
istream& getval(istream &s)
{
        cout<<"Enter boolean value(true/false)\n";
        s>>boolalpha;
        return s;
}
main()
{
        int i;
        bool b;
        cout<<"Enter 3 boolean values\n";
        for(i=0;i<3;i++)
        {
                cin>>getval>>b;
                cout<<boolalpha<<b<<endl;
        }
}
```

Output:

```
Enter 3 boolean values
Enter boolean value(true/false)
true
true
Enter boolean value(true/false)
false
false
Enter boolean value(true/false)
true
true
```