

MODULE 18
C++ STREAM FORMATTED I/O

MODULE 5
C FORMATTED I/O

My Training Period: hours

Abilities

- To understand and use various member functions for C++ formatted I/O.
- To understand and use various stream manipulators for C++ formatted I/O.

18.1 iostream Library

- In Module 5 you have learned the formatted I/O in C by calling various standard functions. In this Module we will discuss how this formatted I/O implemented in C++ by using member functions and stream manipulators.
- If you have completed this [Tutorial #3](#) until [Module 17](#), you should be familiar with class object. In C++ we will deal a lot with classes. It is readily available for us to use.
- We will only discuss the formatted I/O here, for file I/O and some of the member functions mentioned in this Module, will be presented in another Module. The discussion here will be straight to the point because some of the terms used in this Module have been discussed extensively in [Module 5](#).
- The header files used for formatted I/O in C++ are:

Header file	Brief description
iostream.h	Provide basic information required for all stream I/O operation such as cin, cout, cerr and clog correspond to standard input stream, standard output stream, and standard unbuffered and buffered error streams respectively.
iomanip.h	Contains information useful for performing formatted I/O with parameterized stream manipulation.
fstream.h	Contains information for user controlled file processing operations.
strstream.h	Contains information for performing in-memory formatting or in-core formatting. This resembles file processing, but the I/O operation is performed to and from character arrays rather than files.
stdiostrem.h	Contains information for program that mixes the C and C++ styles of I/O.

Table 18.1: iostream library

- The compilers that fully comply with the C++ standard that use the template based header files won't need the .h extension. Please refer to [Module 23](#) for more information.
- The iostream class hierarchy is shown below. From the base class ios, we have a derived class:

Class	Brief description
istream	Class for stream input operation.
ostream	Class for stream output operation.

Table 18.2: ios derived classes

- So, iostream support both stream input and output. The class hierarchy is shown below.

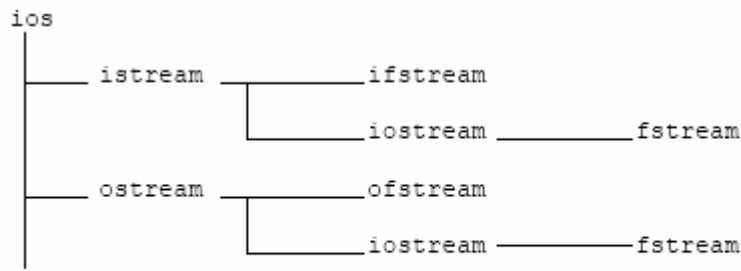


Figure 18.1: ios class hierarchy portion.

18.2 Left and Right Shift Operators

- We have used these operators in most of the Modules in this Tutorial for C++ codes.
- The **left shift operator** (<<) is overloaded to designate stream output and is called **stream insertion operator**.
- The **right shift operator** (>>) is overloaded to designate stream input and is called **stream extraction operator**.
- These operators used with the **standard stream object** (and with other user defined stream objects) is listed below:

Operators	Brief description
cin	Object of istream class, connected to the standard input device , normally the keyboard.
cout	Object of ostream class, connected to standard output device , normally the display screen.
cerr	Object of the ostream class connected to standard error device . This is unbuffered output, so each insertion to cerr causes its output to appear immediately.
clog	Same as cerr but outputs to clog are buffered.

Table 18.3: ostream operators

- For file processing C++ uses (will be discussed in another Module) the following classes:

Class	Brief description
ifstream	To perform file input operations.
ofstream	For file output operation.
fstream	For file input/output operations.

Table 18.4: File input/output classes

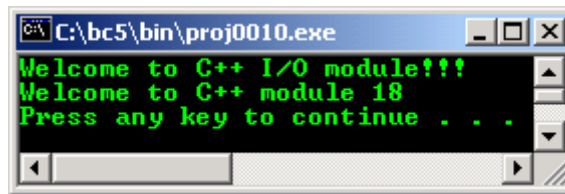
- Stream output program example:

```

//string output using <<
#include <stdlib.h>
#include <iostream.h>

void main(void)
{
    cout<<"Welcome to C++ I/O module!!!"<<endl;
    cout<<"Welcome to ";
    cout<<"C++ module 18"<<endl;
    //endl is end line stream manipulator
    //issue a new line character and flushes the output buffer
    //output buffer may be flushed by cout<<flush; command
    system("pause");
}
  
```

Output :

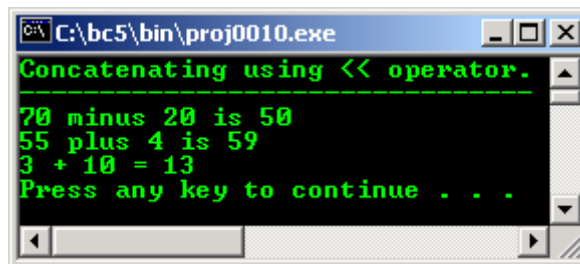


```
//concatenating <<
#include <stdlib.h>
//for system(), if compiled in some compiler
//such as Visual Studio, no need this stdlib.h
#include <iostream.h>

void main(void)
{
    int p = 3, q = 10;

    cout << "Concatenating using << operator.\n"
         << "-----" << endl;
    cout << "70 minus 20 is "<<(70 - 20)<<endl;
    cout << "55 plus 4 is "<<(55 + 4)<<endl;
    cout <<p<<" + "<<q<<" = "<<(p+q)<<endl;
    system("pause");
}
```

Output:



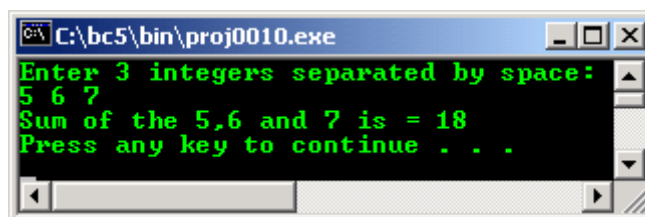
- Stream input program example:

```
#include <stdlib.h>
#include <iostream.h>

void main(void)
{
    int p, q, r;

    cout << "Enter 3 integers separated by space: \n";
    cin >> p >> q >> r;
    //the >> operator skips whitespace characters such as tabs,
    //blank space and newlines. When eof is encountered, zero (false)
    //is returned.
    cout << "Sum of the "<<p<< ", "<<q<< " and "<<r<< " is = "<<(p+q+r)<<endl;
    system("pause");
}
```

Output:



18.3 get() and getline() Member Functions of Stream Input

- For the `get()` function, we have three versions.

1. `get()` without any arguments, input one character from the designated streams including whitespace and returns this character as the value of the function call. It will return EOF when end of file on the stream is encountered. For example:

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

2. `get()` with a character argument, inputs the next character from the input stream including whitespace. It return false when end of file is encountered while returns a reference to the `istream` object for which the `get` member function is being invoked. For example:

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

3. `get()` with three arguments, a character array, a size limit and a delimiter (default value `'\n'`). It reads characters from the input stream, up to one less than the specified maximum number of characters and terminates or terminates as soon as the delimiter is read. For example:

```
char namevar[30];
...
cin.get(namevar, 30);
//get up to 29 characters and inserts null
//at the end of the string stored in variable
//namevar. If a delimiter is found,
//the read terminates. The delimiter
//is left in the stream, not stored
//in the array.
```

4. `getline()` operates like the third `get()` and insert a null character after the line in the character array. It removes the delimiter from the stream, but does not store it in the character array.

- Program examples:

```
//End of file controls depend on system
//Ctrl-z followed by return key - IBM PC
//Ctrl-d - UNIX and MAC

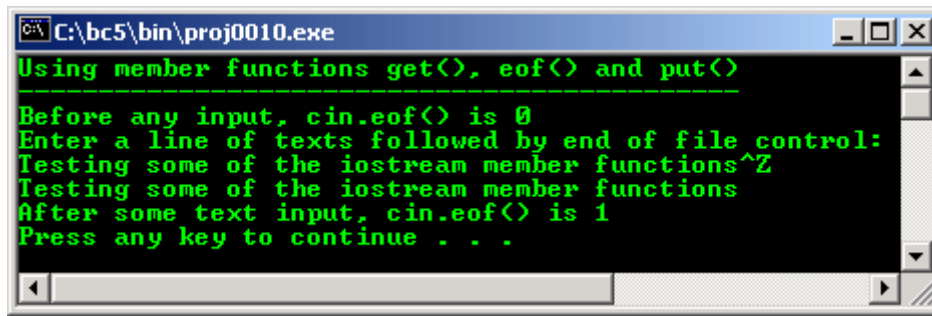
#include <stdlib.h>
#include <iostream.h>

void main(void)
{
    char p;

    cout <<"Using member functions get(), eof() and put()\n"
         <<"-----" <<endl;
    cout <<"Before any input, cin.eof() is " <<cin.eof() <<endl;
    cout <<"Enter a line of texts followed by end of file control: " <<endl;

    while((p = cin.get()) !=EOF)
        cout.put(p);
    cout <<"\nAfter some text input, cin.eof() is " <<cin.eof() <<endl;
    system("pause");
}
```

Output:



```
//Another get() version
#include <stdlib.h>
#include <iostream.h>

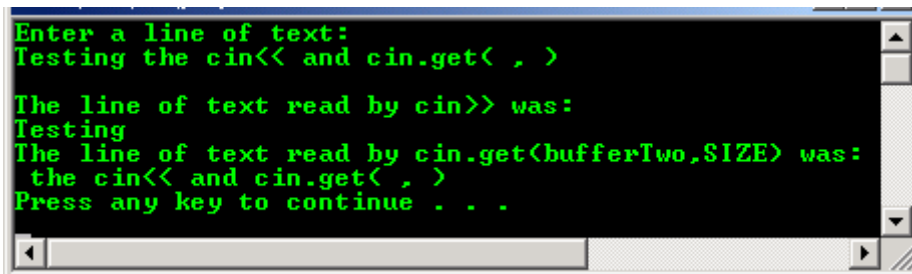
const int SIZE = 100;

void main(void)
{
    char bufferOne[SIZE], bufferTwo[SIZE];

    cout <<"Enter a line of text:"<<endl;
    cin>>bufferOne;
    //store the string in array bufferOne
    //just the first word in the array string, then the
    //first whitespace encountered
    cout<<"\nThe line of text read by cin>> was:"<<endl;
    cout<<bufferOne<<endl;
    cin.get(bufferTwo, SIZE);
    //the rest of the string
    cout<<"The line of text read by cin.get(bufferTwo,SIZE) was:"<<endl;
    cout<<bufferTwo<<endl;
    system("pause");
}

```

Output:



```
//getline() example
#include <stdlib.h>
#include <iostream.h>

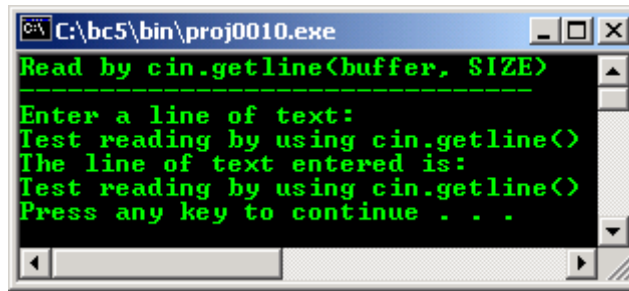
const SIZE = 100;

void main(void)
{
    char buffer[SIZE];

    cout<<"Read by cin.getline(buffer, SIZE)\n"
    <<"-----\n"
    <<"Enter a line of text:"<<endl;
    cin.getline(buffer, SIZE);
    cout<<"The line of text entered is: "<<endl;
    cout<<buffer<<endl;
    system("pause");
}

```

Output:



- `ignore()` member function skips over a designated number of characters (default is one character) or terminates upon encountering a designated delimiter (default is EOF). For example:

```
cin.ignore();           //gets and discards 1 character.

cin.ignore(5);         //gets and discards 5 characters.

cin.ignore(20, '\n');
//gets and discards up to 20 characters or until
//newline character whichever comes first.
```

- `putback()` member function places the previous character obtained by a `get()` on an input stream back onto that stream. For example:

```
char chs;
...
cin.putback(chs);
//put character back in the stream
```

- `peek()` member function returns the next character from an input stream, but does not remove the character from the stream. For example:

```
char chs;
...
chs = cin.peek();
//peek at character
```

- Unformatted I/O performed with `read()` and `write()` member functions. They simply input or output as raw byte.
- The `read()` member function extracts a given number of characters into an array and the `write()` member function inserts `n` characters (nulls included). For example:

```
char texts[100];
...
cin.read(texts, 100);
//read 100 characters from input stream
//and don't append '\0'
```

- Program example:

```
//Using read(), write() and gcount() member functions
#include <stdlib.h>
#include <iostream.h>

const int SIZE = 100;

void main(void)
{
    char buffer[SIZE];

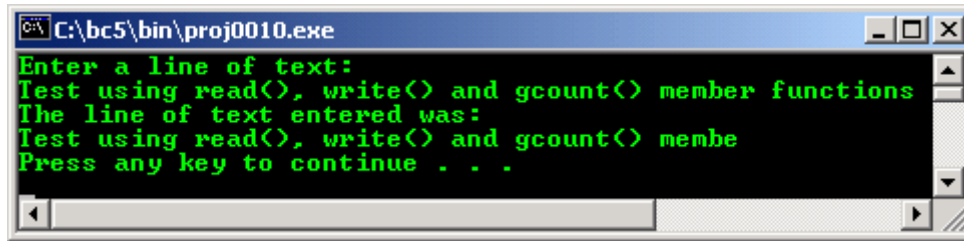
    cout<<"Enter a line of text:"<<endl;
    cin.read(buffer,45);
    cout<<"The line of text entered was: "<<endl;
    cout.write(buffer, cin.gcount());
    //The gcount() member function returns
```

```

        //the number of unformatted characters last extracted
        cout<<endl;
        system("pause");
    }

```

Output :



18.4 Stream Manipulators

- Used to perform formatting, such as:
 - Setting field width.
 - Precision.
 - Unsetting format flags.
 - Flushing stream.
 - Inserting newline in the output stream and flushing the stream.
 - Inserting the null character in the output stream.
 - Skipping whitespace.
 - Setting the fill character in field.

18.4.1 Stream Base

- For stream base we have:

Operator/function	Brief description
hex	To set the base to hexadecimal, base 16.
oct	To set the base to octal, base 8.
dec	To reset the stream to decimal.
setbase()	Changing the base of the stream, taking one integer argument of 10, 8 or 16 for decimal, base 8 or base 16 respectively. setbase() is parameterized stream manipulator by taking argument, we have to include <code>iomanip.h</code> header file.

Table 18.5: Stream base operator and function.

- Program example:

```

//using hex, oct, dec and setbase stream manipulator
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>

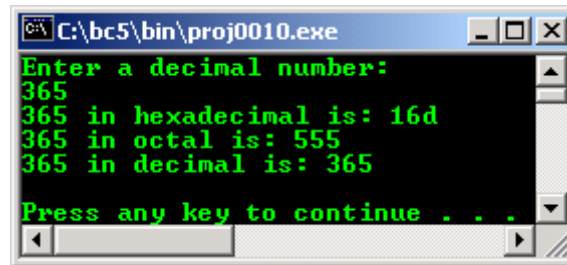
void main(void)
{
    int p;

    cout<<"Enter a decimal number:"<<endl;
    cin>>p;
    cout<<p<< " in hexadecimal is: "
    <<hex<<p<<'\n'
    <<dec<<p<<" in octal is: "
    <<oct<<p<<'\n'
    <<setbase(10) <<p<<" in decimal is: "
    <<p<<endl;
    cout<<endl;
    system("pause");
}

```

```
}
```

Output:



```
C:\bc5\bin\proj0010.exe
Enter a decimal number:
365
365 in hexadecimal is: 16d
365 in octal is: 555
365 in decimal is: 365
Press any key to continue . . .
```

18.4.2 Floating-point Precision

- Used to control the number of digits to the right of the decimal point.
- Use `setprecision()` or `precision()`.
- `precision 0` restores to the default precision of 6 decimal points.

```
//using precision and setprecision
#include <stdlib.h>
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

void main(void)
{
    double theroot = sqrt(11.55);

    cout<<"Square root of 11.55 with various"<<endl;
    cout<<"          precisions"<<endl;
    cout<<"-----"<<endl;
    cout<<"Using 'precision':"<<endl;

    for(int poinplace=0; poinplace<=8; poinplace++)
    {
        cout.precision(poinplace);
        cout<<theroot<<endl;
    }
    cout<<"\nUsing 'setprecision':"<<endl;

    for(int poinplace=0; poinplace<=8; poinplace++)
        cout<<setprecision(poinplace)<<theroot<<endl;
    system("pause");
}
```

Output:


```

C:\bc5\bin\proj0010.exe
Square root of 11.55 with various
precisions
-----
Using 'precision':
3
3
3.4
3.4
3.399
3.3985
3.39853
3.398529
3.3985291
Using 'setprecision':
3
3
3.4
3.4
3.399
3.3985
3.39853
3.398529
3.3985291
Press any key to continue . . .

```

18.4.3 Field Width

- Sets the field width and returns the previous width. If values processed are smaller than the field width, fill characters are inserted as padding. Wider values will not be truncated.
- Use `width()` or `setw()`. For example:

```
cout.width(6); //field is 6 position wide
```

- Program example:

```

//using width member function
#include <iostream.h>
#include <stdlib.h>

void main(void)
{
    int p = 6;
    char string[20];

    cout<<"Using field width with setw() or width()"<<endl;
    cout<<"-----"<<endl;
    cout<<"Enter a line of text:"<<endl;
    cin.width(7);
    while (cin>>string)
    {
        cout.width(p++);
        cout<<string<<endl;
        cin.width(7);
        //use ctrl-z followed by return key or ctrl-d to exit
    }
    system("pause");
}

```

Output:

```

C:\bc5\bin\proj0010.exe
Using field width with setw() or width()
-----
Enter a line of text:
Testing using the setw() or cin.width(?) member function^Z
Testin
    g
    using
    the
    setw()
    or
    cin.wi
    dth(?)
    member
    functi
    on
Press any key to continue . . .

```

18.4.4 Stream Format States

- Format state flag specify the kinds of formatting needed during the stream operations.
- Available member functions used to control the flag setting are: `setf()`, `unsetf()` and `flags()`.
- `flags()` function must specify a value representing the settings of all the flags.
- The one argument, `setf()` function specifies one or more ORed flags and ORs them with the existing flag setting to form a new format state.
- The `setiosflags()` parameterized stream manipulator performs the same functions as the `setf()`.
- The `resetiosflags()` stream manipulator performs the same functions as the `unsetf()` member function. For parameterized stream manipulators you need `iosmanip.h` header file.
- Format state flags are defined as an enumeration in class `ios`. The list for some of the flags is shown below:

Format state flags	Brief description
<code>ios::skipws</code>	Use to skip whitespace on input.
<code>ios::adjustfield</code>	Controlling the padding, left, right or internal.
<code>ios::left</code>	Use left justification.
<code>ios::right</code>	Use right justification.
<code>ios::internal</code>	Left justify the sign, right justify the magnitude.
<code>ios::basefield</code>	Setting the base of the numbers.
<code>ios::dec</code>	Use base 10, decimal.
<code>ios::oct</code>	Use base 8, octal.
<code>ios::hex</code>	Use base 16, hexadecimal.
<code>ios::showbase</code>	Show base indicator on output.
<code>ios::showpoint</code>	Shows trailing decimal point and zeroes.
<code>ios::uppercase</code>	Use uppercase for hexadecimal and scientific notation values.
<code>ios::showpos</code>	Shows the + sign before positive numbers.
<code>ios::floatfield</code>	To set the floating point to scientific notation or fixed format.
<code>ios::scientific</code>	Use scientific notation.
<code>ios::fixed</code>	Use fixed decimal point for floating-point numbers.
<code>ios::unitbuf</code>	Flush all streams after insertion.
<code>ios::stdio</code>	Flush <code>stdout</code> , <code>stderr</code> after insertion.

Table 18.6: State flag format

- `skipws` flags indicates that `>>` should skip whitespace on an input stream. The default behavior of `>>` is to skip whitespace. To change this, use the `unsetf(ios::skipws)`. `ws` stream manipulator also can be used for this purpose.

18.4.5 Trailing Zeroes and Decimal Points

- `ios::showpoint` – this flag is set to force a floating point number to be output with its decimal point and trailing zeroes. For example, floating point `88.0` will print `88` without `showpoint` set and `88.000000` (or many more 0s specified by current precision) with `showpoint` set.

```

//Using showpoint
//controlling the trailing zeroes and floating points
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

void main(void)
{
    cout<<"Before using the ios::showpoint flag\n"
        <<"-----"<<endl;
    cout<<"cout prints 88.88000 as: "<<88.88000
    <<"\ncout prints 88.80000 as: "<<88.80000
    <<"\ncout prints 88.00000 as: "<<88.00000
    <<"\n\nAfter using the ios::showpoint flag\n"
    <<"-----"<<endl;
    cout.setf(ios::showpoint);
    cout<<"cout prints 88.88000 as: "<<88.88000
    <<"\ncout prints 88.80000 as: "<<88.80000
    <<"\ncout prints 88.00000 as: "<<88.00000<<endl;
    system("pause");
}

```

Output :

```

C:\bc5\bin\proj0010.exe
Before using the ios::showpoint flag
-----
cout prints 88.88000 as: 88.88
cout prints 88.80000 as: 88.8
cout prints 88.00000 as: 88

After using the ios::showpoint flag
-----
cout prints 88.88000 as: 88.8800
cout prints 88.80000 as: 88.8000
cout prints 88.00000 as: 88.0000
Press any key to continue . . .

```

18.4.6 Justification

- Use for left, right or internal justification.
- `ios::left` – enables fields to be left-justified with padding characters to the right.
- `ios::right` – enables fields to be right-justified with padding characters to the left.
- The character to be used for padding is specified by the `fill` or `setfill`.
- `internal` – this flag indicates that a number's sign (or base if `ios::showbase` flag is set) should be left-justified within a field, the number's magnitude should be right-justified and the intervening spaces should be padded with the fill character.
- The `left`, `right` and `internal` flags are contained in static data member `ios::adjustfield`, so `ios::adjustfield` argument must be provided as the second argument to `setf` when setting the right, left or internal justification flags because left, right and internal are mutually exclusive.

```

//using setw(), setiosflags(), resetiosflags() manipulators
//and setf and unsetf member functions
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

void main(void)
{
    long p = 123456789L;
    //L - literal data type qualifier for long...
    //F - float, UL unsigned integer...
    cout<<"The default for 10 fields is right justified:\n"
        <<setw(10)<<p
        <<"\n\nUsing member function\n"
        <<"-----\n"

```

```

        <<"\nUsing setf() to set ios::left:\n"<<setw(10);
cout.setf(ios::left,ios::adjustfield);
cout<<p<<"\nUsing unsetf() to restore the default:\n";
cout.unsetf(ios::left);
cout<<setw(10)<<p
        <<"\n\nUsing parameterized stream manipulators\n"
        <<"-----\n"
        <<"\nUse setiosflags() to set the ios::left:\n"
<<setw(10)<<setiosflags(ios::left)<<p
        <<"\nUsing resetiosflags() to restore the default:\n"
        <<setw(10)<<resetiosflags(ios::left)
        <<p<<endl;
    system("pause");
}

```

Output:

- Another program example:

```

//using setw(), setiosflags(), showpos and internal
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

void main(void)
{
    cout<<setiosflags(ios::internal | ios::showpos)
        <<setw(12)<<12345<<endl;
    system("pause");
}

```

Output:

18.4.7 Padding

- fill() – this member function specifies the fill character to be used with adjusted field. If no value is specified, spaces are used for padding. This function returns the prior padding character.
- setfill() – this manipulator also sets the padding character.

```

//using fill() member function and setfill() manipulator

```

```

#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

void main(void)
{
    long p = 30000;

    cout<<p
        <<" printed using the default pad character\n"
        <<"for right and left justified and as hex\n"
        <<"with internal justification.\n"
        <<"-----\n";
    cout.setf(ios::showbase);
    cout<<setw(10)<<p<<endl;
    cout.setf(ios::left,ios::adjustfield);
    cout<<setw(10)<<p<<endl;
    cout.setf(ios::internal,ios::adjustfield);
    cout<<setw(10)<<hex<<p<<"\n\n";

    cout<<"Using various padding character"<<endl;
    cout<<"-----"<<endl;
    cout.setf(ios::right,ios::adjustfield);
    cout.fill('#');
    cout<<setw(10)<<dec<<p<<"\n";
    cout.setf(ios::left,ios::adjustfield);
    cout<<setw(10)<<setfill('$')<<p<<"\n";
    cout.setf(ios::internal,ios::adjustfield);
    cout<<setw(10)<<setfill('*')<<hex<<p<<endl;
    system("pause");
}

```

Output :

```

C:\bc5\bin\proj0010.exe
30000 printed using the default pad character
for right and left justified and as hex
with internal justification.
-----
      30000
30000
0x    7530

Using various padding character
-----
#####30000
30000$$$$$
0x*****7530
Press any key to continue . . .

```

18.4.8 Another Stream Base

- `ios::basefield` – includes the hex, oct and dec bits to specify that integers are to be treated as hexadecimal, octal and decimal values respectively.
- If none of these bits is set, stream insertions default to decimal. Integers starting with 0 are treated as octal values, starting with 0x or 0X are treated as hexadecimal values and all other integers are treated as decimal values. So, set the showbase if you want to force the base of values to be output.
- Program example:

```

//using ios::showbase
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

void main(void)
{

```

```

long p = 2000;

cout<<setiosflags(ios::showbase)
    <<"Printing integers by their base:\n"
    <<"-----\n"
    <<"Decimal    ----> "<<p<<' \n'
    <<"Hexadecimal----> "<<hex<<p<<' \n'
    <<"Octal      ----> "<<oct<<p<<endl;
system("pause");
}

```

Output :

18.4.9 Scientific Notation

- `ios::scientific` and `ios::fixed` flags are contained in the static member `ios::floatfield` (usage similar to `ios::adjustfield` and `ios::basefield`).
- These flags used to control the output format of floating point numbers.
- The `scientific` flag – is set to force the output of a floating point number to display a specific number of digits to the right of the decimal point (specified by the `precision` member function).
- `cout.setf(0, ios::floatfield)` restores the system default format for the floating number output.
- Program example:

```

//Displaying floating number in system
//default, scientific and fixed format

#include <iostream.h>
#include <stdlib.h>

void main(void)
{
    double p = 0.000654321, q = 9.8765e3;

    cout<<"Declared variables\n"
        <<"-----\n"
        <<"0.000654321"<<' \n'<<"9.8765e3"<<' \n\n";

    cout<<"Default format:\n"
        <<"-----\n"
        <<p<<' \t'<<q<<' \n'<<endl;

    cout.setf(ios::scientific,ios::floatfield);
    cout<<"Scientific format:\n"
        <<"-----\n"
        <<p<<' \t'<<q<<' \n';

    cout.unsetf(ios::scientific);
    cout<<"\nDefault format after unsetf:\n"
        <<"-----\n"
        <<p<<' \t'<<q<<endl;
    cout.setf(ios::fixed,ios::floatfield);
    cout<<"\nIn fixed format:\n"
        <<"-----\n"
        <<p<<' \t'<<q<<endl;
}

```

```

    system("pause");
}

```

Output:

```

C:\bc5\bin\proj0010.exe
Declared variables
-----
0.000654321
9.8765e3

Default format:
-----
0.000654321    9876.5

Scientific format:
-----
6.543210e-04   9.876500e+03

Default format after unsetf:
-----
0.000654321    9876.5

In fixed format:
-----
0.000654       9876.500000
Press any key to continue . . .

```

18.4.10 Uppercase and Lowercase

- `ios::uppercase` – this flag is set to force an uppercase X or E to be output with hexadecimal integers or scientific notation floating point values respectively.
- When this flag is set, all letters in a hexadecimal values output uppercase.

```

//Using ios::uppercase flag
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>

void main(void)
{
    long p = 12345678;

    cout<<setiosflags(ios::uppercase)
         <<"Uppercase letters in scientific\n"
         <<"notation-exponents and hexadecimal values:\n"
         <<"-----\n"
         <<5.7654e12<<'\n'
         <<hex<<p<<endl;
    system("pause");
}

```

Output:

```

C:\bc5\bin\proj0010.exe
Uppercase letters in scientific
notation-exponents and hexadecimal values:
-----
5.7654e+12
BC614E
Press any key to continue . . .

```

- Another program example.

```

//Demonstrating the flags() member function
//any format flags() not specified in the
//argument to flags() are reset.

#include <iostream.h>
#include <stdlib.h>

void main(void)
{

long p = 2000;
double q = 0.00124345;

//set a new format state
cout<<"The value of flags variable is: "
  <<cout.flags()<<'\n'
  <<"Print long int and double in original format:\n"
  <<p<<'\t'<<q<<"\n\n";

long OriginalFormat = cout.flags(ios::oct | ios::scientific);
//save the previous format state
cout<<"The value of the flags variable is: "
  <<cout.flags()<<'\n'
  <<"Print long int and double in a new format\n"
  <<"specified using the flags member function:\n"
  <<p<<'\t'<<q<<"\n\n";

cout.flags(OriginalFormat);
//restore the original format setting
cout<<"The value of the flags variable is: "
  <<cout.flags()<<'\n'
  <<"Print values in original format again:\n"
  <<p<<'\t'<<q<<endl;
system("pause");
}

```

Output :

```

C:\bc5\bin\proj0010.exe
The value of flags variable is: 8193
Print long int and double in original format:
2000 0.00124345

The value of the flags variable is: 4040
Print long int and double in a new format
specified using the flags member function:
3720 1.243450e-03

The value of the flags variable is: 8193
Print values in original format again:
2000 0.00124345
Press any key to continue . . .

```

18.4.11 Stream Error States

- eofbit (ios::eofbit) is set automatically for an input stream when end-of-file is encountered. To determine if end-of-file has been encountered on a stream, eof () member function can be used. For example:

```
cin.eof()
```

- Will returns true if end-of-file has been encountered on cin and false otherwise.
- failbit (ios::failbit) is set for a stream when a format error occurs on the stream, but character has not been lost. fail () member function determines if a stream operation has failed, normally recoverable.
- badbit (ios::badbit) – is set for a stream when an error occurs that results in the loss of data. bad () member function determines if a stream operation has failed, normally no recoverable.

- `goodbit(ios::goodbit)` – is set for a stream if none of the bits `eofbit()`, `failbit()` or `badbit()` are set for the stream. `good()` member function returns true if the `bad()`, `fail()` and `eof()` functions would return false.
- `rdstate()` member function returns the error state of the stream. For example

```
cout.rdstate()
```

- Would return the state of the stream which could then be tested.
- `clear()` member function is normally used to restore a streams state to `good()` so that I/O may proceed on the stream.
- Program example:

```
//Using eof(), fail(), bad(), good(), clear()
//and rdstate()

#include <iostream.h>
#include <stdlib.h>

void main(void)
{

int p;

cout<<"Before a bad input operation: \n"
  <<"-----\n"
  <<"  cin.rdstate(): "<<cin.rdstate()
  <<"\n    cin.eof(): "<<cin.eof()
  <<"\n    cin.fail(): "<<cin.fail()
  <<"\n    cin.bad(): "<<cin.bad()
  <<"\n    cin.good(): "<<cin.good()
  <<"\n\nEnter a character (should be integer): "<<endl;
cin>>p;

cout<<"After a bad input operation: \n"
  <<"-----\n"
  <<"  cin.rdstate(): "<<cin.rdstate()
  <<"\n    cin.eof(): "<<cin.eof()
  <<"\n    cin.fail(): "<<cin.fail()
  <<"\n    cin.bad(): "<<cin.bad()
  <<"\n    cin.good(): "<<cin.good()<<"\n\n";
cin.clear();
cout<<"After cin.clear()\n"
  <<"-----\n"
  <<"cin.fail(): "<<cin.fail()<<endl;
system("pause");
}
```

Output:

```

C:\bc5\bin\proj0010.exe
Before a bad input operation:
-----
cin.rdstate(): 0
cin.eof(): 0
cin.fail(): 0
cin.bad(): 0
cin.good(): 1

Enter a character (should be integer):
g
After a bad input operation:
-----
cin.rdstate(): 2
cin.eof(): 0
cin.fail(): 2
cin.bad(): 0
cin.good(): 0

After cin.clear()
-----
cin.fail(): 0
Press any key to continue . . .

```

```

C:\bc5\bin\proj0010.exe
Before a bad input operation:
-----
cin.rdstate(): 0
cin.eof(): 0
cin.fail(): 0
cin.bad(): 0
cin.good(): 1

Enter a character (should be integer):
7
After a bad input operation:
-----
cin.rdstate(): 0
cin.eof(): 0
cin.fail(): 0
cin.bad(): 0
cin.good(): 1

After cin.clear()
-----
cin.fail(): 0
Press any key to continue . . .

```

- Program example compiled using VC++/VC++ .Net.

```

//Displaying floating number in system
//default, scientific and fixed format

#include <iostream>
using namespace std;

void main(void)
{
double p = 0.000654321, q = 9.8765e3;

cout<<"Declared variables\n"
<<"-----\n"
<<"0.000654321"<<' \n'<<"9.8765e3"<<"\n\n";

cout<<"Default format:\n"
<<"-----\n"
<<p<<' \t'<<q<<' \n'<<endl;

cout.setf(ios::scientific,ios::floatfield);
cout<<"Scientific format:\n"
<<"-----\n"
<<p<<' \t'<<q<<' \n';

```

```

cout.unsetf(ios::scientific);
cout<<"\nDefault format after unsetf:\n"
  <<"-----\n"
  <<p<<'t'<<q<<endl;
cout.setf(ios::fixed,ios::floatfield);
cout<<"\nIn fixed format:\n"
  <<"-----\n"
  <<p<<'t'<<q<<endl;
}

```

Output :

```

C:\ "g:\vcnetprojek\searchp...
Declared variables
0.000654321
9.8765e3
Default format:
0.000654321    9876.5
Scientific format:
6.543210e-004  9.876500e+003
Default format after unsetf:
0.000654321    9876.5
In fixed format:
0.000654      9876.500000
Press any key to continue

```

- For C formatted I/O, please refer to [Module 5](#).
- Previous program example compiled using **g++**.

```

/////padding.cpp/////
//using fill() member function and setfill() manipulator
#include <iostream>
#include <iomanip>
using namespace std;

int main(void)
{
    long p = 30000;

    cout<<p <<" printed using the default character pad\n"
      <<"for right and left justified and as hex\n"
      <<"with internal justification.\n"
      <<"-----\n";

    cout.setf(ios::showbase);
    cout<<setw(10)<<p<<endl;
    cout.setf(ios::left, ios::adjustfield);
    cout<<setw(10)<<p<<endl;
    cout.setf(ios::internal, ios::adjustfield);
    cout<<setw(10)<<hex<<p<<"\n\n";

    cout<<"Using character padding"<<endl;
    cout<<"-----"<<endl;
    cout.setf(ios::right, ios::adjustfield);
    cout.fill('#');
    cout<<setw(10)<<dec<<p<<' '\n';
    cout.setf(ios::left, ios::adjustfield);
    cout<<setw(10)<<setfill('$')<<p<<' '\n';
    cout.setf(ios::internal, ios::adjustfield);
    cout<<setw(10)<<setfill('*')<<hex<<p<<endl;
    return 0;
}

```

[bodo@bakawali ~]\$ g++ padding.cpp -o padding

```
[bodo@bakawali ~]$ ./padding
```

```
30000 printed using the default character pad  
for right and left justified and as hex  
with internal justification.
```

```
-----  
          30000  
30000  
0x      7530
```

```
Using character padding
```

```
-----  
#####30000  
30000$$$$$  
0x****7530
```

```
-----o0o-----
```

Further reading and digging:

1. [Check the best selling C/C++, Object Oriented and pattern analysis books at Amazon.com.](#)