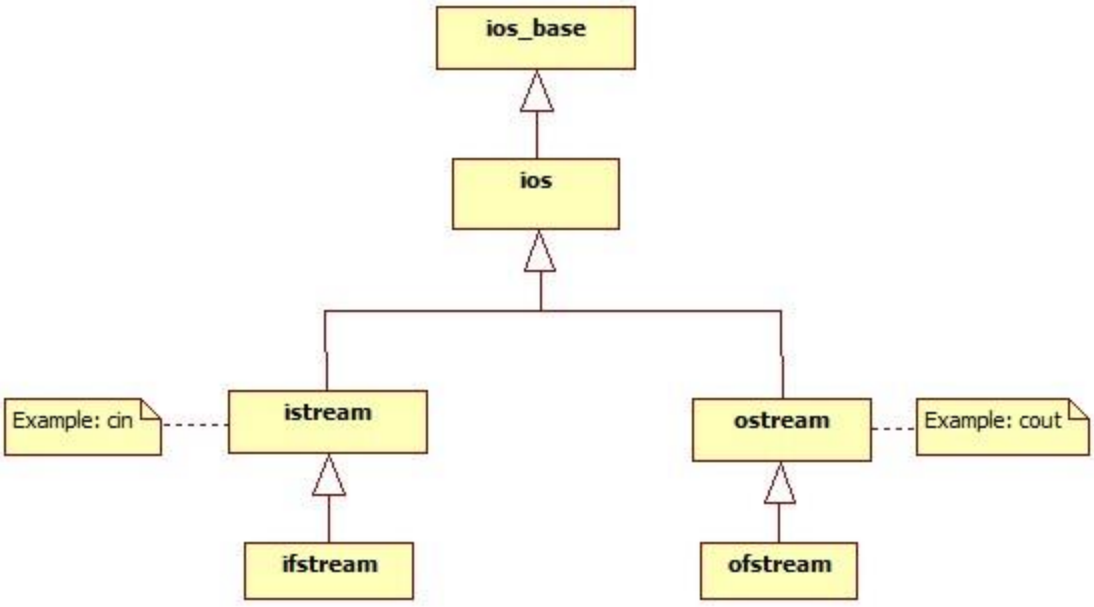
Unit- IV

The Stream Class Hierarchy

- A C++ *stream* is a flow of data into or out of a program, such as the data written to cout or read from cin.
- For this class we are currently interested in four different classes:
  - **istream** is a general purpose input stream. cin is an example of an istream.
  - **ostream** is a general purpose output stream. cout and cerr are both examples of ostreams.
  - **ifstream** is an input file stream. It is a special kind of an istream that reads in data from a data file.
  - **ofstream** is an output file stream. It is a special kind of ostream that writes data out to a data file.
- For example, in the diagram below of (a portion of) the stream class hierarchy, we see that **ifstream** is a specialization of **istream.** What this means is that an ifstream **IS** an



istream, and includes all the properties of the istream class, plus some additional properties of its own.

ios class − This class is the base class for all stream classes. The streams can be input or output streams. This class defines members that are independent of how the templates of the class are defined.

istream Class − The istream class handles the input stream in c++ programming language. These input stream objects are used to read and interpret the input as a sequence of characters. The cin handles the input.

ostream class − The ostream class handles the output stream in c++ programming language. These output stream objects are used to write data as a sequence of characters on the screen. cout and puts handle the out streams in c++ programming language.

The ifstream Class

- An **ifstream** is an **input file stream**, i.e. a stream of data used for reading input from a file.

- Because an ifstream IS an istream, anything you can do to an istream you can also do the same way to an ifstream.
    - In particular, cin is an example of an istream, so anything that you can do with cin you can also do with any ifstream.
- The use of ifstreams (and ofstreams) requires the inclusion of the fstream header:

```
#include <fstream>
```

- Before you can use an ifstream, however, you must create a variable of type ifstream and connect it to a particular input file.
    - This can be done in a single step, such as:

```
ifstream fin( "inputFile.txt" );
```

    - Or you can create the ifstream and open the file in separate steps:

```
ifstream fin;
fin.open( "inputFile.txt" );
```

    - You can even ask the user for the filename, and then open the file they request:

```
string filename;
ifstream fin;
cerr << "Please enter a file name > ";
cin >> filename;
fin.open( filename.c_str( ) ); // The c_str method generates a C-style character string.
```

    - (Note that **string** is also a class, so in the above example, filename is an object of type string. The string class has a method c_str( ), which we "call" through the specific object using the dot operator, just as we call the open( ) method of the ifstream class..)
- Before you use a newly opened file, you should always check to make sure the file opened properly. Every stream object has a fail( ) method that returns "true" if the stream is in a failed state, or "false" otherwise:

```
if( fin.fail( ) ) {
   cerr << "Error - Failed to open " << filename << endl;
   exit( -1 );  // Or use a loop to ask for a different file name.
}
```

- Once you have created an ifstream and connected it to an open file, you read data out of the file the same way that you read from cin:

```
fin >> xMin;
```

- After you are completely done using a stream,you should always close it to prevent possible corruption.
    - This is especially true for output files, i.e. ofstreams.

```
fin.close( );
```

    - After you have closed a stream, you can re-open it connected to a different file if you wish. (I.e. you can reuse the stream variable.)

The ofstream Class

- An **ofstream** is an **output file stream**, and works exactly like ifstreams, except for output instead of input.
- Once an ofstream is created, opened, and checked for no failures, you use it just like cout:

```
ofstream fout( "outputFile.txt" );
fout << "The minimum oxygen percentage is " << minO2 << endl;
```

Reading Data Files

- One of the key issues when reading input data files is knowing how much data to read, and when to stop reading data. There are three commonly used techniques, as shown below. (Which can also be used when reading from the keyboard.)

I. **Specified Number of Records**

- One of the easiest ways is to first read in a number indicating how many data items to read in, and then read in that many data items:

```
int nData;
double x, y, z;

fin >> nData;
for( int i = 0; i < nData; i++ ) {
   fin >> x >> y >> z;
   // Do something with x, y, z
} for loop reading input data
```

- The difficulty with this method is that the number of data items present must be know when the file is created.

II. **Sentinel Value**

- Another commonly used method is to look for a special value (combination) as a trigger to stop reading data:

```
double x, y;

while( true ) {
   fin >> x >> y;
   if( x = = 0.0 && y = = 0.0 )
      break;
   // Do something with x and y
}  // while loop reading input data
```

- The difficulty with this method is that the sentinel value must be carefully chosen so as not to be possible as valid data.

III. **Detect End of File**

- If you know that the data you are reading goes all the way to the end of the file (i.e. there is no other data in the file after the data you are reading), then you can just keep on reading data until you detect that the end of the file has been reached.
- All istreams (and ifstreams) have a method, eof( ), that returns a boolean value of true AFTER an attempt has been made to read past the end of the file, and false otherwise.
- Because the true value isn't set until AFTER you have gone too far, it is important to: (1) read some data first, then (2) check to see if you've gone past the end of the file, and finally (3) use the data only after you have verified that the reading succeeded. Note carefully in the following code that the check for the end of file always occurs AFTER reading the data and BEFORE using the data that was read:

```
double x, y, z;

fin >> x >> y >> z;
while ( !fin.eof( ) ) {
   // Do something with x, y, z
```

```
        // Read in the new data for the next loop iteration.
        fin >> x >> y >> z;
    } // while loop reading until the end of file
```

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

void open(const char *filename, ios::openmode mode);

FilePointer.open("Path",ios::mode);

- Example of file opened for writing: st.open("E:\studytonight.txt",ios::out);
- Example of file opened for reading: st.open("E:\studytonight.txt",ios::in);
- Example of file opened for appending: st.open("E:\studytonight.txt",ios::app);
- Example of file opened for truncating: st.open("E:\studytonight.txt",ios::trunc);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Searches for the file, opens it and positions the pointer at the end of the file. This mode when used with ios::binary, ios::in and ios::out modes, allows you to modify the content of a file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax −

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows −

```
fstream  afile;
```

```
afile.open("file.dat", ios::out | ios::in );
```

Opening a File

```
#include<iostream>
#include <fstream>

using namespace std;

int main()
{
   fstream st;                    // Step 1: Creating object of fstream class
   st.open("abc.txt",ios::out);  // Step 2: Creating new file
   if(!st)                        // Step 3: Checking whether file exist
   {
      cout<<"File creation failed";
   }
   else
   {
      cout<<"New file created";
      st.close(); // Step 4: Closing file
   }
   return 0;
}
```

output:
New file created

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.
Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Ex:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   fstream st;                    // Step 1: Creating object of fstream class
   st.open("abc.txt",ios::out);  // Step 2: Creating new file
   if(!st)                        // Step 3: Checking whether file exist
   {
      cout<<"File creation failed";
```

```
   }
   else
   {
      cout<<"New file created";
      st<<"Hello";                  // Step 4: Writing to file
      st.close(); // Step 5: Closing file
   }
  return 0;
}
```

output:
New file created
$cat abc.txt
Hello

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Ex:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   fstream st;                      // step 1: Creating object of fstream class
   st.open("abc.txt",ios::in);   // Step 2: Creating new file
   if(!st)                          // Step 3: Checking whether file exist
   {
      cout<<"No such file";
   }
   else
   {
      char ch;
      while (!st.eof())
      {
         st.get(ch);                        // Step 4: Reading from file
         cout<<ch;                         // Message Read from file
      }
      st.close(); // Step 5: Closing file
   }
  return 0;
}
```

output:
Hello

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

Ex:

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

int main ()
{
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open("afile.dat");

   cout << "Writing to the file" << endl;
   cout << "Enter your name: ";
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << "Enter your age: ";
   cin >> data;
//   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;
 // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();
   return 0;
```

}
output:
Writing to the file
Enter your name: ram
Enter your age: 19
Reading from the file
ram
19

Binary input and output functions read() and write() in c++

read()- performs file input operation i.e to read the objects stored in afile.

Syntax:
read((char*)&obj, sizeof(obj));
obj is an object of class which will be written to a file

write()-performs file output operation i.e to write the objects to a file,which is stored in a binary form.(only data members of the object are written and not its member functions

Syntax:
write((char*)&obj,sizeof(obj));
obj is an object of class which will be read from a file

Ex:

```
//to illustrate read() and write ()
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
        fstream fs;
        fs.open("test1.txt",ios::out|ios::binary);
        double ar[5]={10,20,30,40,50};
        int i;
        if(!fs)
        {
                cout<<"no file";
                exit(-1);
        }
        else
        {
                fs.write((char *)&ar,sizeof ar);
                fs.close();
        }
        for(i=0;i<=4;i++)
        ar[i]=0;
        fs.open("test1.txt",ios::in|ios::binary);
```

```
        fs.read((char *)&ar,sizeof(ar));
        for(i=0;i<=4;i++)
        cout<<"\n"<<ar[i];
        return 0;
}
```

Output:
10
20
30
40
50

File Position Pointers

- **tellp()** - It tells the current position of the put pointer.
  Syntax: filepointer.tellp()

- tellg() - It tells the current position of the get pointer.
  Syntax: filepointer.tellg()

- seekp() - It moves the put pointer to mentioned location.
  Syntax: filepointer.seekp(no of bytes,reference mode)

- seekg() - It moves get pointer(input) to a specified location.
  Syntax: filepointer.seekg((no of bytes,reference point)

- put() - It writes a single character to file.
- get() - It reads a single character from file.

Note: *For seekp and seekg three reference points are passed:*
ios::beg *- beginning of the file*
ios::cur *- current position in the file*
ios::end *- end of the f*

Ex:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
   fstream st; // Creating object of fstream class
   st.open("data.txt",ios::out);  // Creating new file
   if(!st) // Checking whether file exist
   {
      cout<<"File creation failed";
```

```
    }
    else
    {
        cout<<"New file created"<<endl;
        st<<"Hello Friends"; //Writing to file

        // Checking the file pointer position
        cout<<"File Pointer Position is "<<st.tellp()<<endl;

        st.seekp(-1, ios::cur); // Go one position back from current position

        //Checking the file pointer position
        cout<<"As per tellp File Pointer Position is "<<st.tellp()<<endl;

        st.close(); // closing file
    }
    st.open("data.txt",ios::in);   // Opening file in read mode
    if(!st) //Checking whether file exist
    {
        cout<<"No such file";
    }
    else
    {
        char ch;
     st.seekg(5, ios::beg);  // Go to position 5 from begning.
        cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer
position
        cout<<endl;
        st.seekg(1, ios::cur); //Go to position 1 from beginning.
        cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer
position
        st.close(); //Closing file
    }

    return 0;
}
```

output:

New file created
File Pointer Position is 13
As per tellp File Pointer Position is 12
As per tellg File Pointer Position is 5
As per tellg File Pointer Position is 6

Ex:

//C++ program to demonstrate example of tellg() and tellp() function.

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main()
{
   fstream file;
   //open file sample.txt in and Write mode
   file.open("sample.txt",ios::out);

   if(!file)
   {
      cout<<"Error in creating file!!!";
      return 0;
   }
   //write A to J
   file<<"ABCDEFGHIJ";
   //print the position
   cout<<"Current position is: "<<file.tellp()<<endl;
   file.close();

   //again open file in read mode
   file.open("sample.txt",ios::in);
   if(!file)
   {
      cout<<"Error in opening file!!!";
      return 0;
   }
   cout<<"After opening file position is: "<<file.tellp();

   //read characters untill end of file is not found
   char ch;
   while(file.get(ch))
   {
         cout<<"character is"<<ch<<"\n";
      cout<<"\nposition : "<<file.tellg();   //current position
   }

   //close the file
   file.close();
   return 0;
}
```

output:

Current position is: 10
After opening file position is: 0 character is A
position : 1character isB
position : 2character isC
position : 3character isD
position : 4character isE
position : 5character isF
position : 6character isG

position : 7character isH
position : 8character isI
position : 9character isJ
position : 10

//program to demonstrate seekg(), seekp()

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
   fstream st; // Creating object of fstream class
   st.open("data.txt",ios::out);  // Creating new file
   if(!st) // Checking whether file exist
   {
      cout<<"File creation failed";
   }
   else
   {
      cout<<"New file created"<<endl;
      st<<"Hello Friends"; //Writing to file

      // Checking the file pointer position
      cout<<"File Pointer Position is "<<st.tellp()<<endl;

      st.seekp(-1, ios::cur); // Go one position back from current position

      //Checking the file pointer position
      cout<<"As per tellp File Pointer Position is "<<st.tellp()<<endl;

      st.close(); // closing file
   }
   st.open("data.txt",ios::in);   // Opening file in read mode
   if(!st) //Checking whether file exist
   {
      cout<<"No such file";
   }
   else
   {
      char ch;
    st.seekg(5, ios::beg);  // Go to position 5 from begning.
      cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer
position
      cout<<endl;
      st.seekg(1, ios::cur); //Go to position 1 from current position.
      cout<<"As per tellg File Pointer Position is "<<st.tellg()<<endl; //Checking file pointer
position
```

```
    st.close(); //Closing file
  }

  return 0;
}
```
Output:
New file created
File Pointer Position is 13
As per tellp File Pointer Position is 12
As per tellg File Pointer Position is 5
As per tellg File Pointer Position is 6

Ex: Using put() store the charcters A to Z in a file
```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
        fstream fs;
        fs.open("test.txt",ios::out);
        for(int i=65;i<91;i++)
        {
        fs.put((char)i);
        }
fs.close();
return 0;
}
```
Output:
Open test.txt file
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Ex: Program to illustrate ios::ate and ios::app
```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
  fstream fi;
  fi.open("data.txt",ios::out|ios::ate ) ;
  fi << "## ****************************** line 1\n" ; // writes at the end of the file
    fi.seekp(0) ; // seek to the beginning
    fi << "## line 2\n" ; // writes at the beginning of the file
                // (advances the put position by the number of characters written)
    fi << "## line 3\n" ; // writes at the current put position
   fi.close();
   // if the file exists, do not truncate it
   fi.open("data.txt",ios::app ) ;
    fi << "** line 4\n" ; // appends at the end of the file
    fi.seekp(0) ; // seek to the beginning
```

```
        fi << "** line 5\n" ; // always appends at the end of the file
        fi.seekp(0) ; // seek to the beginning
        fi << "** line 6\n" ; // always appends at the end of the file
fi.close();
   return 0;
}
output:
## line 2
## line 3
********** line 1
** line 4
** line 5
** line 6
```

Ex: Program to illustrate ios::trunc

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
   fstream fi;
   fi.open("data.txt",ios::out) ;
   fi << "------- line 1---------\n" ;
   fi << "###### line 2 ########\n" ;
   fi.close();
      fi.open("data.txt",ios::out|ios::trunc) ;
      fi << "hello mgit\n" ;
      fi.close();
   return 0;
}
output:
hello mgit
```

Formatted I/O Functions

Formatted console input/output functions are used for performing input/output operations at console and the resulting data is formatted and transformed.

Through the **cin** and **cout** objects, we can access the formatted I/O functions.

Some of the most important formatted console input/output functions are –

Functions	Description

width(int width)	Using this function, we can specify the width of a value to be displayed in the output at the console.
fill(char ch)	Using this function, we can fill the unused white spaces in a value(to be printed at the console), with a character of our choice.
setf(arg1, arg2)	Using this function, we can set the flags, which allow us to display a value in a particular format.
peek()	The function returns the next character from input stream, without removing it from the stream.
ignore(int num)	The function skips over a number of characters, when taking an input from the user at console.
putback(char ch)	This function appends a character(which was last read by get() function) back to the input stream.
precision(int num_of_digts)	Using this function, we can specify the number of digits(num_of_digits) to the right of decimal, to be printed in the output.

Ex: Program to illustrate width() and fill()

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

int main()
{
char ch = 'a';

//Adjusting the width of the next value to displayed to 5 columns.
cout.width(5);

cout<<ch <<"\n";
ch = 'b';

//Calling the fill function to fill the white spaces in a value with a character of our choice.
cout.fill('(');
```

```
cout.width(10);
cout<<ch <<"\n";
return 0;
}
```

output:

output:

```
   a
(((((((((b
```

//program to illustrate peek(), putback()

```
#include <iostream>

using namespace std;

int main()
{
   char ch, peekCh;
   cout << "Enter a string: ";
   cin.get(ch);
   cout << "After first cin.get(ch): ";
   cout << "ch = " << ch << endl;

   cin.get(ch);
   cout << "After second cin.get(ch): ";
   cout << "ch = " << ch << endl;

   //put the character back in the stream
   cin.putback(ch);
   cin.get(ch);
   cout << "After putback, the third cin.get(ch): ";
   cout << "ch = " << ch << endl;

   return 0;
}
```

output:

```
Enter a string: abcdefgh
After first cin.get(ch): ch = a
After second cin.get(ch): ch = b
After putback, the third cin.get(ch): ch = b
```

Ex**: program to illustrate ignore()**

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

int main()
{
   int a;
   char str[80];
   cout<<"\nenter rollno:";
   cin >> a;
   cin.ignore();
  cout<<"\nenter stname:";
   cin.getline(str, 80);
   cout <<"\nrollno:"<< a << endl;
   cout <<"\nname="<< str << endl;
   return 0;
```

```
}
output:
enter rollno:1234

enter stname:abcd

rollno:1234

name=abcd
```

Ex:to illustrate precision()

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

int main()
{
   double pi = 3.14159;
/*Calling setf() function to set the flags to display a fixed number of digits after decimal*/
cout.setf(ios::fixed, ios::floatfield);
cout.precision(3);
cout<< pi;
return 0;
}

output:
3.142
```

Error handling functions:

Function	Meaning
int bad()	Returns a non-zero value if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operations.
int eof()	Returns non-zero (true value) if end-of-file is encountered while reading; otherwise returns zero (false value).
int fail()	Returns non-zero (true) when an input or output operation has failed.
int good()	**Returns non-zero (true)** if no error has occurred. This means, all the above functions are false. For example, if fin.good() is true, everything is okay with the stream named as fin and we can proceed to perform I/O operations. **When it returns zero**, no further operations can be carried out.
clear()	Resets the error state so that further operations can be attempted.

Ex: to demonstrate error handling functions

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

```
int main()
{
        ifstream in;
        in.open("abc.txt",ios::in);
        if(!in)
        cout<<"\n file not found";
        else
        cout<<"\nFile="<<in;

        cout<<"\n good()="<<in.good();
        cout<<"\n eof="<<in.eof();
        cout<<"\n fail()="<<in.fail();
        cout<<"\n bad()="<<in.bad();
        in.close();
}
```

Output:

file not found
good()=0
eof=0
fail()=1
bad()=0

Stringstream

A stringstream associates a string object with a stream allowing you to read from the string as if it were a stream (like cin).

Basic methods are –

clear() — to clear the stream
str() — to get and set string object whose content is present in stream.
operator << — add a string to the stringstream object.
operator >> — read something from the stringstream object,

Ex:

```
//to illustrate string stream

#include<iostream>
#include<sstream>
using namespace std;
int main()
{
  stringstream os,os1;
  os << "hello mgit" << endl;
cout <<"\nos="<< os.str();
os1.str("hello cse!"); // set the stringstream buffer to "hello cse"
```

```
cout <<"\nos1="<< os1.str();
os.str("");//clear the buffer
os.clear();//clear() resets any error flags that may have been set and returns the stream back to the ok state
os << "hello" << endl;
cout <<"\nos="<< os.str();
//There are similarly two ways to get data out of a stringstream:
// Use the str() function to retrieve the results of the buffer:
stringstream os2;
os2 << "12345 67.89" << endl;
cout <<"\nos2="<< os2.str();
}
```

Output:

```
os=hello mgit

os1=hello cse!
os=hello

os2=12345 67.89
```

Ex:

```
// CPP program to count words in a string
// using stringstream.
#include <sstream>
#include<iostream>>
using namespace std;

int countwords(string str1)
{
   // breaking input into word using string stream
   stringstream s(str1); // Used for breaking words
   string word; // to store individual words

   int count = 0;
   while (s >> word)
   {
         cout<<"\nword="<<word;
               count++;
            }
   return count;
}

// Driver code
int main()
{
   string s1= "this is cse2 class room";
   cout << " Number of words are: " << countwords(s1);
   return 0;
```

```
}
Output:
word=this
word=is
word=cse2
word=class
word=room Number of words are: 5


// A program to convert string to numbers
#include <iostream>
#include <sstream>
using namespace std;

int main()
{
   string s = "12345";

   // object from the class stringstream
   stringstream st(s);

   // The object has the value 12345 and stream
   // it to the integer x
   int x = 0;
   st >> x;

   // Now the variable x holds the value 12345
   cout << "Value of x : " << x;

   int y=1234;
   y=y+x;
   cout<< "\nvalue of y ="<<y;
return 0;
}

Output:
Value of x : 12345
value of y =13579
```

Operator Overloading:

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Syntax of Operator Overloading

```
class className
{
   ... .. ...
   public
     returnType operator symbol (arguments)
     {
        ... .. ...
     }
   ... .. ...
};
```

- Here, returnType is the return type of the function.
- The returnType of the function is followed by operator keyword.
- Symbol is the operator symbol you want to overload. Like: +, <, -, ++
- You can pass arguments to the operator function in similar way as functions.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(.*)
- ternary operator(?:)

Restrictions on Operator Overloading in C++

Following are some restrictions to be kept in mind while implementing operator overloading.

1. Precedence and Associativity of an operator cannot be changed.
2. Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
3. No new operators can be created, only existing operators can be overloaded.
4. Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Unary Operator(++,- -)

```
#include <iostream>
using namespace std;
class Unary
{
   int a,b;
public:
   void getvalue() {
      cout << "Enter a and b values:";
      cin >> a>>b;
   }
   void operator++() {
      ++a;
   }
   void operator--() {
      --b;
   }
   void display() {
      cout << "a="<<a <<"\t b=" << b << endl;
   }
};
int main() {
   Unary obj;
   obj.getvalue();
   ++obj;    cout << "Increment Number\n";
   obj.display();    --obj;    cout << "Decrement Number\n";
   obj.display();
   }
```

Output:

```
Enter a and b values:
10
20
Increment Number
a=11     b=20
Decrement Number
a=11     b=19
```

//program to overload the comparison operators.(operator overloading)

```
// Example program
#include <iostream>
#include <iostream>
using namespace std;
class Test
{
   public:
   int a;
   void get()
   {
      cout<<"enter a value";
      cin>>a;
```

```
    }
    void operator==(Test t)
    {
        if(a==t.a)
        cout<<"equal";
        else
        cout<<"not equal";
    }
};

int main()
{
   Test t1,t2;
   t1.get();
   t2.get();
   t1==t2;
   return 0;
}
```

Output:

enter a value5
enter a value5
equal

```
// Example binaryoperator +
#include <iostream>
#include <string.h>
using namespace std;
class Test
{
    public:
    char str[100];
    void get()
    {
        cout<<"enter a string";
        cin>>str;
    }
    Test operator+(Test t)
    {
        Test t1;
        strcat(str,t.str);
        strcpy(t1.str,str);
        return t1;

    }
    void disp()
    {
        cout<<"\nconcatenated str is="<<str;
    }
```

```
};

int main()
{
  Test t1,t2,t3;
  t1.get();
  t2.get();
  t3=t1+t2;
  t3.disp();
  return 0;
}
```

output:
enter a stringaaa
enter a stringbbb

concatenated str is=aaabbb

Example:
//Example of Binary Operator Overloading

```
#include<iostream>
using namespace std;
class Rectangle
{

    int L,B;
    public:
    Rectangle()          //Default Constructor
    {
        L = 0;
        B = 0;
    }
    Rectangle(int x,int y)          //Parameterize Constructor
    {
        L = x;
        B = y;
    }
    Rectangle operator+(Rectangle Rec)      //Binary operator overloading func.
    {
        Rectangle R;
            R.L = L + Rec.L;
            R.B = B + Rec.B;

        return R;
    }
void Display()
    {
        cout<<"\n\tLength : "<<L;
```

```
            cout<<"\n\tBreadth : "<<B;
        }
int main()
    {

        Rectangle R1(2,5),R2(3,4),R3;
        //Creating Objects

        cout<<"\n\tRectangle 1 : ";
        R1.Display();

        cout<<"\n\n\tRectangle 2 : ";
        R2.Display();

        R3 = R1 + R2;
        cout<<"\n\n\tRectangle 3 : ";
        R3.Display();
    }
```

output:

```
        Rectangle 1 :
        Length : 2
        Breadth : 5

        Rectangle 2 :
        Length : 3
        Breadth : 4

        Rectangle 3 :
        Length : 5
        Breadth : 9
```

In statement 1, Left object R1 will invoke operator+() function and right object R2 is passing as argument.

Another way of calling binary operator overloading function is to call like a normal member function as follows,

```
R3 = R1.operator+ ( R2 );
```

Example: Subtraction of two complex numbers using operator overloading(binary operator -)

```
#include <iostream>
  using namespace std;
  class Complex
  {
     private:
       float real;
       float imag;
     public:
       Complex(): real(0), imag(0){ }
```

```
        void input()
        {
            cout << "Enter real and imaginary parts respectively: ";
            cin >> real;
            cin >> imag;
        }
        // Operator overloading
        Complex operator - (Complex c2)
        {
            Complex temp;
            temp.real = real - c2.real;
            temp.imag = imag - c2.imag;
            return temp;
        }
        void output()
        {
            if(imag < 0)
                cout << "Output Complex number: "<< real << imag << "i";
            else
                cout << "Output Complex number: " << real << "+" << imag << "i";
        }
    };
  int main()
    {
        Complex c1, c2, result;
        cout<<"Enter first complex number:\n";
        c1.input();
        cout<<"Enter second complex number:\n";
        c2.input();
        // In case of operator overloading of binary operators in C++ programming,
        // the object on right hand side of operator is always assumed as argument by compiler.
        result = c1 - c2;
        result.output();
        return 0;
    }
```

output:

```
Enter first complex number:
Enter real and imaginary parts respectively: 11
12
Enter second complex number:
Enter real and imaginary parts respectively: 14
14
Output Complex number: -3-2i
```

Overloading Binary Operator using a Friend function:

In this approach, the operator overloading function must precede with friend keyword, and declare a function class scope. Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator. All the working and implementation would

same as binary operator function except this function will be implemented outside of the class scope.

Ex:

```
using namespace std;
class B;
class A
{
int n1;
public:
void get()
{
cout<<"enter n1 val:";
cin>>n1;
}
friend void operator > (A,B);
};
class B
{
int n2;
public:
void get()
{
cout<<"enter n2 val:";
cin>>n2;
}
friend void operator > (A,B);
};
void operator>(A a,B b)
{
if(a.n1>b.n2)
cout<<"n1 is greater";
else
cout<<"n2 is greater";
}
int main()
{
A a1; B b1;
a1.get();
b1.get();
a1>b1;
return 0;
}
```

output:

```
enter n1 val:4
enter n2 val:6
n2 is greater
```

Ex:to overload operator >>

```
#include<iostream>
using namespace std;
class Stud
{
int rno;
char name[10];
public:
friend void operator>>(istream &in, Stud &s)
{
cout<<"enter rno:";
in>>s.rno;
cout<<"enter name:";
in>>s.name;
}
friend void operator<<(ostream &os, Stud &s)
{
os<<"rno="<<s.rno;
os<<"\n name="<<s.name;
}
};
int main()
{
Stud s1;
cin>>s1;
cout<<s1;
return 0;
}
```

output:

```
enter rno:1234
enter name:abcd
rno=1234
 name=abcd
```

Ex: complex number addition using friend function

```
#include<iostream>
using namespace std;
  class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
{
real = r;
   imag = i;
}
    void print()
```

```
{
 cout << real << " + i" << imag << endl;
 }
  // The global operator function is made friend of this class so
// that it can access private members
friend Complex operator + (Complex const &, Complex const &);
};

Complex operator + (Complex const &c1, Complex const &c2)
{
    return Complex(c1.real + c2.real, c1.imag + c2.imag);
}
   int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2; // An example call to "operator+"
   c3.print();
   return 0;
}
```

output:

12 + i9