UNIT V

FILES AND ADVANCED FEATURES

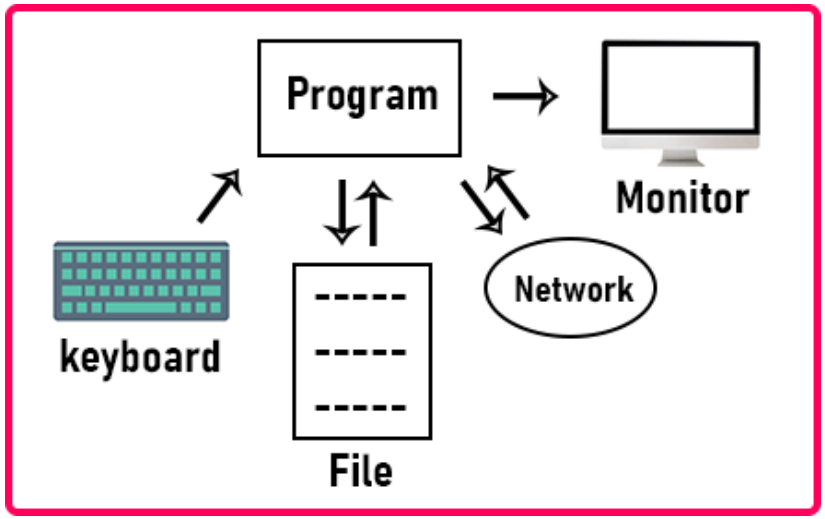
UNIT V - FILES AND ADVANCED FEATURES

- **Topics to be discussed,**
  - C++ Stream classes
  - Formatted IO
  - File classes and File operations
  - Standard Template Library
  - Case Study

UNIT V - FILES AND ADVANCED FEATURES

- **Topics to be discussed,**
  - **C++ Stream classes**
    - Formatted IO
    - File classes and File operations
    - Standard Template Library
    - Case Study

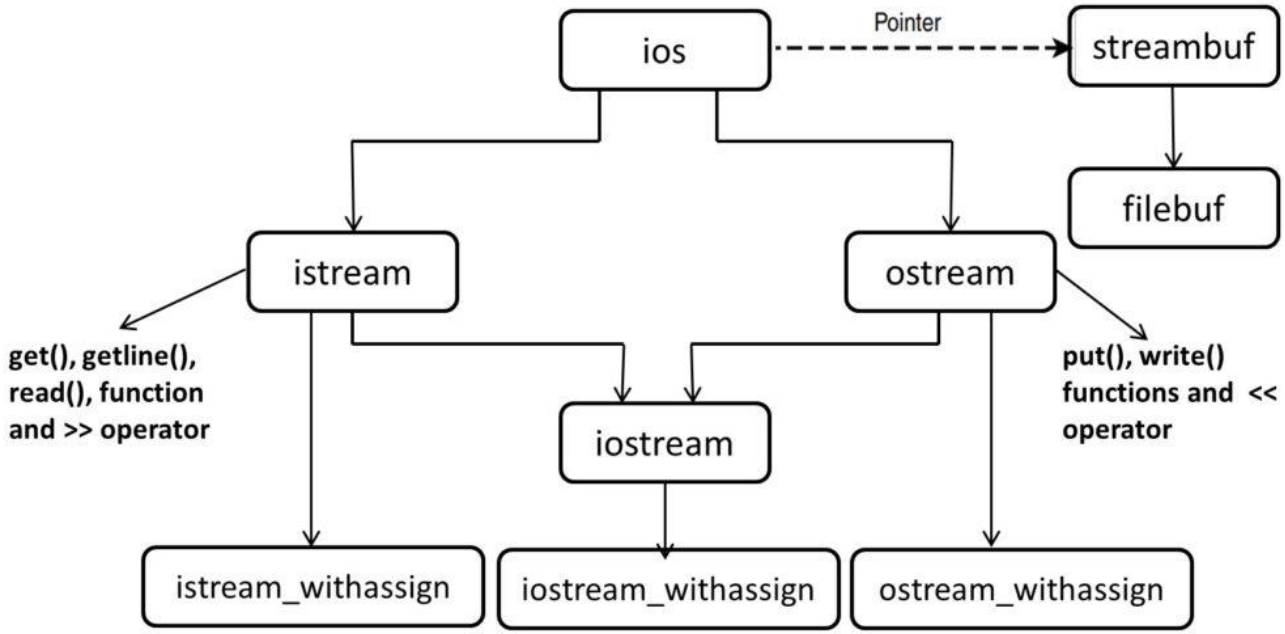
C++ Stream classes



- **What are Streams in C++?**
  - In C++, I/O occurs in **streams, which are sequences of bytes.**
  - In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection, etc.) to main memory.
  - In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, etc.).
  - An application associates meaning with bytes.
  - The bytes could represent characters, raw data, graphics images, digital speech, digital video or any other information an application may require.

C++ Stream classes - Cont'd

- The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files.
- These classes are called stream classes.



I/O Class Hierarchy

- These classes are declared in the header file iostrem.
- The file should be included in all programs that communicate with the console unit.

C++ Stream classes - Cont'd

- As shown in the previous figure, ios is the base class for istream(input stream) and ostream(output stream) which are base classes for iostream(input/output stream).
- The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.
- The class ios provides the basic support for formatted and unformatted input/output operations.
- The class istream provides the facilities for formatted and unformatted input while the class ostream(through inheritance) provides the facilities for formatted output.
- The class iostream provides the facilities for handling both input output streams.
- Three classes namely istream_withassign, ostream_withassign and iostream_withassign add assignment operators to these classes.

C++ Stream classes - Cont'd

Facilities provided stream classes:

- **The ios Class**
  - The **ios** class is the **granddaddy** of all the **stream classes** and contains most of the features you need to operate the C++ stream.
- **The istream and ostream**
  - The **istream** and **ostream** classes are derived from **ios** and are dedicated to **input** and **output**, respectively
  - The **istream** class contains such functions as **get(), getline(), read()**, and the overloaded extraction **(>>)** operators, while **ostream** contains **put()** and **write(),** and the overloaded insertion **(<<)** operators.

C++ Stream classes - Cont'd

- **The iostream class**
  - The iostream class is derived from both istream and ostream by multiple inheritances.
  - Classes derived from it can be used with devices, such as disk files, that may be simultaneously opened for both input and output.
  - Three classes—istream_withassign, ostream_withassign, and iostream_withassign—are inherited from istream, ostream, and iostream, respectively.
  - They add assignment operators to these classes.
- **The streambuf class**
  - Provides an interface to physical devices through buffers and acts as a base class for filebuf.

```
#include<iostream>
using namespace std;
int main()
{
   int count=0;
   char c;
    cout<<"Enter text\n";
   cin.get(c);
    while(c!='\n')
    {
       cout.put(c);
       count++;
       cin.get(c);
   }
    cout<<"\nTotal characters="<<count<<endl;
   return 0;
}
```

Output:

```
Enter text
welcome
welcome
Total characters=7
```

C++ Stream classes - Cont'd

- **istream_withassign class:**
  - This class is variant of **istream** that allows object assignment.
  - The predefined object **cin** is an object of this class and thus may be reassigned at run time to a different **istream** object.

istream_withassign class

```
#include <iostream>
using namespace std;
class Number
{
   public:
      int n1,n2;
      // operator overloading using friend function
      friend void operator>>( istream& mycin, Number& n)
      {
            // cin assigned to another object mycin
            mycin >> n.n1 >> n.n2;
      }
};
int main()
{
    Number nobj;
    cout << "Enter two numbers n1 and n2\n";
     cin>>nobj;
     cout << "n1 = " << nobj.n1 << "\tn2 = " << nobj.n2;
}
```

Output:

```
Enter two numbers n1 and n2
45 78
n1 = 45 n2 = 78
```

C++ Stream classes - Cont'd

- **ostream_withassign class:**
  - This class is variant of **ostream** that allows object assignment.
  - The predefined objects **cout, cerr, clog** are objects of this class and thus may be reassigned at run time to a different **ostream** object.

ostream_withassign class

```
#include <iostream>
using namespace std;
class Number
{
   public:
      int n1,n2;
      Number(int x,int y)
      {
         n1=x;
         n2=y;
      }
      friend void operator<<( ostream& mycout, Number& n)
      {
            // cin assigned to another object mycin
            mycout<< "n1="<<n.n1 <<"\tn2="<< n.n2;
      }
};
int main()
{
   Number nobj(23,89);
   cout<<nobj;
}
```

Output:

```
n1=23    n2=89
```

UNIT V - FILES AND ADVANCED FEATURES

- **Topics to be discussed,**
  - C++ Stream classes
  - **Formatted IO**
  - File classes and File operations
  - Standard Template Library
  - Case Study

Formatted IO

- The C++ programming language provides the several built-in functions to display the output in formatted form.
- These built-in functions are available in the header file **iomanip** and **ios** class of header file **iostream**.
- In C++, there are two ways to perform the formatted IO operations.
  - **Using the member functions of ios class.**
  - **Using the special functions called manipulators defined in iomanip**

Formatted IO – Cont'd

Formatted IO using ios class members

- The **ios** class contains several member functions that are used to perform formatted IO operations.
- The **ios** class also contains few format flags used to format the output.
- It has format flags like **showpos**, **showbase**, **oct**, **hex**, etc.
- The format flags are used by the function **setf( )**.

Formatted IO – Cont'd

Formatted IO using ios class members

Function	Description
width(int)	Used to set the width in number of character spaces for the immediate output data.
fill(char)	Used to fill the blank spaces in output with given character.
precision(int)	Used to set the number of the decimal point to a float value.
setf(format flags)	Used to set various flags for formatting output like showbase, showpos, oct, hex, etc.
unsetf(format flags)	Used to clear the format flag setting.

- All the above functions are called using the built-in object **cout**.

Formatted IO using ios class members – Example1

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Example for formatted IO" << endl;
   cout << "Default: " << endl;
   cout << 123 << endl;
   cout << "width(5): " << endl;
   cout.width(5);
   cout<< 123 << endl;
   cout << "width(5) and fill('*'): " << endl;
   cout.width(5);
   cout.fill('*');
   cout << 123 << endl;
   cout.precision(5);
   cout << "precision(5) ---> " << 123.4567890 << endl;
   cout << "precision(5) ---> " << 9.876543210 << endl;
   cout << "setf(showpos): " << endl;
   cout.setf(ios::showpos);
   cout << 123 << endl;
   cout << "unsetf(showpos): " << endl;
   cout.unsetf(ios::showpos);
   cout << 123 << endl;
   return 0;
}
```

Output:

```
Example for formatted IO
Default:
123
width(5):
  123
width(5) and fill('*'):
**123
precision(5) ---> 123.46
precision(5) ---> 9.8765
setf(showpos):
+123
unsetf(showpos):
123
```

Formatted IO using ios class members – Example2

```
#include <iostream>
using namespace std;
int main()
{
   int number = 15;
  cout.setf( ios::dec );
  cout << "Decimal: " << number << endl;
  cout.unsetf( ios::dec );
  cout.setf( ios::hex );
  cout << "Hexadecimal: " << number << endl;
  cout.unsetf( ios::hex );
  cout.setf( ios::oct );
  cout << "Octal: " << number << endl;
}
```

Output:

```
Decimal: 15
Hexadecimal: f
Octal: 17
```

Formatted IO – Cont'd

Formatted IO using manipulators

- The **iomanip** header file contains several special functions that are used to perform formatted IO operations

Function	Description
setw(int)	Used to set the width in number of characters for the immediate output data.
setfill(char)	Used to fill the blank spaces in output with given character.
setprecision(int)	Used to set the number of digits of precision.
setbase(int)	Used to set the number base.
setiosflags(format flags)	Used to set the format flag.
resetiosflags(format flags)	Used to clear the format flag.

Formatted IO – Cont'd

Formatted IO using manipulators

- The **iomanip** also contains the following format flags using in formatted IO.

Flag	Description
endl	Used to move the cursor position to a newline.
ends	Used to print a blank space (null character).
dec	Used to set the decimal flag.
oct	Used to set the octal flag.
hex	Used to set the hexadecimal flag.
left	Used to set the left alignment flag.
right	Used to set the right alignment flag.
showbase	Used to set the showbase flag.
noshowbase	Used to set the noshowbase flag.
showpos	Used to set the showpos flag.
noshowpos	Used to set the noshowpos flag.
showpoit	Used to set the showpoit flag.
noshowpoint	Used to set the noshowpoint flag.

```cpp
#include <iostream>
#include<iomanip>
using namespace std;
void line()
{
   cout << "----------------------------" << endl;
}
int main()
{
   cout << "Example for formatted IO" << endl;
   line();
   cout << "setw(10): " << endl;
   cout << setw(10) << 99 << endl;
   line();
   cout << "setw(10) and setfill('*') left: " << endl;
   cout << setw(10) << setfill('*') <<left<< 99 << endl;
   line();
   cout << "setw(10) and setfill('*') right: " << endl;
   cout << setw(10) << setfill('*') << right<<99 << endl;
   line();
   cout << "setprecision(5): " << endl;
   cout << setprecision(5) << 123.4567890 << endl;
   line();
   cout << "showpos: " << endl;
   cout << showpos << 999 << endl;
   line();
   cout << "hex: " << endl;
   cout << hex << 100 << endl;
   line();
   cout << "hex and showbase: " << endl;
   cout << showbase << hex << 100 << endl;
   line();
   return 0;
}
```

Output:

```
Example for formatted IO
------------------------------
setw(10):
        99
------------------------------
setw(10) and setfill('*') left:
99********
------------------------------
setw(10) and setfill('*') right:
********99
------------------------------
setprecision(5):
123.46
------------------------------
showpos:
+999
------------------------------
hex:
64
------------------------------
hex and showbase:
0x64
------------------------------
```

UNIT V - FILES AND ADVANCED FEATURES

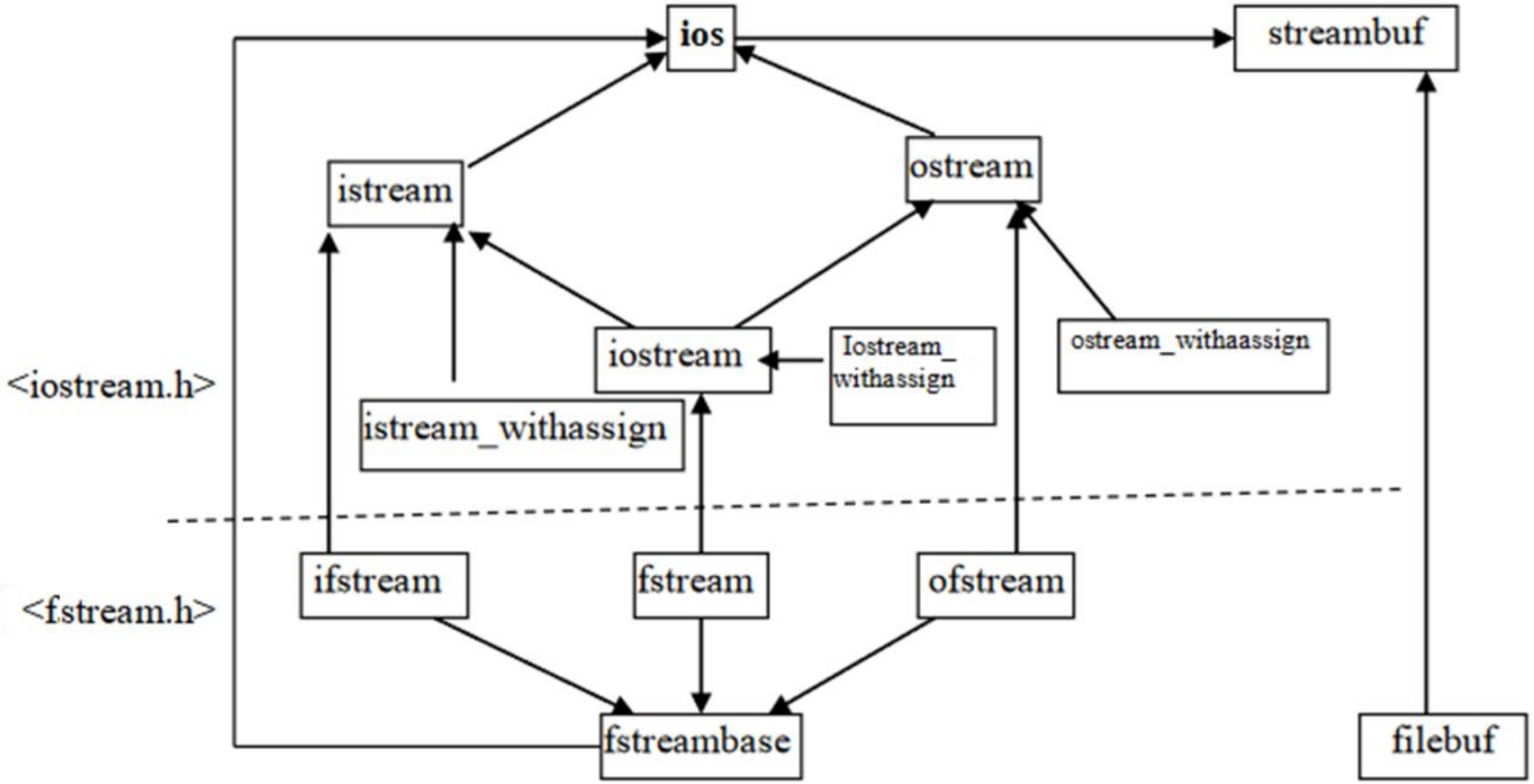
- **Topics to be discussed,**
  - C++ Stream classes
  - Formatted IO
  - **File classes and File operations**
  - Standard Template Library
  - Case Study

File classes and File operations

- Files are used to store data in a storage device permanently.
- File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.
- A stream is an abstraction that represents a device on which operations of input and output are performed.
- The input and output operation between the executing program and the devices like keyboard and monitor are known as "**console I/O operation**".
- The input and output operation between the executing program and files are known as "**disk I/O operation**".

File classes and File operations – Cont'd

The Stream class hierarchy



File classes and File operations – Cont'd

- **The Stream class hierarchy:**
  - **fstreambase:** Provides operations common to file streams. Serves as a base for fstream, ifstream and ofstream and contains open() and close() functions.
  - **ifstream:** Contains input operations in file. Contains open() with default input mode, inherits get(), getline() read(), seekg(), tellg() from istream.
  - **ofstream:** Provides output operation in file. Contains open() with default output mode, inherits put(). Seekp(), tellp() and write() from ostream.
  - **fstream:** Provides support for simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions of istream and ostream through iostream.
  - **filebuf:** The class filebuf sets the file buffer to read and write.

File classes and File operations – Cont'd

- In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream available in fstream headerfile.
- **ofstream:** This Stream class signifies the output file stream and is applied to create files for writing information to files
- **ifstream:** This Stream class signifies the input file stream and is applied for reading information from files
- **fstream:** This Stream class can be used for both read and write from/to files.

File classes and File operations – Cont'd

- For achieving file handling we need to follow the following steps:-

  STEP 1 -Opening a file

  STEP 2 -Writing data into the file

  STEP 3 -Reading data from the file

  STEP 4 -Closing a file.

File classes and File operations – Cont'd

Opening a File

- **Before performing any operation on a file, we must first open it.**
- If we need to write to the file, open it using fstream or ofstream objects.
- If we only need to read from the file, open it using the ifstream object.
- The three objects, that is, fstream, ofstream, and ifstream, have the **open() function** defined in them

File classes and File operations – Cont'd

Opening a File

- **Syntax:**

  open (file_name, mode);

  - The file_name parameter denotes the name of the file to open.
  - The mode parameter is optional.
- **Example:**

  fstream myFile;

  myFile.open(" msg.txt", ios::in);
- A File can be opened in Different modes to perform reading and Writing Operations.
- We can use More than one file mode using bit wise operator " | "
- The default value for fstream mode parameter is in | out. It means that file is opened for reading and writing when we use fstream class.
- When we use ofstream class, default value for mode is out and the default value for ifstream class is in.

File Mode Parameter	Meaning	Explanation
ios::in	Read	Open the file for Reading Purpose Only: if the file don't exist, it will generate an error.
ios::out	Write	Open the file for Writing Purpose Only: if the file already exist, then this mode will open the file and format it. and if there is no file exist, then this mode will create New file.
ios::app	Appending	Open the file for Appending data to end of the file. Using this mode we can Open an already existing file and over write it. but in this mode we can write only at the end of the file.
ios::ate	Appending	It's same like ios::app mode. Open file using this mode will take us to end of the file. but we can write any where in file.
ios::binary	Binary	Open file in Binary mode:
ios::trunc	Truncate/Discard	When we will open the file using this Mode. this mode will delete all data if the file already existing.
ios::nocreate	Don't Create	This Mode can just open an already existing file. if the file not exist, it will show an error.
ios::noreplace	Don't Replace	This mode is used to open new file. if the file is already exist, it will show an error.

File classes and File operations – Cont'd

- Opening file in **ios::out** mode also opens in the **ios::trunc** mode default
- **ios::app** and **ios::ate** takes to the end-of-file when opening but only difference is that ios::app allows to add data only end-of-file but ios::ate allows us to add or modify data at anywhere in the file. In both case file is created if it does not exists.
- Creating a stream **ofstream default implies output(write**) mode and **ifstream implies input(read),** but fstream stream does not provide default parameter so we must provide the mode parameter with fstream.
- The mode can combine two or more parameters using bitwise OR operator (|)
- e.g. fout.open("test.txt",ios::app|ios::out);

File classes and File operations – Cont'd

- **Check the File for Errors (Method 1)**
  - In file handling, it's important to ensure the file was opened by **Checking the File Object** without any error before we can perform any further operations on it.

```
fstream my_file;
my_file.open("my_file", ios::out);
if (!my_file) // check if the file has been opened properly
{
   cout << "File not created!"; // print error message
}
else
{
   cout << "File created successfully!";
   my_file.close();
}
```

if (!my_file) {...}

- This method checks if the file is in an error state by evaluating the file object itself.
- If the file has been opened successfully, the condition evaluates to true.
- If there's an error, it evaluates to false, and you can handle the error accordingly.

File classes and File operations – Cont'd

- **Check the File for Errors (Method 2) -Using the is_open() Function**
- The is_open() function returns
  - **true** - if the file was opened successfully.
  - **false** - if the file failed to open or if it is in a state of error.

```
ofstream my_file("example.txt");
if (!my_file.is_open())
{
    cout << "Error opening the file." << endl;
    return 1;
}
```

File classes and File operations – Cont'd

- **Check the File for Errors (Method 3) - Using the fail() Function**
- The fail() function returns
  - **true** - if the file failed to open or if it is in a state of error.
  - **false** - if the file was opened successfully.

```
ofstream my_file("example.txt");
if (my_file.fail())
{
    cout << "Error opening the file." << endl;
    return 1;
}
```

File classes and File operations – Cont'd

Closing a File

- Once a C++ program terminates, it automatically flushes the streams releases the allocated memory and closes opened files.
- However, as a programmer, we should learn to close open files before the program terminates.
- The fstream, ofstream, and ifstream objects have the close() function for closing files.
- **Syntax:**
  - void close();

File classes and File operations – Cont'd

Writing to a File

- In writing**, we access a file on disk through the output stream and then provide some sequence of characters to be written in the file**.
- **The steps listed below need to be followed in writing a file,**
  - Create a file stream object capable of writing a file, such as an object of ofstream or fstream class.

```
ofstream streamObject;
// Or
fstream streamObject;
```

  - Open a file through constructor while creating a stream object or by calling the open method with a stream object.

```
ofstream streamObject("myFile.txt");
// Or
streamObject.open("myFile.txt");
```

  - Check whether the file has been successfully opened. If yes, then start writing.

```
if(streamObject.is_open())
{
// File Opened successfully
 }
```

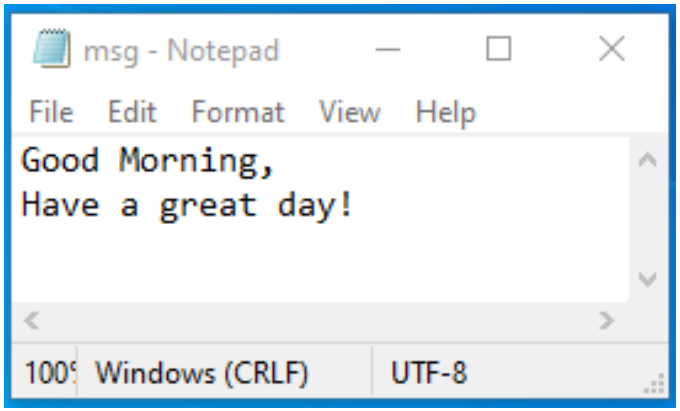
Writing to a File - Example

<< operator is used for writing on the file.

```
#include <fstream>
#include<iostream>
using namespace std;
int main ()
{
   // By default, it will be opened in normal write mode, which is ios::out.
   ofstream myfile("msg.txt"); //equivalent to ofstream myfile; myfile.open("msg.txt")
   if (!myfile) // check if the file has been opened properly
   {
      cout << "File not created!"; // print error message
   }
   else
   {
      cout << "File created successfully!";
      myfile << "Good Morning, \n";
      myfile << "Have a great day!";
      myfile.close();
   }
   return 0;
}
```

Output:

```
File created successfully!
Process returned 0 (0x0)   execution time : 0.078 s
Press any key to continue.
```



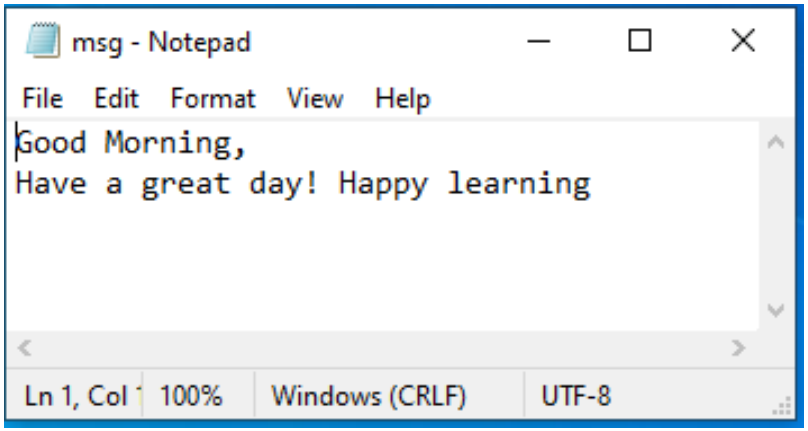
File classes and File operations – Cont'd

Writing in Append Mode

```
#include <fstream>
#include<iostream>
using namespace std;
int main ()
{
   //open file in append mode
   ofstream myfile("msg.txt",ios::app); //equivalent to ofstream myfile; myfile.open("msg.txt",ios::app)
   if (!myfile) // check if the file has been opened properly
   {
      cout << "File not created!"; // print error message
   }
   else
   {
      myfile << " Happy learning\n";
      myfile.close();
   }
   return 0;
}
```

Output:

```
Process returned 0 (0x0)   execution time : 0.079 s
Press any key to continue.
```



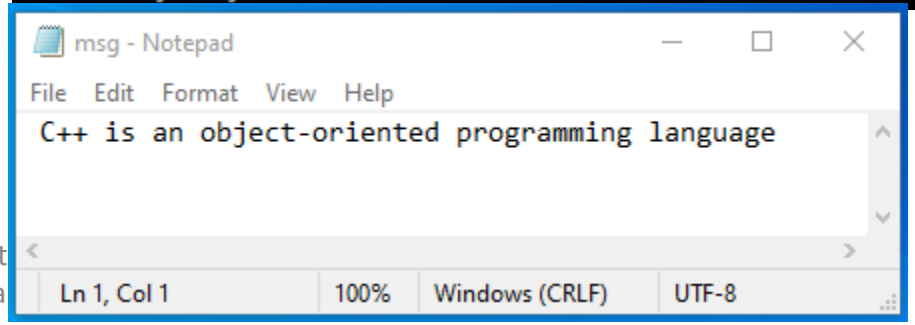
File classes and File operations – Cont'd

Writing in Truncate Mode

```
#include <fstream>
#include<iostream>
using namespace std;
int main ()
{
   //open file in truncate mode
   ofstream myfile("msg.txt",ios::trunc); //equivalent to ofstream myfile; myfile.open("msg.txt",ios::trunc)
   if (!myfile) // check if the file has been opened properly
   {
      cout << "File not created!"; // print error message
   }
   else
   {
      myfile << " C++ is an object-oriented programming language\n";
      myfile.close();
   }
   return 0;
}
```

Output:

```
Process returned 0 (0x0)   execution time : 0.210 s
Press any key to continue.
```



File classes and File operations – Cont'd

Reading from a File

- **We read the data of a file stored on the disk through a stream.**
- **The following steps must be followed before reading a file,**
  - Create a file stream object capable of reading a file, such as an object of ifstream or fstream class.

```
ifstream streamObject;
// Or
fstream streamObject;
```

  - Open a file through constructor while creating a stream object or by calling the open method with a stream object.

```
ifstream streamObject("myFile.txt");
// Or
 streamObject.open("myFile.txt");
```

  - Check whether the file has been successfully opened using is_open(). If yes, then start reading.

```
if(streamObject.is_open())
{
// File Opened successfully.
}
```

File classes and File operations – Cont'd

Error Checking Using the Stream State

- It's quite common that **errors may occur during file operations**.
- There may be different reasons for arising errors while working with files.
- The following are the common problems that lead to errors during file operations.
  - When trying to open a file for reading might not exist.
  - When trying to read from a file beyond its total number of characters.
  - When trying to perform a read operation from a file that has opened in write mode.
  - When trying to perform a write operation on a file that has opened in reading mode.
  - When trying to operate on a file that has not been open.

File classes and File operations – Cont'd

- During the file operations in C++, **the status of the current file stream stores in an integer flag** defined in ios class.
- The following are the file stream flag states with meaning.

Flag Bit	Meaning
badbit	is set when corrupted data is read, i.e. when the type of data in the file does not match the type being read, false otherwise.
failbit	is set when a file fails to open, or when the end of file is read, or when corrupted data is read, , false otherwise
goodbit	is set to true whenever the other three bits are all false, and is false otherwise.
eofbit	is set to true when end-of-file is encountered, false otherwise.

- We use the above flag bits to handle the errors during the file operations.

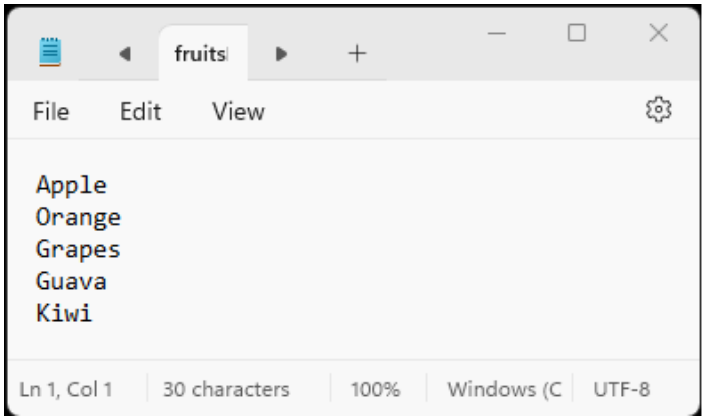
File classes and File operations – Cont'd

- The following methods are used to check and reset the bits:
  - **eof( )** - returns the state of the eof bit.
  - **bad( )** - returns the state of the bad bit.
  - **fail( )** - returns the state of the fail bit.
  - **good( )** - returns the state of the good bit.
  - **clear( )** - Sets the good bit to true and all others to false. This is needed to reset the state if asking the user to enter a new file name after a bad name was entered, or when re-using a stream variable for a new file after encountering the end of a previous file.

File classes and File operations – Cont'd

Read a File in C++ Using the >> Operator

- **We can use the stream input operator >> to read in data from the file.**




```
#include <fstream>
#include<iostream>
using namespace std;
int main ()
{
  ifstream myfile("fruitsList.txt"); //equivalent to ifstream myfile; myfile.open("fruitsList.txt");
  if (myfile.is_open())
  {
    string str;
    while (myfile.good())
    {
      myfile >> str;
      cout << str;
    }
  }
  return 0;
}
```

Output:

```
AppleOrangeGrapesGuavaKiwi
```

- If we notice here, **we are not getting any kind of space or line break after any words.**
- The reason behind this is the >> operator.
- The >> operator simply removes all the whitespace or line breaks from the input file to the output screen.

Now the question is how we can resolve this issue?

The answer to that question lies in the **get() function**.

File classes and File operations – Cont'd

Read a File in C++ Using get()

- We can replace >> with **get(), a member function of our fstream class**
- The **get()** function **reads only a single character at a time**
- The great thing about get() is that it does not ignore white space and instead treats it as a series of ordinary characters.

Using get() - Example

```
#include <fstream>
#include<iostream>
using namespace std;
int main ()
{
  ifstream myfile("fruitsList.txt");
  if (myfile.is_open())
  {
    char ch;
    while (myfile.good())
    {
      ch=myfile.get();
      cout << ch;
    }
  }
  else
  {
    cout << "Error opening in file\n";
  }
  return 0;
}
```

Output:

```
Apple
Orange
Grapes
Guava
Kiwi
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

File classes and File operations – Cont'd

Read a File in C++ Using getline()

- we can use the **getline()** method to read the file line by line.
- **Syntax:**
- istream& getline (istream& is, string& str, char delim);
- Here,
  - "**is**" is the object of the stream class. Along with this, "is" also tells the method about the stream from where it will read input.
  - **str** is simply a string that will store the input after the stream read it from the text file.
  - The **delim** is an optional argument in the getline() function syntax. It tells the stream when to stop reading the input. This argument takes a character as input, and as soon as the stream finds the character, it stops reading.
- Generally, the getline() function takes input until the file ends or encounters a new line (\n).

Using getline() - Example

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
int main ()
{
   ifstream file;
   file.open("fruitsList.txt");
   string str;
   if ( file.is_open() )
   {
      while ( !file.eof() ) //while ( file.good() )
      {
         getline (file, str);
         cout << str<< '\n' ;
      }
   }
   else
   {
      cout << "Couldn't open file\n";
   }
   return 0;
}
```

Output:

```
Apple
Orange
Grapes
Guava
Kiwi
```

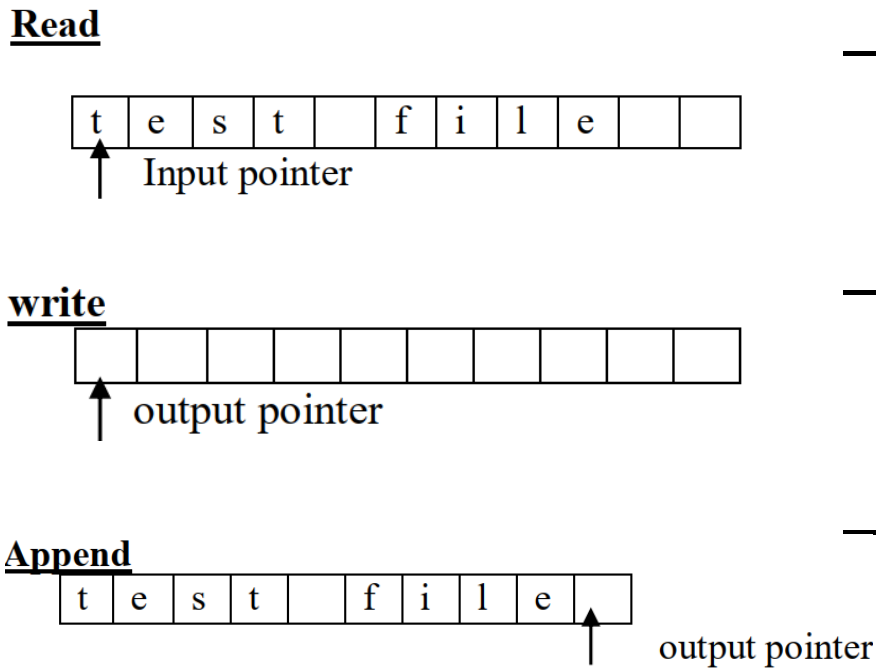
File classes and File operations – Cont'd

File Position Pointers

- The file management system associates two types of pointers with each file.
  - **get pointer (input pointer)**
  - **put pointer (output pointer)**
- These pointers facilitate the movement across the file while reading and writing.
- The **get pointer** specifies a location from where **current read operation** initiated.
- The **put pointer** specifies a location from where current **write operation** initiated.

File classes and File operations – Cont'd

File Position Pointers



- The file pointer is set to a suitable location initially depending upon the mode which it is opened.
  - **Read-only Mode**: When a file is opened in read-only mode, the input (get) pointer is initialized to the beginning of the file.
  - **Write-only: mode**: In this mode, existing contents are deleted if file exists and put pointer is set to beginning of the file.
  - **Append mode**: In this mode, existing contents are unchanged and put pointer is set to the end of file so writing can be done from end of file.

File classes and File operations – Cont'd

- **Functions manipulating file pointers**:
  - C++ I/O system supports 4 functions for setting a file to any desired position inside the file.
  - The functions are

Function	Member of class	Action
seekg()	ifstream	moves get file pointer to a specific
seekp()	ofstream	moves put file pointer to a specific location
tellg()	ifstream	Return the current position of the get ptr
tellp()	ofstream	Return the current position of the put ptr

- These all four functions are available in fstream class by inheritance.

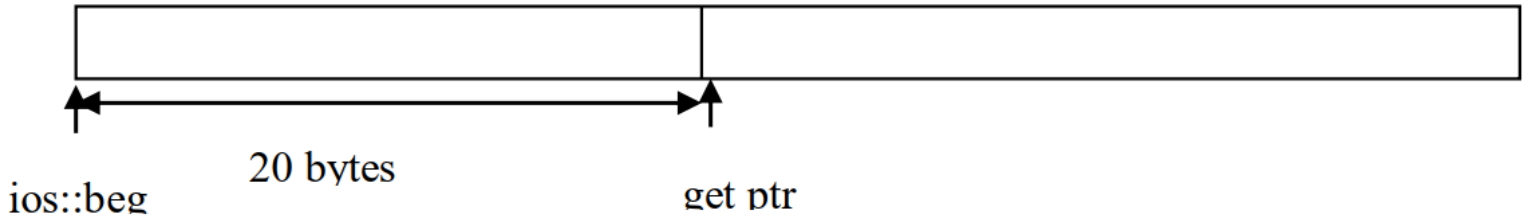
File classes and File operations – Cont'd

- The two seek() functions have following prototypes.
  - istream & seekg (long offset, seek_dir origin =ios::beg);
  - ostream & seekp (long offset, seek_dir origin=ios::beg);
- Both functions set file ptr to a certain offset relative to specified origin.
- The origin is relative point for offset measurement.
- The default value for origin is ios::beg.
- (seek_dir) an enumeration declaration given in ios class as

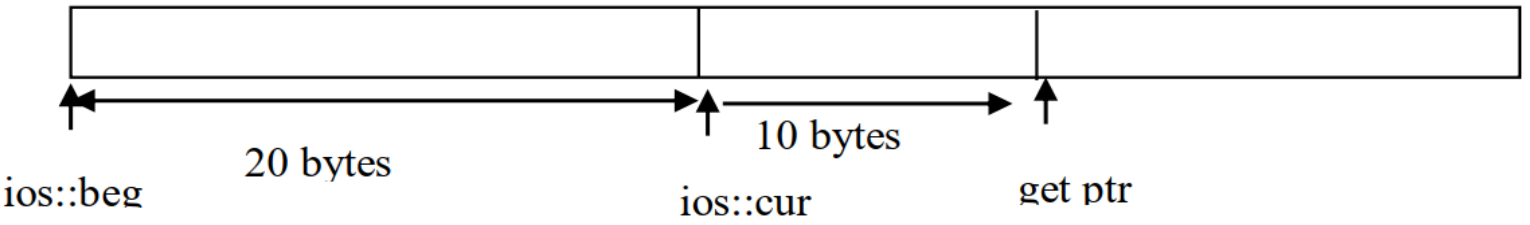
orgin value	seek from
ios::beg	seek from beginning of file
ios::cur	seek from current location
ios::end	seek from end of file

File classes and File operations – Cont'd

- e.g. ifstream infile;
  - infile.seekg(20,ios::beg); or infile.seekg(20); // default ios::beg move file ptr to 20th byte in the file. The reading start from 21st item [byte start from 0] with file.



- Then after, infile.seekg(10,ios::cur); moves get pointer 10 bytes further from current position.



File classes and File operations – Cont'd

- **Similarly**:
- ofstream outfile;
- outfile.seekp(20,ios::beg); // outfile. seekp (20);
  - moves file put pointer to 20thbyte and if write operation is initiated, start writing from 21stitem
- ofsteam outfile("student",ios::app);
- int size=outfile.tellp();
  - Return the size of file in byte to variable size since ios::app takes file put ptr at end of file.
  - The function tellp() returns the current position of put ptr.

File classes and File operations – Cont'd

- **Equivalently**:
- ifstream infile("student");
- infile.seekg(0,ios::end);
- int size=infile.tellg() ;
  - This returns the current file pointer position which is at end of file so we get he size of file "student".

File classes and File operations – Cont'd

- Some of pointer offset calls and their actions:
- **Assume: ofstream fout;**

Seek	Action
fout.seekg(0,ios::beg)	Go to beginning of the file
fout.seekg(0,ios::cur)	Stay at current location
fout.seekg(0,ios::end)	Go to the end of file
fout.seekg(n,ios::beg)	move to (n+1) byte from beginning of file.
fout.seekg (n,ios::cur)	move forward by n bytes from current position
fout.seekg(-n,ios:: cur)	move backward by n bytes from current position
fout.seekp(n,ios:: beg)	move write pointer (n+1) byte location
fout.seekp(-n,ios:: cur)	move write ptr n bytes backwards.

tellp() & tellg() - Example

```
#include <iostream>
#include <fstream>
using namespace std;
int main ()
{
   ofstream file;
   file.open ("myfile.txt", ios::out); // Open file in write mode.
   cout <<"Position of put pointer before writing:" <<file.tellp () << endl;
   file << "Welcome Friends"; // Write on file.
   cout <<"Position of put pointer after writing:" <<file.tellp () << endl;
   file.close ();
   ifstream file1;
   file1.open ("myfile.txt", ios::in); // Open file in read mode.
   cout <<"Position of get pointer before reading:"<< file1.tellg() << endl;
   int iter = 6;
   while(iter--)
   {
      char ch;
      file1 >> ch; // Read from file.
      cout<<ch;
   }
   cout<< endl << "Position of get pointer after reading:"<<file1.tellg();
   file1.close ();
}
```

Output:

```
Position of put pointer before writing:0
Position of put pointer after writing:15
Position of get pointer before reading:0
Welcom
Position of get pointer after reading:6
```

seekg() & seekp() - Example

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   fstream myFile("myfile.txt", ios::out);
   myFile << "123456789";
   myFile.seekp(5);
   myFile<<"*";
   myFile.close();

   myFile.open("myfile.txt", ios::in);
   myFile.seekg(3);
   string myline;
   while (myFile.good())
   {
        std::getline (myFile, myline);
        std::cout << myline << endl;
   }
     myFile.close();
}
```

Output:

```
45*789
```

File classes and File operations – Cont'd

- **Flushing a Stream**
  - In C++, the streams get buffered by default for performance reasons, so we might not get the expected change in the file immediately during a write operation.
  - To force all buffered writes to be pushed into the file, we can either use the flush() function or std::flush manipulator.

File classes and File operations – Cont'd

File I/O with fstrem class

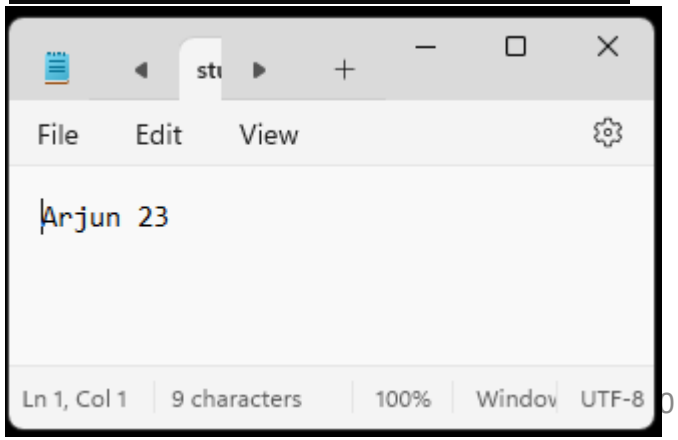
- fstream class supports simultaneous input/output operations.
- It inherits function from istream and ostream class through iostream

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   string name;
   int age;
   fstream file;
   file.open("stuDet.txt",ios::out);
   if(!file)
   {
    cout<<"Error in creating file.."<<endl;
    return 0;
   }
   cout<<"\nFile created successfully."<<endl;
    //read values from user
   cout<<"Enter your name: ";
   cin>>name;
   cout<<"Enter age: ";
   cin>>age;
   //write into file
   file<<name<<" "<<age<<endl;
    file.close();
   cout<<"\nFile saved and closed successfuly."<<endl;
    //re open file in input mode and read data
   //open file
   file.open("stuDet.txt",ios::in);
   if(!file)
   {
      cout<<"Error in opening file..";
      return 0;
   }
   file>>name;
   file>>age;
   cout<<"Name: "<<name<<",Age:"<<age<<endl;
   return 0;
}
```

Output:

```
File created successfully.
Enter your name: Arjun
Enter age: 23

File saved and closed successfuly.
Name: Arjun,Age:23
```



- Write a program to read the name and percentage of ‘n’ different students. Calculate the grade according to following criteria and store the name, percentage and grade in a file. Open the file and display all the details.
- Grade Calculation criteria:

```
if(percentage>=75)
     Grade= A
if(percentage>=45)
    Grade= B
else if(percentage>=35)
   Grade= C
else
  Grade= F
```

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   int count,i;
   string name ;
   char grade;
   float percentage;
   fstream file;
   file.open("stuGrade.txt",ios::out);
   if(!file)
   {
    cout<<"Error in creating file.."<<endl;
    return 0;
   }
   cout<<"\nFile created successfully."<<endl;
   cout<<"\nEnter total number of students:";
   cin>>count;
   for(i=1;i<=count;i++)
   {
      cout<<"Enter your name: ";
      cin>>name;
      cout<<"Enter your percentage: ";
      cin>>percentage;
      if(percentage>=75)
         grade='A';
      else if(percentage>=45)
         grade='B';
      else if(percentage>=35)
         grade='C';
      else
         grade='F';
      //write into file
      file<<name<<" "<<percentage<<" "<<grade<<endl;
   }
   file.close();
   cout<<"\nFile saved and closed successfuly."<<endl;
```

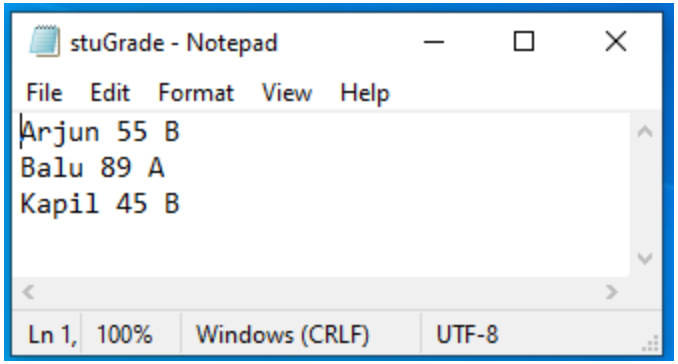
```
 //re open file in input mode and read data
//open file
file.open("stuGrade.txt",ios::in);
if(!file)
{
   cout<<"Error in opening file..";
   return 0;
}
for(i=1;i<=count;i++)
{
   file>>name;
   file>>percentage;
   file>>grade;
   cout<<"Name: "<<name<<",Percentage:"<<percentage<<",Grade:"<<grade<<endl;
}
return 0;
}
```

Output:

```
File created successfully.

Enter total number of students:3
Enter your name: Arjun
Enter your percentage: 55
Enter your name: Balu
Enter your percentage: 89
Enter your name: Kapil
Enter your percentage: 45

File saved and closed successfuly.
Name: Arjun,Percentage:55,Grade:B
Name: Balu,Percentage:89,Grade:A
Name: Kapil,Percentage:45,Grade:B
```



```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
   int count,i;
   string name, grade,per;
   float percentage;
   fstream file;
   file.open("stuGrade.txt",ios::out);
   if(!file)
   {
    cout<<"Error in creating file.."<<endl;
    return 0;
   }
   cout<<"\nFile created successfully."<<endl;
   cout<<"\nEnter total number of students:";
   cin>>count;
   for(i=1;i<=count;i++)
   {
      cout<<"Enter your name: ";
      cin>>name;
      cout<<"Enter your percentage: ";
      cin>>percentage;
      if(percentage>=75)
         grade="first Division/distinction";
      else if(percentage>=45)
         grade=" Second Division";
      else if(percentage>=35)
         grade="Passed";
      else
         grade="Failed";
      //write into file
      file<<name<<"\t"<<percentage<<"\t"<<grade<<endl;
   }
   file.close();
   cout<<"\nFile saved and closed successfuly."<<endl;
    //re open file in input mode and read data
   //open file
   ifstream ifile;
   ifile.open("stuGrade.txt",ios::in);
   if(!file)
   {
      cout<<"Error in opening file..";
      return 0;
   }
```

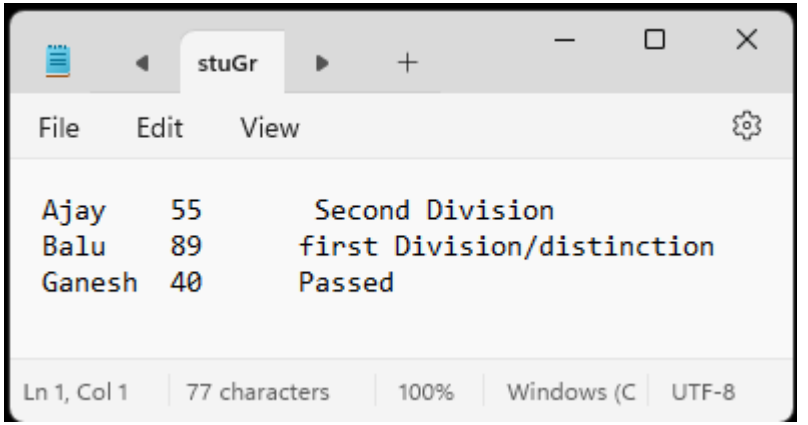
```
  for(i=1;i<=count;i++)
  {
     getline(ifile,name,'\t');
     getline(ifile,per,'\t');
     getline(ifile,grade,'\n');
     cout<<"Name: "<<name<<",Percentage:"<<per<<",Grade:"<<grade<<endl;
  }
  return 0;
}
```

Output:

```
File created successfully.

Enter total number of students:3
Enter your name: Ajay
Enter your percentage: 55
Enter your name: Balu
Enter your percentage: 89
Enter your name: Ganesh
Enter your percentage: 40

File saved and closed successfuly.
Name: Ajay,Percentage:55,Grade: Second Division
Name: Balu,Percentage:89,Grade:first Division/distinction
Name: Ganesh,Percentage:40,Grade:Passed
```



File classes and File operations – Cont'd

File I/O with fstrem class

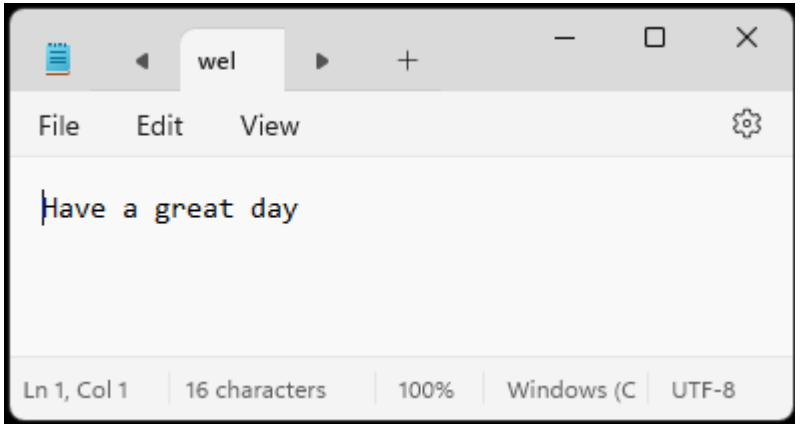
- **The put () and get () function**:
  - The function get() is a member function of the class fstream, and used to read a single character from file.
  - The function put() is member function of the class fstream class and used to write a single character into file.

get () and put () function - Example

```
#include <iostream>
#include <fstream>
#include<cstring>
using namespace std;
int main()
{
   fstream file;
   char ch,str[100];
   cout<<"Enter string:";
   cin.getline(str,100);
   file.open("wel.txt",ios::in|ios::out|ios::trunc);
   if(!file)
   {
    cout<<"Error in creating file.."<<endl;
    return 0;
   }
   else
   {
     for(int i=0;i<strlen(str);i++)
     {
        file.put(str[i]);
     }
     file.seekg(0); // seek to the begining
     cout<<"output string:";
     while(file.good())
     {
        ch=file.get();
        cout<<ch;
     }
     file.close();
   }
}
```

Output:

```
Enter string:Have a great day
output string:Have a great day
```



File classes and File operations – Cont'd

Working with binary files

- Opening binary file requires **open mode ios::binary** to be specified.
- **Binary input/output operations** can be performed using **read() and write() methods**.
- Performing error checks is similar to those in text files.
- **Opening Binary file**
  - Opening binary file is similar to opening text files, but requires specifying ios::binary as additional open mode.

Ex. ifstream myFile ("data.bin", ios::in | ios::binary);

Ex. ofstream myFile;

myFile.open ("data2.bin", ios::out | ios::binary);

Ex. fstream myFile ("data.bin", ios::in | ios::out | ios::binary);

File classes and File operations – Cont'd

Working with binary files

- **Note**: When working with text files, one may omit the second parameter (the i/o mode parameter) to use the default mode.
- However, in order to manipulate binary files, it is necessary to specify the i/o mode also along with ios::binary mode.

File classes and File operations – Cont'd

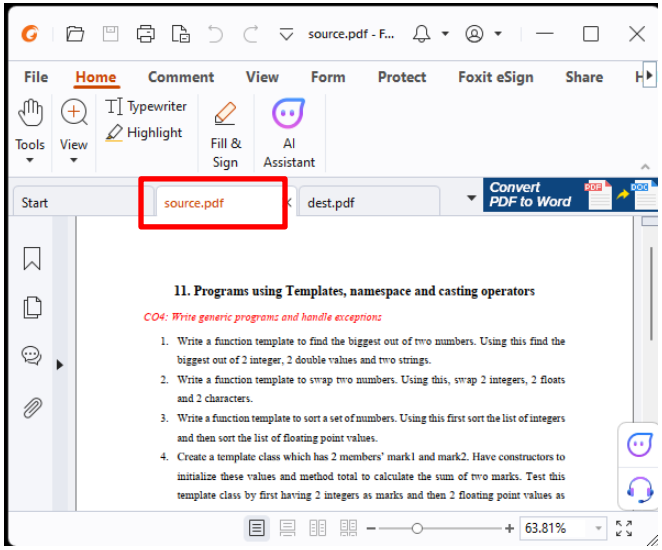
- Manipulating binary file:
  - As binary format does not require type conversion and formatting, conventional text-oriented << and >> operators are not used normally with binary files.
  - **File streams include member functions read() and write() specifically designed to read and write binary data.**
  - One may use methods get() and put() to read/write single character in binary.
  - But, for binary i/o of numbers and other complex data types like struct or class, it requires using methods read() and write().

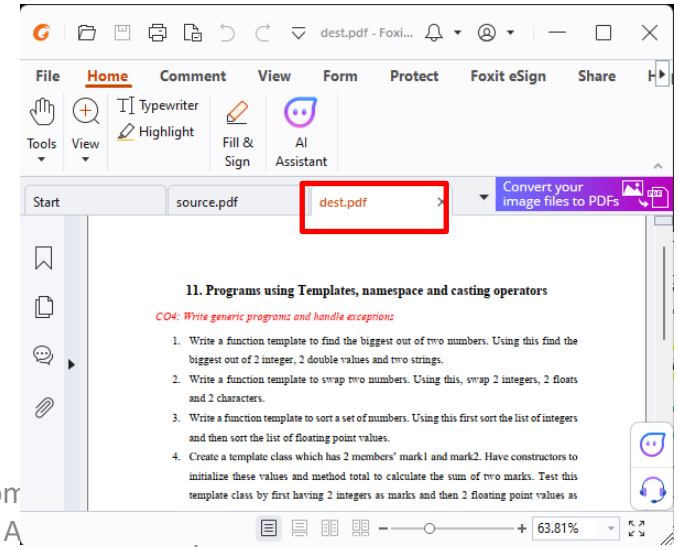
Program to copy contents of one file to another file (binary file)using get() and put() methods

```
#include<fstream>
using namespace std;
int main()
{
   ifstream fin;
   fin.open("source.pdf", ios::in | ios::binary);
   ofstream fout;
   fout.open("dest.pdf", ios::out | ios::binary);
   if (!(fin.is_open())||!(fout.is_open()))
   {
        cout <<"Error opening file...";
        exit(1);
   }
   char ch;
   while(!fin.eof())
   {
        ch = fin.get();
        fout.put(ch);
   }
   cout<<"\nFile copied Successfully....";
   fin.close();
   fout.close();
   return 0;
}
```

Output:

```
File copied Successfully....
Process returned 0 (0x0)   execution time : 0.070 s
Press any key to continue.
```





File classes and File operations – Cont'd

- Function **write() is a member function of ostream** (inherited by ofstream); and **read() is a member function of istream** (inherited by ifstream).
- Objects of class **fstream have both these member functions** available as fstream is derived from ifstrem and ofstream.
- **Syntax:**

  istream& istream::read(char* buf, int size)

  ostream& ostream::write(const char* buf, int size)

  - buf is of type char * (pointer to char) and it specifies the starting address of an array of bytes where the read data elements are to be stored or from where the data elements are to be written.
  - **Note that first parameter should be a C-type array of characters and not a string type of C++.**
  - The size parameter specifies the number of characters to be read or written from/to file.

read () and write () – Normal variables- Example

```
#include<fstream>
#include<iostream>
using namespace std;
int main()
{
  int inum=250;
  float fnum=456.987;
  // open file in write binary mode
  ofstream ofile("number.bin",ios::binary);
  if(ofile.is_open())
  {
  ofile.write((char*)&inum, sizeof(inum) );
  ofile.write((char*)&fnum, sizeof(float) );
  }
  else
  {
    cout<<"\nError in file opening in write mode...";
  }
  ofile.close();
  // open file in read binary mode
  ifstream ifile ("number.bin",ios::binary);
  if(ifile.is_open())
  {
    ifile.read((char*)&inum,sizeof(int));
    ifile.read((char*) &fnum,sizeof(fnum));
    cout<<inum<<" "<<fnum<<endl;
  }
  else
  {
    cout<<"\nError in file opening in read mode...";
  }
  ifile.close();
}
```

Output:

```
250 456.987
```

read () and write () – Single Object - Example

```
#include<iostream>
#include<fstream>
#include<iomanip>
using namespace std;
class student
{
   string name;
   int roll;
   string add;
   public:
   void readdata()
   {
      cout<<"Enter name:";cin>>name;
      cout<<"Enter Roll. no.:";cin>>roll;
      cout<<"Enter address:";cin>>add;
   }
   void showdata()
   {
      cout<<setiosflags(ios::left)<<setw(10)<<roll<<setiosflags(ios::left)<<setw(10)
      <<name<<setiosflags(ios::left)<<setw(10)<<add<<endl;
   }
};
```

```
int main()
{
   student stu;
   ofstream fout("stuData.dat",ios::out|ios::binary);
   cout<<"\nEnter data of a student:\n";
   stu.readdata();
   if(fout.is_open())
   {
      fout.write((char*)&stu,sizeof(stu));
      cout<<"\nObject written to file...\n";
   }
   else
   {
      cout<<"\nError in file opening in write mode...";
   }
   fout.close();

   ifstream fin("stuData.dat",ios::in|ios::binary);
    if(fin.is_open())
   {
      fin.read((char*)&stu,sizeof(stu));
      cout<<"\nObject detail from file:\n";
      stu.showdata();
   }
   else
   {
      cout<<"\nError in file opening in read mode...";
   }
   fin.close();
}
```

Output:

```
Enter data of a student:
Enter name:Balu
Enter Roll. no.:1123
Enter address:Chennai

Object written to file...

Object detail from file:
1123       Balu       Chennai
```

read () and write () – Array of Objects - Example

```
#include<iostream>
#include<fstream>
#include<iomanip>
using namespace std;
class student
{
   string name;
   int roll;
   string add;
   public:
   void readdata()
   {
      cout<<"Enter name:";cin>>name;
      cout<<"Enter Roll. no.:";cin>>roll;
      cout<<"Enter address:";cin>>add;
   }
   void showdata()
   {
      cout<<setw(10)<<roll<<setiosflags(ios::left)<<setw(10)
      <<name<<setiosflags(ios::left)<<setw(10)<<add<<endl;
   }
};
```

```
int main()
{
   student s[50];
   int n;
   cout<<"\nEnter n:";
   cin>>n;
   fstream file;
   file.open("record.dat", ios::in|ios::out|ios::binary);
   cout<<"enter detail for "<<n<<" students:\n";
   for(int i=0;i<n;i++)
   {
      s[i].readdata();
      file.write((char*)&s[i],sizeof(s[i]));
   }
   file.seekg(0);//move pointer begining.
   cout<<"Output from file"<<endl;
   cout<<setiosflags(ios::left)<<setw(10)<<"RollNo"
   <<setiosflags(ios::left)<<setw(10)<<"Name"
   <<setiosflags(ios::left)<<setw(10)<<"Address"<<endl;
   for(int i=0;i<n;i++)
   {
      file.read((char*)&s[i],sizeof(s[i]));
      s[i].showdata();
   }
   file.close();
}
```

Output:

```
Enter n:3
enter detail for 3 students:
Enter name:Aparna
Enter Roll. no.:111
Enter address:Chennai
Enter name:Balu
Enter Roll. no.:222
Enter address:Madurai
Enter name:Sanjay
Enter Roll. no.:333
Enter address:Trichy
Output from file
RollNo    Name      Address
111       Aparna    Chennai
222       Balu      Madurai
333       Sanjay    Trichy
```

Write a program to copy entire content of one file at a time to another file using read() and write()

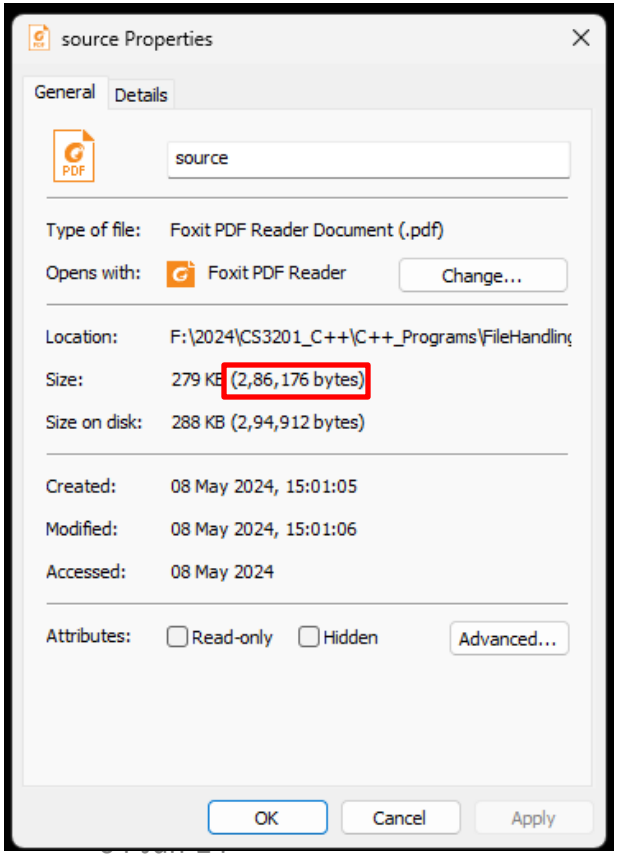
- If a file is small enough to get accommodated in memory, binary copy can be performed by reading and writing entire file at a time using single read and write operation.

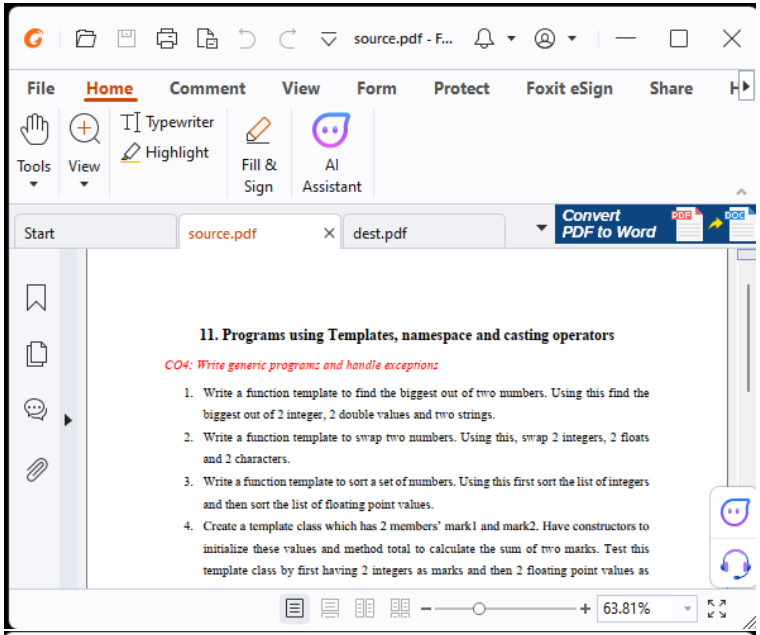
Program to copy entire content of one file at a time to another file using read() and write()

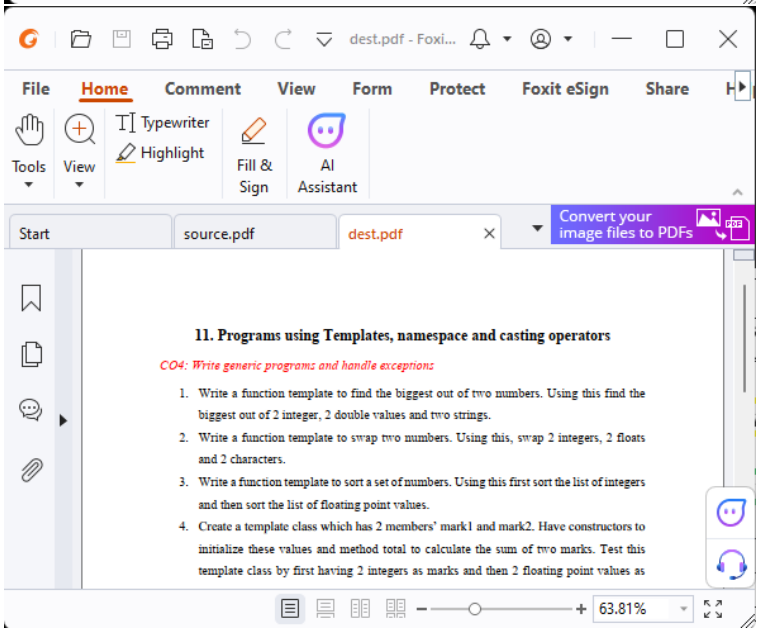
```
#include <iostream>
#include<fstream>
using namespace std;
int main()
{
  ifstream fin;
  fin.open("source.pdf", ios::in | ios::binary|ios::ate); //File pointer at the end of the file
  ofstream fout;
  fout.open("dest.pdf", ios::out | ios::binary);
  if (!(fin.is_open())||!(fout.is_open()))
  {
       cout <<"Error opening file…";
       exit(1);
  }
  streampos size;    //std::streampos - represent positions in a stream.
  char *buf;
  size=fin.tellg();
  cout<<"Size:"<<size;
  buf=new char[size]; //get size of source file and allocate memory of size bytes
  if(buf) // memory allocation successful
  {
    fin.seekg(0,ios::beg); //Pointer at beginning of file
    fin.read(buf,size);
    fout.write(buf,size);
    cout<<"\nFile copied Successfully....";
  }
```

```
  else
  {
     cout<<"Error allocating memory...";
  }
  fin.close();
  fout.close();
  return 0;
}
```

```
Size:286176
File copied Successfully....
```







Write a program to copy entire content of one file at a time to another file using read() and write()

- **If a file is very large, it may not possible to get enough memory to allocate.**
- It is safer to copy block by block.
- Block size should be selected to be large enough to reduce number of read/write operations, but to get successful memory allocation.
- In the previous code, instead of copying the whole file, we can create buffer of predefined size and can copy block by block as shown below

```
//copy block by block
streampos size = 4096;
char buf [4096];
while (!fin.eof())
{
   fin.read (buf, size);
   fout.write (buf, fin.gcount());
}
```

The **std::basic_istream::gcount()** returns the number of characters extracted by the last unformatted input operation.

Other File Management functions

- The copy and delete operations are also associated with file management.
- <cstdio>contains functions for deleting and renaming files.
- **Deleting a file:**

  int remove(const char* pathname);

  – Deletes the file identified by the character string pointed to by pathname.
- **Renaming a file** :

  int rename(const char* old_filename, const char* new_filename);

  – Changes the filename of a file.
  – The file is identified by character string pointed to by old_filename.
  – The new filename is identified by character string pointed to by new_filename.

```
#include<iostream>
#include<fstream>
#include<iomanip>
#include <cstdio>
using namespace std;
class student
{
   string name;
   int roll;
   string add;
   public:
   void readdata()
   {
      cout<<"Enter name:";cin>>name;
      cout<<"Enter Roll. no.:";cin>>roll;
      cout<<"Enter address:";cin>>add;
   }
   void showdata()
   {
      cout<<setiosflags(ios::left)<<setw(10)<<roll<<setiosflags(ios::left)<<setw(10)
      <<name<<setiosflags(ios::left)<<setw(10)<<add<<endl;
   }
   void getAddress()
   {
      cout<<"\nEnter address:";
      cin>>add;
   }
   void create();
   void append();
   void modify();
   void display();
   void delRec();
};
```

```
void student::create()
{
   int n;
   student stu;
   cout<<"\nHow many students record you want to enter?";
   cin>>n;
   ofstream file;
   file.open("record.dat", ios::out|ios::binary);
   if(file.is_open())
   {
      cout<<"enter detail for "<<n<<" students:\n";
      for(int i=0;i<n;i++)
      {
            stu.readdata();
            file.write((char*)&stu,sizeof(stu));
      }
   }
   else
   {
      cout<<"\nError in file opening in write mode...";
   }
   file.close();
}
```

```
1.Create records...
2.Append record
3.Display records...
4.Modify a record
5.Delete a record
6.Exit...
Enter your choice:1

How many students record you want to enter?3
enter detail for 3 students:
Enter name:Ajay
Enter Roll. no.:111
Enter address:Madurai
Enter name:Balu
Enter Roll. no.:222
Enter address:Chennai
Enter name:Devi
Enter Roll. no.:333
Enter address:Trichy
```

```
void student::display()
{
   student stu;
   ifstream file;
   file.open("record.dat", ios::in|ios::binary);
   if(file.is_open())
   {
      cout<<setiosflags(ios::left)<<setw(10)<<"RollNo"
      <<setiosflags(ios::left)<<setw(10)<<"Name"
      <<setiosflags(ios::left)<<setw(10)<<"Address"<<endl;
      while(!file.eof())
      {
         file.read((char*)&stu,sizeof(stu));
         if(file.eof())
            break;
         stu.showdata();
      }
   }
   else
   {
      cout<<"\nError in file opening in read mode...";
   }
   file.close();
   return;
}
```

```
1.Create records...
2.Append record
3.Display records...
4.Modify a record
5.Delete a record
6.Exit...
Enter your choice:3
RollNo    Name      Address
111       Ajay      Madurai
222       Balu      Chennai
333       Devi      Trichy
```

```
void student::append()
{
   student stu;
   ofstream file;
   file.open("record.dat", ios::out|ios::binary|ios::app);
   cout<<"\nEnter data of a student:\n";
   stu.readdata();
   if(file.is_open())
   {
      file.write((char*)&stu,sizeof(stu));
      cout<<"\nObject written to file...\n";
   }
   else
   {
      cout<<"\nError in file opening in write mode...";
   }
   file.close();
}
```

```
1.Create records...
2.Append record
3.Display records...
4.Modify a record
5.Delete a record
6.Exit...
Enter your choice:2

Enter data of a student:
Enter name:Sanjay
Enter Roll. no.:444
Enter address:Chennai

Object written to file...

1.Create records...
2.Append record
3.Display records...
4.Modify a record
5.Delete a record
6.Exit...
Enter your choice:3
RollNo    Name      Address
111       Ajay      Madurai
222       Balu      Chennai
333       Devi      Trichy
444       Sanjay    Chennai
```

```
void student::modify()
{
  student stu;
  bool found=false;
  //file for read/write purpose, so use fstream class
  fstream file("record.dat", ios::in | ios::out | ios::binary);
  if (!file )
  {
    cout << "Error opening input file...";
    exit(3);
  }
  int rno;
  string add;
  cout << "Enter the roll number of the student to modify address: ";
  cin >> rno;
  file.read( (char*)&stu, sizeof(stu));
  while (!file.eof())
  {
    if (stu.roll==rno) // object found
    {
      found=true;
      cout<<"\nCurrent data:";
      stu.showdata();
      stu.getAddress();
      long pos = -1 * sizeof(stu); //for backward move
      file.seekp (pos, ios::cur); //back from current
      file.write ((char*)&stu, sizeof(stu));
      cout<<"\nAddress modified...\n";
      break;
    }
    file.read( (char*)&stu, sizeof(stu)); //read next
  }
  if(!found)
    cout<<"\nGiven student roll number not present.";
  file.close();
}
```

```
1.Create records...
2.Append record
3.Display records...
4.Modify a record
5.Delete a record
6.Exit...
Enter your choice:4
Enter the roll number of the student to modify address: 222

Current data:222        Balu      Chennai

Enter address:Madurai

Address modified...

1.Create records...
2.Append record
3.Display records...
4.Modify a record
5.Delete a record
6.Exit...
Enter your choice:3
RollNo    Name      Address
111       Ajay      Madurai
222       Balu      Madurai
333       Devi      Trichy
444       Sanjay    Chennai
```

```
void student::delRec()
{
   student stu;
   ifstream infile("record.dat", ios::in | ios::binary);
   if (!infile )
   {
         cout << "Error opening input file...";
         exit(2);
   }
   ofstream tmpfile;
   // copy objects in temp file
   tmpfile.open("temp.dat", ios::out | ios::binary);
   if (!tmpfile.is_open())
   {
      cout << "Error opening outfile file...";
      exit(4);
   }
   int rno;
   cout << "Enter the roll number of the student to be deleted: ";
   cin >> rno;
```

```
infile.read( (char*)&stu, sizeof(stu));
while (!infile.eof())
{
   if (stu.roll!=rno) // object not to be deleted
   {
      tmpfile.write ((char*)&stu, sizeof(stu));
   }
   infile.read( (char*)&stu, sizeof(stu)); //read next
}
infile.close();
tmpfile.close();
remove("record.dat"); // remove old student.date file
rename("temp.dat", "record.dat"); // rename temp.dat to record.dat
}
```

```
1.Create records...
2.Append record
3.Display records...
4.Modify a record
5.Delete a record
6.Exit...
Enter your choice:5
Enter the roll number of the student to be deleted: 111

1.Create records...
2.Append record
3.Display records...
4.Modify a record
5.Delete a record
6.Exit...
Enter your choice:3
RollNo    Name      Address
222       Balu      Madurai
333       Devi      Trichy
444       Sanjay    Chennai
```

```
int main()
{
   student stu;
   int ch;
   while(1)
   {
      cout<<"\n1.Create records...";
      cout<<"\n2.Append record";
      cout<<"\n3.Display records...";
      cout<<"\n4.Modify a record";
      cout<<"\n5.Delete a record";
      cout<<"\n6.Exit...";
      cout<<"\nEnter your choice:";
      cin>>ch;
      switch(ch)
      {
            case 1: stu.create(); break;
            case 2: stu.append(); break;
            case 3: stu.display();  break;
            case 4: stu.modify(); break;
            case 5: stu.delRec(); break;
            case 6: exit(0); break;
      }
   }
}
```

```
1.Create records...
2.Append record
3.Display records...
4.Modify a record
5.Delete a record
6.Exit...
Enter your choice:6

Process returned 0 (0x0)   execution time : 4991.609 s
Press any key to continue.
```

UNIT V

FILES AND ADVANCED FEATURES

UNIT V - FILES AND ADVANCED FEATURES

- **Topics to be discussed,**
  - C++ Stream classes
  - Formatted IO
  - File classes and File operations
  - **Standard Template Library**
  - Case Study

Standard Template Library

- STL (standard template library) is a **software library** for the C++ language that **provides a set of well structured generic C++ components** that work together in a seamless way.
- STL is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.
- *Alexander Stepanov* invented it in 1994, and later it was included in the standard library.
- Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.

Standard Template Library – Cont'd

- STL in C++ helps us to code quickly, efficiently, and in a generic way, because most of the data structures and algorithms that we need to use in our code are already implemented in STL library.

Standard Template Library – Cont'd

- **Components of STL in C++**
  - There are four major components of STL in C++:
    - **Containers**
    - **Iterators**
    - **Algorithms**
    - **Function objects or Functors**

Standard Template Library – Cont'd

- **Containers** are just like array data structures that store a collection of objects.
- **iterators** are like pointers objects that allow the traversal of containers
- **Algorithms** are a set of functions that are implemented in an efficient way and these implement different algorithm operations such as search, sort, modify, count, etc. There are many more algorithms and these are generally defined in algorithm.h. we need to include <algorithm>.
- **functors or function objects** are classes that override and overload the parenthesis operator such that they can be used as functions that maintain state and can be parameterized and these are generally defined in <functional.h>.

Standard Template Library – Cont'd

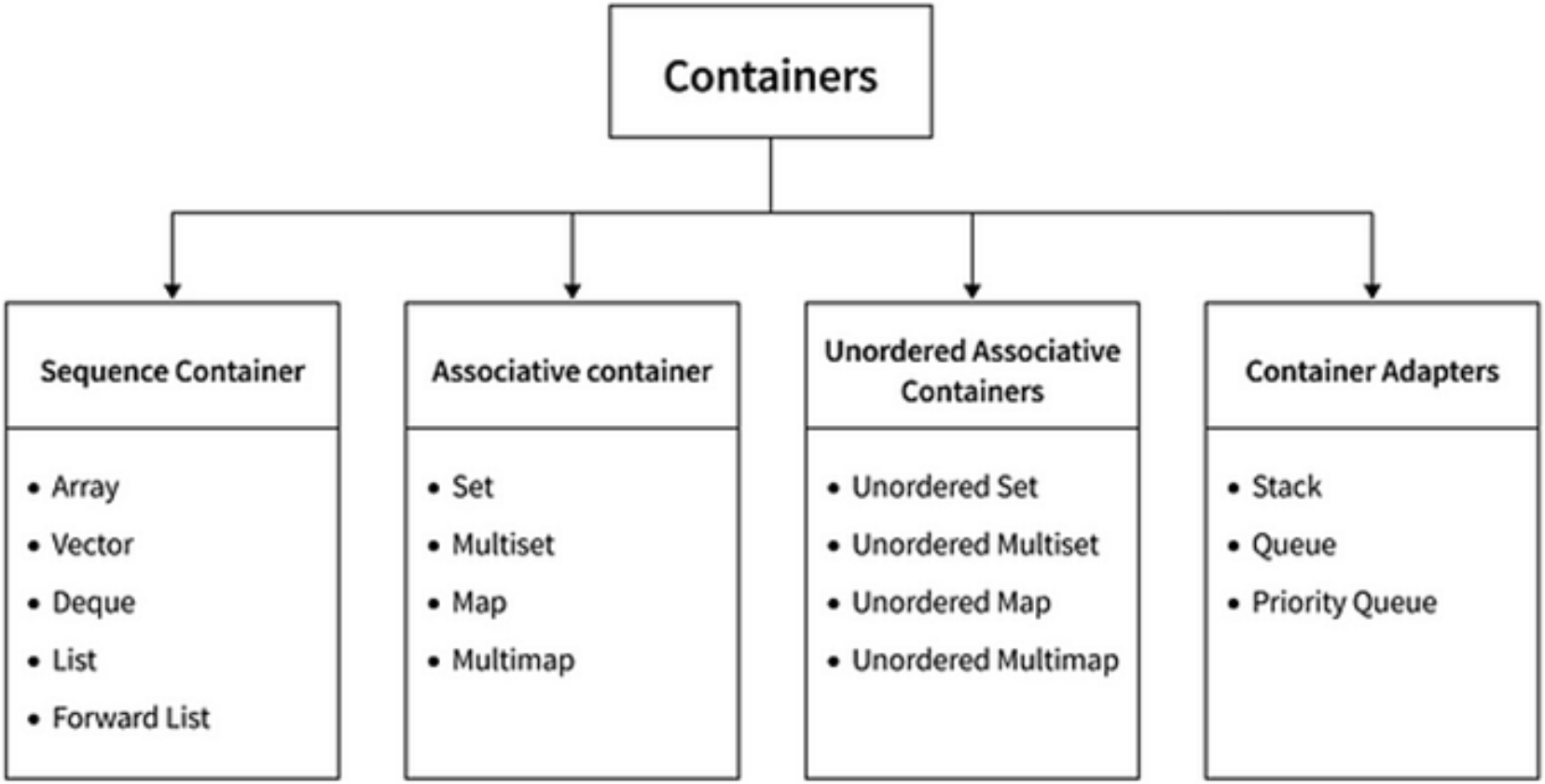
Containers

- Containers are **objects that store a collection of elements.**
- They are implemented as class templates, providing flexibility in the types of elements they can hold.
- Containers manage the storage space for their elements and offer member functions to access them, either directly or through iterators.
- For example, if we need to store a list of names, we can use a vector.

Standard Template Library – Cont'd

Containers

- **Types of STL Container in C++**



Standard Template Library – Cont'd

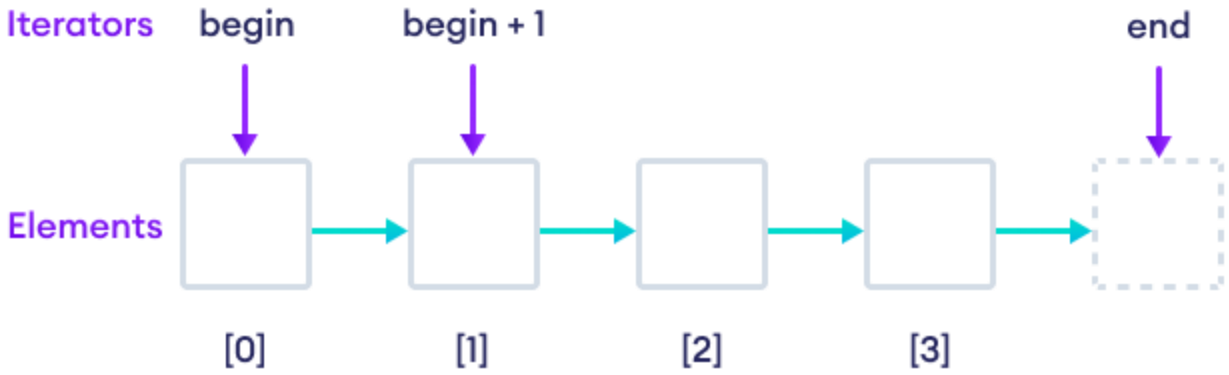
Iterators

- An **iterator** is an object (like a pointer) that points to an element inside the container.
- We can **use iterators to move through the contents of the container**.
- They can be visualized as something similar to a pointer pointing to some location and we can access the content at that particular location using them.
- Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers.

Standard Template Library – Cont'd

Iterators

- Suppose we have a vector named nums of size **4**.



- **nums.begin()** points to the first element in the vector i.e **0th** index
- **nums.begin() + i** points to the element at the **ith** index.
- **nums.end()** points to one element past the final element in the vector

Standard Template Library – Cont'd

Iterators

- nums.begin() points to the first element in the vector i.e **0th** index
- nums.begin() + i points to the element at the **ith** index.
- nums.end() points to one element past the final element in the vector

Standard Template Library – Cont'd

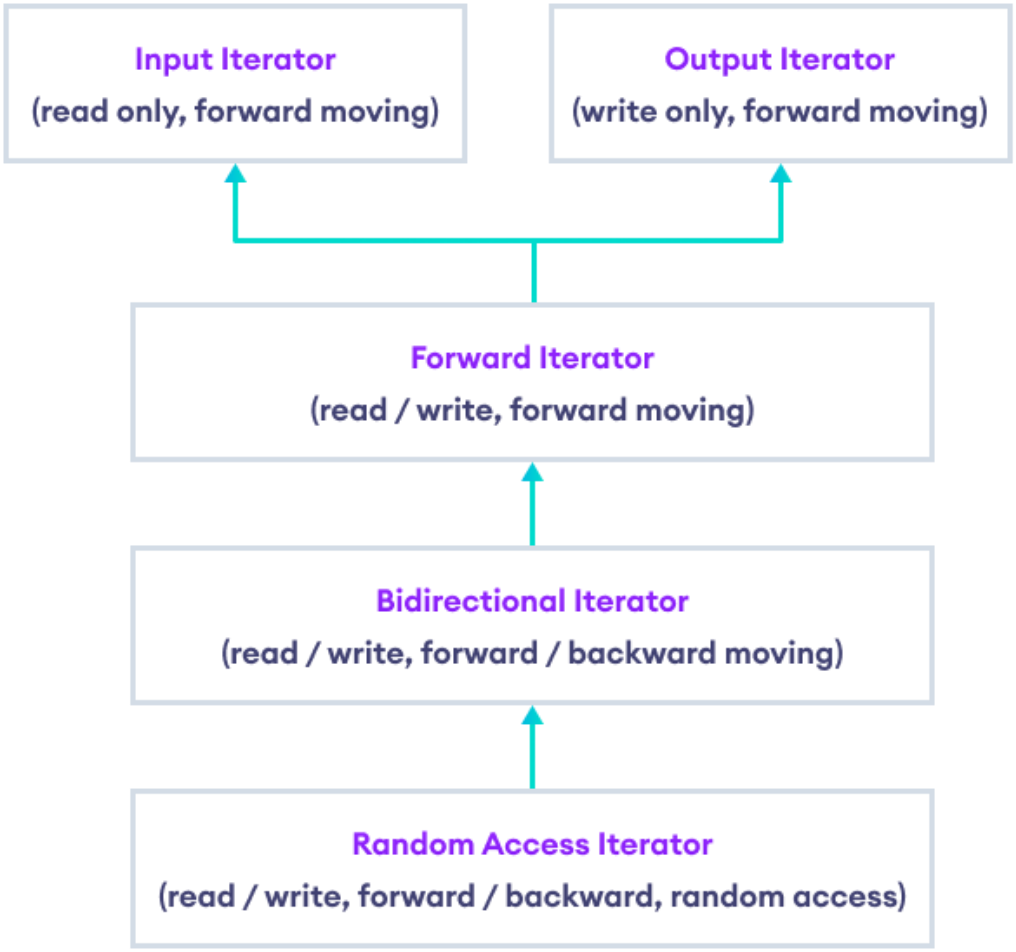
Iterators

- **Iterator Fundamental Operations**
  - Some fundamental operations that can be performed on iterators are shown in the table below.

Operations	Description
*itr	returns the element at the current position
itr->m	returns the member value m of the object pointed by the iterator and is equivalent to (*itr).m
++itr	moves iterator to the next position
-–itr	moves iterator to the previous position
itr + i	moves iterator by i positions
itr1 == itr2	returns true if the positions pointed by the iterators are the same
itr1 != itr2	returns true if the positions pointed by the iterators are not the same
itr = itr1	assigns the position pointed by itr1 to the itr iterator

Standard Template Library – Cont'd

Iterators



- **Iterator Types**
  - The C++ standard template library provides five types of iterators.
  - They are:
    - Input Iterator
    - Output Iterator
    - Forward Iterator
    - Bidirectional Iterator
    - Random-access-iterator

Standard Template Library – Cont'd

Algorithms

- The C++ algorithms library is a **vast collection of functions**.
- It provides us with **various inbuilt functions on searching, sorting, counting, manipulating, etc**.
- The algorithms library operates on a range of elements and the range are defined as [first, last) where the last element is not included in the range.
- The algorithms library is present in #include<algorithm> header file.
- Algorithms operate directly on values through iterators and do not change the structure of the container.
- Thus, we can use the algorithms on any type of container.
- Also, this library saves our time and effort and provides us with the most reliable implementation of the algorithm.

Standard Template Library – Cont'd

Algorithms

- **Types of Algorithms in Algorithm Library**
  - Modifying algorithms
  - Non-modifying algorithms
  - Sorting algorithms
  - Searching algorithms
  - Maximum and minimum operations

Standard Template Library – Cont'd

Algorithms

- **Modifying algorithms**
- Some examples of modifying algorithms are:
  - copy(first1, last1, first2): It copies the elements from the range first1 to last1 excluding last1 into the range starting from first2.
  - fill(first, last, value): It assigns the given value to all the elements in the range.
  - swap(container1, container2): It swaps the elements of one container with another.
  - reverse(first, last): It reverses the order of elements in the given range.

Standard Template Library – Cont'd

Algorithms

- **Non Modifying algorithms**
- Some examples of non-modifying algorithms are:
  - count(first, last, value): It returns the number of occurrences of the given value in the given range.
  - equal(first1, last1, first2): It compares the value in the given range and returns true if the values are equal otherwise false.
  - search(first1, last1, first2, last2): It searches the sequence [first1, last1) for the first occurrence of the subsequence defined by [first2, last2), and returns an iterator to its first element of the occurrence, or last1 if no occurrences are found.

Standard Template Library – Cont'd

Algorithms

- **Sorting algorithms**
- Some examples of sorting algorithms are:
  - sort(start_iterator, end_iterator): It sorts the elements of the container in the given range.
  - partial_sort(start, middle, end): It sorts the elements partially in the range (start, end) such that elements from start to middle are sorted in ascending order and are smallest in the container,
  - is_sorted(start_iterator, end_iterator): It returns true if the elements in the given range are sorted.

Standard Template Library – Cont'd

Algorithms

- **Searching algorithms**
- Some examples of searching algorithms are:
  - binary_search(first, last, value): It searches for the value in the given range and returns true if it is found.
  - equal_range(first, last, value): It returns a subrange of elements where the elements are equal to the given value.
  - upper_bound(start_iterator, end_iterator, value): It returns the upper bound of the value in the given sorted range.
  - lower_bound(start_iterator, end_iterator, value): It returns the lower bound of the value in the given sorted range.

Standard Template Library – Cont'd

Algorithms

- **Maximum and Minimum operations**
- Some functions performing minimum and maximum operations are:
  - min(value1, value2): It returns the minimum of the two values.
  - max(value1, value2): It returns the maximum of the two values.
  - min_element(start_iterator, end_iterator): It returns the minimum element in the given range.
  - max_element(start_iterator, end_iterator): It returns the maximum element in the given range.

Standard Template Library – Cont'd

Containers

- **Sequence Containers in C++**
  - Sequence Containers are used to **store elements in a particular order.**
  - They **provide efficient random access** to elements and allow the insertion and deletion of elements at any position.
  - Internally, sequential containers are implemented as **arrays** or **linked lists** data structures**.**
  - Sequence Containers can store various data types, such as integers, characters, strings, and user-defined data types.
  - They are implemented as dynamic arrays, linked lists, and arrays of fixed-size elements.
  - These containers provide several functions to manipulate the elements, such as accessing elements, inserting and deleting elements, and searching for elements.

Standard Template Library – Cont'd

Containers - Sequence

- The **five sequence containers** supported by STL are:
- **Array**
  - Arrays are **sequential homogeneous containers of fixed size**. The elements are stored in contiguous memory locations.
  - **Syntax:**

    array<object_type, size> array_name;
- **Vector**
  - Vectors are **dynamic arrays**, allowing the insertion and deletion of data from the end.
  - They can grow or shrink as required. Hence their size is not fixed, unlike arrays.
  - **Syntax:**

    vector<object_type> vector_name;

Standard Template Library – Cont'd

Containers - Sequence

- **Deque**
  - Deque is **double-ended queue** that allows inserting and deleting from both ends. They are more efficient than vectors in case of insertion and deletion. Its size is also dynamic.
  - **Syntax:**

    deque<object_type> deque_name;
- **List**
  - The list is a sequence container that allows insertions and deletions from anywhere. It is a **doubly linked list**. They allow non-contiguous memory allocation for the elements.
  - **Syntax:**

    list<object_type> list_name;
- **Forward List**
  - Forward Lists are introduced from C++ 11. They are implemented **as singly linked list** in STL in C++. It uses less memory than lists and allows iteration in only a single direction.
  - **Syntax:**

    forward_list<object_type> forward_list_name;

Standard Template Library – Cont'd
Containers - Sequence

- **Array :**
  - In C++, std::array is a container class that encapsulates **fixed size arrays**.
  - It is **similar to the C-style arrays** as it stores multiple values of similar type.
  - std::array is defined in the <array> header so we must include this header before we can use std::array.

```
#include <array>
```

Standard Template Library – Cont'd

Containers - Sequence

- **Array Declaration**
  - **Syntax:**
  - array<data_type,size> name;
    - data_type: The data types of the elements to be stored in the array
    - size: The size of the array
  - **Example:**
  - array < int, 5 > arr;
    - initializes a array of size 5 which stores integer values
- **Array Initialization**
- **Syntax:**
  - array<data_type,size> name{initial_values};
    - initial_values: optional parameter which initializes the array with the given values.
  - **Example:**
  - array < int, 5 > arr { 1, 2, 3, 4, 5 };
    - initializes a array of size 5 having intial values 1,2,3,4,5

Standard Template Library – Cont'd

Containers - Sequence

- **Functions on arrays**
  - **[i]**: It is used to access the element stored at index i.
  - **at(i)**: It is also used to access the element stored at index i.
  - **front():** It returns the first element of the array.
  - **back():** It returns the last element of the array.
  - **size():** It tells us the number of elements in the array.
  - **fill(element):** It inserts the given value at every array index.
  - **begin():** It returns an iterator pointing to the first element of the array.
  - **end():** It returns an iterator pointing to the last element of the array.
  - **empty():** It tells us whether the array is empty or not.
  - **data():** It returns a pointer pointing to the first element of the array.
  - **swap(array_name):** It swaps the content of the two arrays.
  - **sort** is used to sort the elements in a range in ascending order. We need to add a header **algorithm** to use it

```cpp
#include<iostream>
#include<array>
#include<algorithm>
using namespace std;
int main()
{
 array < int, 5 > arr { 1, 2, 3, 4, 5 };
 for (int i = 0; i < 5; i++)
 {
  cout << arr[i] << " ";
 }
 cout << '\n';
 cout << "Size of array is " << arr.size() << '\n';
 cout << "First element of array is " << arr.front() << '\n';
 cout << "Last element of array is " << arr.back() << '\n';
 cout << "Second element of array is " << arr.at(1) << '\n';
 cout << "First element of array using data method is " << * (arr.data()) << '\n';
 arr.fill(10);
 cout << "The new array is ";
 for (int i = 0; i < 5; i++)
 {
  cout << arr[i] << " ";
 }
 cout << '\n';
```

```
1 2 3 4 5
Size of array is 5
First element of array is 1
Last element of array is 5
Second element of array is 2
First element of array using data method is 1
The new array is 10 10 10 10 10
```

```
if (arr.empty() == false)
{
 cout << "array is not empty!";
}
//taking values of elements from user
 for(int i = 0; i < arr.size() ; i++)
 {
 cout << "\nEnter value of arr[" << i << "]";
    cin >> arr[i];
 }
 cout<<"\nArray elements:";
 for(auto it=arr.begin();it!=arr.end();it++)
    cout<<*it<<" ";
 sort(arr.begin(),arr.end());
 cout<<"\nArray elements after sorting:";
 for(const &i:arr)
    cout<<i<<" ";
 return 0;
}
```

```
array is not empty!
Enter value of arr[0]34

Enter value of arr[1]22

Enter value of arr[2]1

Enter value of arr[3]56

Enter value of arr[4]78

Array elements:34 22 1 56 78
Array elements after sorting:1 22 34 56 78
```

Standard Template Library – Cont'd
Containers - Sequence

- **Vectors**
  - Vectors are part of the C++ Standard Template Library.
  - To use vectors, we need to include the vector header file in our program.

    #include <vector>
  - These are containers that can store elements.
  - **It holds objects of the same data type**.
  - The elements are stored in the containers in **sequential order**.
  - The main advantage of a container is that it changes their size dynamically, in other words, it can be considered as a **dynamic array**.
  - The elements in a container are stored in **contiguous storage** locations allowing the elements to be accessed using pointers.
  - Vectors can **resize automatically** whenever an element is added or removed.

Standard Template Library – Cont'd
Containers - Sequence

- **Declaration of a vector in C++:**
- **syntax:**

  std::vector<data_type> vector_name;

  – std::vector is the namespace for vectors
  – <data_type> is the type of data the vector will store, such as int, double, or std::string
  – vector_name is the name of the vector
- For example, to declare a vector of integers named numbers, we would write:

  std::vector<int> numbers;
- This vector can initially hold no elements, but its size can be increased or decreased as needed.

Standard Template Library – Cont'd

Containers - Sequence

- **C++ Vector Initialization**
- There are different ways to initialize a vector in C++.
- **Using an initializer list:**
  - An initializer list is a comma-separated list of values enclosed in curly braces. It is the simplest and most common way to initialize a vector.
  - **std::vector<int> numbers = {1, 2, 3, 4, 5};**
  - **std::vector<int> numbers2 {1, 2, 3, 4, 5};**
    - The above codes initializes a vector named numbers with the values 1, 2, 3, 4, and 5.
- **Using the fill() algorithm:**
  - The fill() algorithm fills all the elements of a vector with a specified value.
  - **std::vector<int> numbers(10, 0);**
    - This code initializes a vector of 10 integer elements, all with the value 0.

Standard Template Library – Cont'd

Containers - Sequence

- **Functions of C++ Vector**
- The functions related to vector can be categorized into three types,
  - **Iterators:** The functions under iterators are **used to iterate through the vector**. These functions act as a pointer. Five types of iterators are available. They are input, forward, output, random, and bidirectional. Some of the iterator's functions are begin(), end(), rbegin(), rend().
  - **Modifiers:** The modifier functions are **used to modify the vector** such as modifying the data type of the vector. Some example of functions under this category are assign(), push_back(), pop_back() etc.
  - **Capacity:** The capacity category functions are **used to modify the capacity or size of the vector**. Some example of this category of functions are size(), max_size() capacity() etc.

Standard Template Library – Cont'd

Containers - Sequence

- **Modifiers:**
  - **assign() –** It assigns a new value to the existing elements of the vector.
  - **push_back()** – It pushes the element from back in the vector.
  - **pop_back()** – It removes elements from the back of the vector.
  - **insert()** – It inserts an element before a specified element in the vector.
  - **erase()** – It is used to remove elements from a specified element or a range in the vector.
  - **swap()** – It is used to swap the contents of two vectors of the same datatype. The sizes of vectors may differ.
  - **clear() –** It is used to remove all the elements from the vector.

```
The vector contains: 1 2 3 4 5
After pushing 10,The vector contains: 1 2 3 4 5 10
The last element is: 10
After deleting last elements,The vector contains: 1 2 3 4 5
```

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   vector<int> Nums={1,2,3,4,5};
   cout << "\nThe vector contains: ";
   for (int i = 0; i < Nums.size(); i++)
      cout << Nums[i] << " ";
    // inserting 10 to the last position of vector
   Nums.push_back(10);
   cout << "\nAfter pushing 10,The vector contains: ";
   for (int i = 0; i < Nums.size(); i++)
      cout << Nums[i] << " ";
   int n = Nums.size();
   cout << "\nThe last element is: " << Nums[n - 1];
   // removing the last element from vector
   Nums.pop_back();
  // printing the vector  contents
   cout << "\nAfter deleting last elements,The vector contains: ";
   for (int i = 0; i < Nums.size(); i++)
      cout << Nums[i] << " ";
```

```
// inserting 7 at the position 3 of vector
Nums.insert(Nums.begin()+2, 7);
cout << "\nAfter inserting 7 at position 3,The vector contains: ";
for (int i = 0; i < Nums.size(); i++)
    cout << Nums[i] << " ";
// removing the second element
Nums.erase(Nums.begin()+1);
cout << "\nAfter removing the second element,The vector contains: ";
for (int i = 0; i < Nums.size(); i++)
    cout << Nums[i] << " ";
// erasing the vector
Nums.clear();
//printing the vector after erasing it
cout << "\nVector size after erase(): " << Nums.size();
// Creating two vectors of integer type
vector<int> Nums1;
Nums1.assign(4, 3);
vector<int> Nums2={1,2,3};
//Pushing values in vector
Nums2.push_back(5);
Nums2.push_back(6);
```

```
After inserting 7 at position 3,The vector contains: 1 2 7 3 4 5
After removing the second element,The vector contains: 1 7 3 4 5
Vector size after erase(): 0
```

```
  //printing vector Nums1
   cout << "\n\nVector 1 is: ";
   for (const int& i : Nums1)
   {
      cout << i << "  ";
   }
   //printing vector 'Nums2'
   cout << "\nVector 2 is: ";
   for (int i = 0; i < Nums2.size(); i++)
      cout << Nums2[i] << " ";
   // Swaping vectors Nums1 and Num2
   Nums1.swap(Nums2);
    //Printing vector Nums1 after swapping with Nums2
   cout << "\nAfter Swap \nVector 1 is: ";
   for (int i = 0; i < Nums1.size(); i++)
      cout << Nums1[i] << " ";
    //printing vector Nums2 after swapping with Nums1
    cout << "\nVector 2 is: ";
    for (int i = 0; i < Nums2.size(); i++)
       cout << Nums2[i] << " ";
    return 0;
}
```

```
Vector 1 is: 3  3  3  3
Vector 2 is: 1 2 3 5 6
After Swap
Vector 1 is: 1 2 3 5 6
Vector 2 is: 3 3 3 3
```

Standard Template Library – Cont'd

Containers - Sequence

- **Iterators:**
  - **begin()** – It returns an iterator pointing to the first element in the vector.
  - **end()** – It returns an iterator pointing to the last element in the vector.
  - **rbegin()** – It returns a reverse iterator pointing to the last element in the vector.
  - **rend()** – It returns a reverse iterator pointing to the element preceding the first element in the vector. Basically considered as a reverse end.
  - **cbegin()** – It returns a constant iterator pointing to the first element in the vector.
  - **cend()** – It returns a constant iterator pointing to the element that follows the last element in the vector.
  - **crbegin()** – It returns a constant reverse iterator pointing to the last element in the vector.
  - **crend()** – It returns a constant reverse iterator pointing to the element preceding the first element in the vector.

```
using namespace std;
int main()
{
   vector<int> Nums; //Declaration of vector in C++
   for (int i = 1; i <=7 ; i++)
      Nums.push_back(i);
   vector<int>::iterator it;
    cout << "\nOutput of begin and end Function: ";
   for ( it = Nums.begin(); it != Nums.end(); ++it)
      cout << *it << " ";
   //Printing the Output of vector using iterators  begin() and end()
   cout << "\nOutput of begin and end Function: ";
   for (auto i = Nums.begin(); i != Nums.end(); ++i)
      cout << *i << " ";
  //Printing the Output of vector using iterators cbegin() and cend()
   cout << "\nOutput of cbegin and cend Function: ";
   for (auto i = Nums.cbegin(); i != Nums.cend(); ++i)
      cout << *i << " ";
```

```
Output of begin and end Function: 1 2 3 4 5 6 7
Output of begin and end Function: 1 2 3 4 5 6 7
Output of cbegin and cend Function: 1 2 3 4 5 6 7
```

```
  //Printing the output of vector using iterators rbegin() and rend()
   cout << "\nOutput of rbegin and rend Function: ";
   for (auto ir = Nums.rbegin(); ir != Nums.rend(); ++ir)
      cout << *ir << " ";
   //Printing the output of vector using iterators crbegin() and crend()
   cout << "\nOutput of crbegin and crend Function: ";
   for (auto ir = Nums.crbegin(); ir != Nums.crend(); ++ir)
      cout << *ir << " ";
   return 0;
}
```

```
Output of rbegin and rend Function: 7 6 5 4 3 2 1
Output of crbegin and crend Function: 7 6 5 4 3 2 1
```

Standard Template Library – Cont'd

Containers - Sequence

- **Capacity:**
  - **size()** – It returns the number of elements currently present in the vector.
  - **max_size()** – It returns the maximum number of elements that a vector can hold.
  - **capacity()** – It returns the storage capacity currently allocated to the vector.
  - **resize(n)** – It resizes the container to store 'n' elements.
  - **empty()** – It returns whether the container is empty or not.
- **Element access:**
  - **reference_operator [g]:** It returns a reference to the 'g' element in the vetor.
  - **at(g):** It returns a reference to the element at position 'g' in the vector.
  - **front():** It returns a reference to the first element in the vector.
  - **back():** It returns a reference to the last element in the vector.
  - **data():** It returns a direct pointer to the memory array which is used internally by the vector to store its owned elements.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  vector<int> Nums={1,2,3,4,5,6};
  //Printing size of the vector
   cout << "Size : " << Nums.size();
  //Printing the Capacity of the vector
   cout << "\nCapacity : " << Nums.capacity();
  //Printing the maximum size of the vector 'a'
   cout << "\nMax_Size : " << Nums.max_size();
  // resizing  the vector size to 4
   Nums.resize(4);
   // printing the vector size after resize() function
   cout << "\nSize : " << Nums.size();
   vector<int>::iterator it;
   cout<<"\nVector elements:";
   for ( it = Nums.begin(); it != Nums.end(); ++it)
      cout << *it << " ";
    // checks if the vector is empty or not
   if (Nums.empty() == true)
      cout << "\nVector is empty";
   else
      cout << "\nVector is not empty";
   return 0;
}
```

```
Size : 6
Capacity : 6
Max_Size : 4611686018427387903
Size : 4
Vector elements:1 2 3 4
Vector is not empty
```

```cpp
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   vector<int> Nums;
   for (int i = 1; i <= 10; i++)
      Nums.push_back(i * 2);
   cout<<"\nVector data:";
   for(const &i:Nums)
      cout<<i<<" ";
   cout << "\nUsing Reference operator Nums[2]:" << Nums[2];
   cout << "\nUsing at ,Nums.at(4): " << Nums.at(4);
   cout << "\nfront() element: " << Nums.front();
  cout << "\nback() element: " << Nums.back();
  // pointer to the first element
   int* pos = Nums.data();
  //printing the first element of vector using pointer 'pos'
   cout << "\nThe first element is " << *pos;
   return 0;
}
```

```
Vector data:2 4 6 8 10 12 14 16 18 20
Using Reference operator Nums[2]:6
Using at ,Nums.at(4): 10
front() element: 2
back() element: 20
The first element is 2
```

Standard Template Library – Cont'd

Containers - Sequence

- **Vector of Vectors in C++ STL**
  - Vector of Vectors is a two-dimensional vector with a variable number of rows where each row is considered as a vector.
  - Each index of vector stores a vector in it.
  - It can be accessed or traversed using iterators.
  - Basically, it can be considered as the array of vectors with dynamic properties.
  - **Syntax:**

    vector<vector<data_type>> vector_name;
  - **Example:**

    vector<vector<int> > a;

```
#include <iostream>
#include <vector>
using namespace std;
// Defining the rows and columns of
// vector of vectors
#define row 3
#define col 4
int main()
{
   // Initializing the vector of vectors
   vector<vector<int> > mat;
   int num ;

   // Inserting elements into vector
   for (int i = 0; i < row; i++)
   {
      // Vector to store column elements
      vector<int> vec1;
     //pushing values in vector vec1
      for (int j = 0; j < col; j++)
      {
         cout<<"Enter data:";cin>>num;
         vec1.push_back(num);
      }
       // for creating a 2D vector mat pushing 1D vector vec1 into it
      mat.push_back(vec1);
   }
```

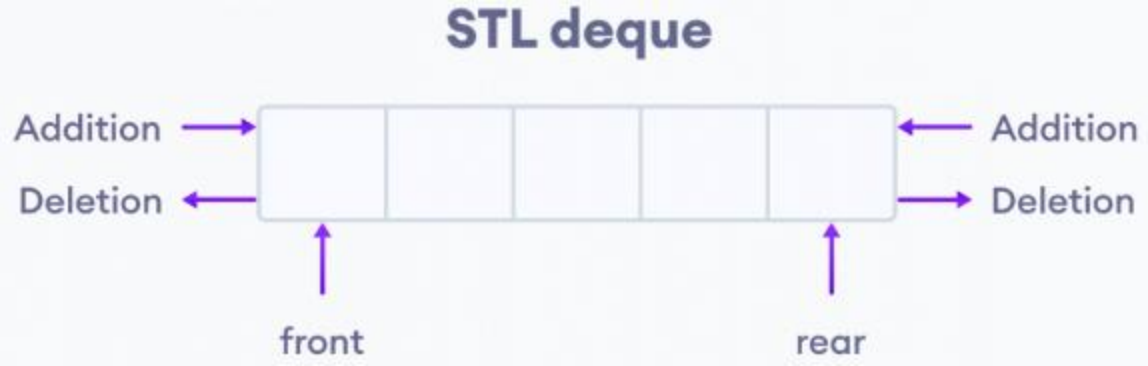
```
    // Displaying the 2D vector mat
    cout<<"2D vector contains:"<<"\n";
    for (int i = 0; i < mat.size(); i++)
    {
       for (int j = 0; j < mat[i].size(); j++)
          cout << mat[i][j] << " ";
       cout << endl;
    }
    //Deleting in 2D vector
    mat[2].pop_back();
    //inserting in 2D vector
    mat[1].push_back(100);
       cout<<"2D vector traversal after deletion:"<<"\n";
    //Traversing in 2D vector using iterator after deletion
    for (int i = 0; i < mat.size(); i++)
    {
       for ( auto it = mat[i].begin(); it != mat[i].end(); it++)
          cout << *it << " ";
       cout << endl;
    }
    return 0;
}
```

```
Enter data:1
Enter data:2
Enter data:3
Enter data:4
Enter data:5
Enter data:6
Enter data:7
Enter data:8
Enter data:9
Enter data:2
Enter data:8
Enter data:7
2D vector contains:
1 2 3 4
5 6 7 8
9 2 8 7
2D vector traversal after deletion:
1 2 3 4
5 6 7 8 100
9 2 8
```

Standard Template Library – Cont'd

Containers - Sequence

- Deque
  - In C++, the STL deque is a sequential container that **provides the functionality of a double-ended queue data structure**.
  - In a regular queue, elements are added from the rear and removed from the front.
  - However, in a deque, we can insert and remove elements from both the **front** and **rear**.



Standard Template Library – Cont'd

Containers - Sequence

- **Create C++ STL Deque**
  - In order to create a deque in C++, we first need to include the deque header file.

    #include <deque>
  - **syntax:**

    deque<type> dq;
    - Here, type indicates the data type we want to store in the deque.
  - **Example,**
    - // create a deque of integer data type

      deque<int> dq_integer;
    - // create a deque of string data type

      deque<string> dq_string;
- **Initialize a deque:**
  - **// method 1: initializer list**

    deque<int> deque1 = {1, 2, 3, 4, 5};
  - **// method 2: uniform initialization**

    deque<int> deque2 {1, 2, 3, 4, 5};

Standard Template Library – Cont'd

Containers - Sequence

- **Member Functions of Deque**
  - **push_back(element e)** inserts an element **e** at the back of the deque
  - **push_front(element e)** inserts the element **e** at the front of the deque.
  - **insert() method has three variations :**
    - **insert(iterator i, element e) :** Inserts element **e** at the position pointed by iterator **i** in the deque.
    - **insert(iterator i, int count, element e)** : Inserts element **e**, **count** number of times from the position pointed by iterator **i**.
    - **insert(iterator i, iterator first, iterator last)** : Inserts the element in the range [first,last] at the position pointed by iterator **i** in deque.
  - **pop_back()** removes an element from the back of the deque
  - **pop_front()** removes an element from the front of the deque
  - **empty()** returns Boolean true if the deque is empty, else Boolean false is returned.
  - **size()** returns the number of elements present in the deque
  - **max_size()** returns the number of element the given deque can hold.
  - **swap** function is used to swap elements of two deques.

```
#include<iostream>
#include<deque>
using namespace std;
void print(deque < int > q)
{
   deque < int > ::iterator it;
   for (it = q.begin(); it != q.end(); ++it)
      cout << * it << ' ';
   cout << '\n';
}
int main()
{
   deque < int > dq;
   for (int i = 0; i < 5; i++)
   {
      dq.push_back(i + 1);
   }
   cout<<"Deque:";
   print(dq);
   dq.push_front(10);
   cout << "After adding 10 in the front, the elements of deque are ";
   print(dq);
   cout << "Size of deque is " << dq.size() << "\n";
   cout << "First element of the deque " << dq.front() << "\n";
   cout << "Last element of the deque " << dq.back() << "\n";
```

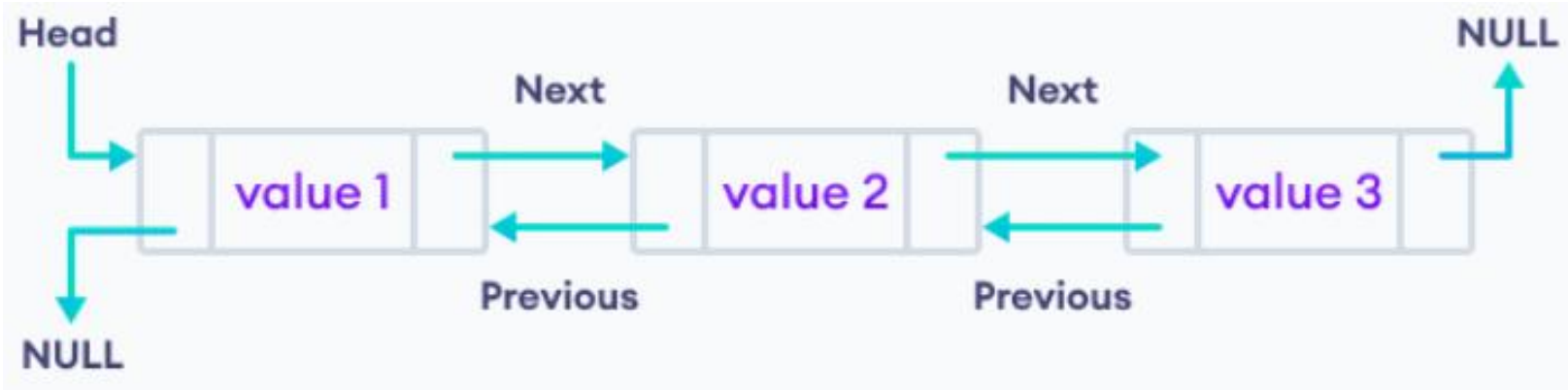
```
  dq.pop_front();
  cout << "After deleting 10 in the front, the elements of deque are ";
  print(dq);
  if (!dq.empty())
     cout << "Deque is not empty" << "\n";
  dq.push_back(100);
  cout << "After push back:";
  print(dq);
  dq.pop_back();
  cout << "After pop back:";
  print(dq);
  cout << "Size of deque is " << dq.size() << "\n";
  cout << "The elements of deque are ";
  print(dq);
  int a[]={1,2,3,4,5};
  dq.insert(dq.begin()+2,a,a+4);
  cout << "After Insertion :";
  print(dq);
  cout << "Size of deque is " << dq.size() << "\n";
 return 0;
}
```

```
Deque:1 2 3 4 5
After adding 10 in the front, the elements of deque are 10 1 2 3 4 5
Size of deque is 6
First element of the deque 10
Last element of the deque 5
After deleting 10 in the front, the elements of deque are 1 2 3 4 5
Deque is not empty
After push back:1 2 3 4 5 100
After pop back:1 2 3 4 5
Size of deque is 5
The elements of deque are 1 2 3 4 5
After Insertion :1 2 1 2 3 4 3 4 5
Size of deque is 9
```

Standard Template Library – Cont'd

Containers - Sequence

- **List:**
  - Lists are one of the sequence containers available in C++ STL that **store elements in a non-contiguous manner.**
  - It **permits iteration in both directions**.
  - Insert and erase operations anywhere inside the sequence are completed in constant time.
  - **List containers are constructed as doubly-linked lists**, which allow each of the elements, they contain to be stored in non-continuous memory locations.



Standard Template Library – Cont'd

Containers - Sequence

- **Create C++ STL List**
  - To create a list, we need to include the list header file in our program, **#include<list>**
  - **Syntax**:

    list<Type> list_name = {value1, value2, ...};
    - list - declares an STL container of type list
    - <Type> - the data type of the values to be stored in the list
    - list_name - a unique name given to the list
    - value1, value2, ... - values to be stored in the list
  - **Example**,

    list<int> numbers = {1, 2, 3, 4, 5}; // creates a list of integer type

    list<char> vowels = {'a', 'e', 'i', 'o', 'u'}; // creates a list of character type

Standard Template Library – Cont'd

Containers - Sequence

- **Functions on lists**
  - **push_back(element):** It pushes the value at the end of the list.
  - **push_front(element):** It adds a new element to the front of the list.
  - **pop_back():** It deletes the last element of the list.
  - **pop_front():** It deletes the first element of the list.
  - **front():** It returns the first element of the list.
  - **back():** It returns the last element of the list.
  - **begin():** It returns an iterator pointing to the first element of the list.
  - **end():** It returns an iterator pointing to the last element of the list.
  - **clear():** It deletes all the elements from the list.
  - **empty():** It tells us whether the list is empty or not.
  - **reverse():** It reverses the list.
  - **list1_name.merge(list2_name)**:it merges two sorted lists into a single sorted list

Standard Template Library – Cont'd

Containers - Sequence

- **Copying one list to another**

  list < int > list2 = list1; //copies list1 into list2

  – Any change in list2 will not affect list1.

  – The same can be done using reference but it will not create a copy, it will contain a reference to the original list.

  list < int > & list2 = list1; //list2 references list1

- **Inserting elements into a list**

  – The insert() function is used to add elements at any position in a list.

  – **Syntax:**

  – **insert(pos, no_of_elements, element);**

    - **pos:** Iterator pointing to the position where the element is to be inserted.
    - **no_of_elements:** Number of elements to insert. Default is 1.
    - **element:** The element which is to be inserted.

  – list < int > ::iterator it = l.begin();

  – l.insert(it, 5); // inserts 5 at 1st position

Standard Template Library – Cont'd

Containers - Sequence

- **Deleting elements from the list**
- The **erase()** function is used to delete a single element or a range of elements from a list.
- **Syntax:**
  - listName.erase(iterator pos) //to delete a single element
  - listName.erase(iterator first, iterator last) // to delete a range of elements
    - **pos:** Iterator pointing to the position of the element to be deleted.
    - **first:** Iterator pointing to the position of the first element in the range to be deleted.
    - **last:** Iterator pointing to the position of the last element in the range to be deleted. The last element doesn't get deleted.

```
list < int > ::iterator itr = l.begin();
l.erase(itr);
itr1 = demoList.begin();
itr2 = demoList.begin();
itr2++;
l.erase(itr1, itr2);
```

```
#include<iostream>
#include<list>
using namespace std;
void print(list < int > l)
{
  list < int > ::iterator itr;
  for (itr = l.begin(); itr != l.end(); ++itr)
    cout << * itr << ' ';
  cout << '\n';
}
int main()
{
  list < int > l;
  for (int i = 0; i < 5; ++i)
  {
    l.push_back(i);
  }
  cout<<"\nList elements:";
  print(l);
  cout << "First element " << l.front() << '\n';
  cout << "Last element " << l.back() << '\n';
  l.pop_front();
  cout<<"\nAfter pop front:";
  print(l);
```

```
l.pop_back();
cout<<"\nAfter pop back:";
print(l);
list < int > l2 = l;
cout<<"\nList1:";
print(l);
cout<<"\nList2:";
print(l2);
l.pop_front();
cout<<"\nList1:";
print(l);
cout<<"\nList2:";
print(l2);
list < int >& l3 = l;
cout<<"\nList1:";
print(l);
cout<<"\nList3:";
print(l3);
l.pop_front();
cout<<"\nList1:";
print(l);
cout<<"\nList2:";
print(l3);
l.push_front(1);
l.push_back(5);
```

```
list < int > ::iterator it = l.begin();
it++;
l.insert(it,35);
cout<<"\nList1:";
print(l);
l.reverse();
cout<<"\nList1:";
print(l);
return 0;
}
```

```
List elements:0 1 2 3 4
First element 0
Last element 4

After pop front:1 2 3 4

After pop back:1 2 3

List1:1 2 3

List2:1 2 3

List1:2 3

List2:1 2 3

List1:2 3

List3:2 3

List1:3

List2:3

List1:1 35 3 5

List1:5 3 35 1
```

Standard Template Library – Cont'd

Containers - Sequence

- **Forward_list:**
- **Forward_list** in STL is a sequential container.
- It is very **similar to list** in STL but the list stores both the location of the previous and the next elements of the list while forward_list **only stores the location of the next element**, thus it saves memory.
- Insertion and deletion are efficient in forward_list and take only constant time.
- It is **implemented as a singly LinkedList** in memory.
- Forward_lists are included in,

#include <forward_list> header file.

Standard Template Library – Cont'd

Containers - Sequence

- **Forward_list Declaration**
  - **Syntax:**

    forward_list<data_type> name;
    - **data_type:** data type of the elements to be stored in the list.
  - **Example:**
  - **forward_list<int> l;**
    - initializes a forward list of size 0 which stores integer values
- **Forward_list Initialization**

  forward_list<data_type> name={initial_values};
  - initial_values: optional parameter which initializes the list with the given values.
  - **Example:**
    - **forward_list<int> l = {1,2,3,4,5};**
      - initializes a forward list of size 5 with values 1,2,3,4,5

Standard Template Library – Cont'd

Containers - Sequence

- **Functions on forward_list**
  - **push_front(element):** It adds a new element to the front of the list.
  - **pop_front():** It deletes the first element of the list.
  - **insert_after(iterator,{values}):** It inserts the values after the given position.
  - **erase_after(iterator):** It deletes the values after the given position.
  - **remove(element):** It removes all the occurrences of the given element.
  - **front():** It returns the first element of the list.
  - **begin():** It returns an iterator pointing to the first element of the list.
  - **end():** It returns an iterator pointing to the last element of the list.
  - **max_size():** It returns the maximum number of elements that can be stored by the forward_list.
  - **clear()** :- This function clears the contents of forward list. After this function, the forward list becomes empty.
  - **empty():** It tells us whether the list is empty or not
  - **reverse():** It reverses the list.

Standard Template Library – Cont'd

Containers - Sequence

- **merge()** : This function is used to merge one forward list with other. If both the lists are sorted then the resultant list returned is also sorted.
- **operator "="** : This operator copies one forward list into other. The copy made in this case is deep copy.
- **sort()** : This function is used to sort the forward list.
- **unique()** : This function deletes the multiple occurrences of a number and returns a forward list with unique elements. The forward list should be sorted for this function to execute successfully.
- **swap()** : This function swaps the content of one forward list with other.

```
#include<iostream>
#include<forward_list>
using namespace std;
void print(forward_list <int> l)
{
   forward_list<int>::iterator itr;
   for(itr=l.begin();itr!=l.end();++itr)
      cout<<*itr<<' ';
   cout<<'\n';
}
int main()
{
   forward_list<int> l={1,2,3};
   for(int i=0;i<5;++i)
   {
      l.push_front(i);
   }
   print(l);
   cout<<"First element " <<l.front()<<'\n';
   l.pop_front();
   l.insert_after(l.begin(),{5,6,7});
   print(l);
   l.erase_after(l.begin());
   print(l);
```

```
4 3 2 1 0 1 2 3
First element 4
3 5 6 7 2 1 0 1 2 3
3 6 7 2 1 0 1 2 3
```

```
    l.remove(2);
    cout<<"List after removing all occurrences of 2: ";
    print(l);
    if(!l.empty())
    {
    cout<<"Max size of list "<<l.max_size()<<'\n';
    }
    l.sort();
    print(l);
    l.unique();
    print(l);
    return 0;
}
```

```
List after removing all occurrences of 2: 3 6 7 1 0 1 3
Max size of list 1152921504606846975
0 1 1 3 3 6 7
0 1 3 6 7
```

Standard Template Library – Cont'd

Associative containers

- Associative containers **implement sorted data structures** that can be quickly searched (O(log n) complexity).
  - **Set:** Collection of unique keys, sorted by keys (class template)
  - **Map:** Collection of key-value pairs, sorted by keys, keys are unique (class template).
  - **multiset:** Collection of keys, sorted by keys (class template)
  - **multimap:** Collection of key-value pairs, sorted by keys (class template)

Standard Template Library – Cont'd

Associative containers

- **Sets**
  - Sets in C++ are containers that **store unique elements** in a sorted manner.
  - The elements inside the set cannot be changed, once they are added to the set, they can only be inserted or deleted.
  - They are **implemented as BST (Binary Search Trees)** in memory.
  - Set is present in **#include<set>** header file.
  - The elements inside the set can be accessed using iterators.
  - **Set Declaration**
  - **Syntax:**

    set <data_type> name = {initial_values};

    - **data_type:** data type of the elements to be stored inside the set
    - **initial_values:** optional parameter which initializes the set with the given values

Standard Template Library – Cont'd

Associative containers

- **Note:** By default the set stores values in non-decreasing order. To store the values in non-increasing order, we use an inbuilt comparator function
- **set <data_type, greater <data_type>> name;**
- set <int> s;
  - initializes a set of size 0 which stores integer values arranged in non-decreasing order
- set <int> s = {10, 20, 30};
  - initializes a set having initial values as 10,20,30
- set <int, greater <int>> s;
  - initializes a set of size 0 which stores integer values arranged in non-increasing order

Standard Template Library – Cont'd

Associative containers

- **Methods on set**
  - **begin()**: This method returns an iterator that points to the first element in the set.
  - **end()**: This function returns an iterator that points to the theoretical position next to the last element of the set.
  - **empty()**: This set method is used for checking whether the set is empty or not.
  - **size()**: This function gives the number of elements present in a set .
  - **max_size()**: This method returns upper bound of elements in a set, i.e. the maximum number that a set can hold.
  - **rbegin()**: In contrary to the begin() method, this method returns a reverse iterator pointing to the last element of a set.
  - **rend()**: In contrary to the begin() method, this method returns a reverse iterator pointing to the logical position before the last element of a set.
  - **erase (iterator_position)**: This method when applied on a set, removes the element at the position pointed by the pointer given in the parameter.
  - **erase (const_n)**: This function directly deletes the value 'n' passed in the parameter from a set .
  - **insert (const_n)**: This function inserts a new element 'n' into the set .
  - **find(n)**: This method searches for the element 'n' in the set and returns an iterator pointing to the position of the found element. If the element is not found, it returns an iterator pointing at the end.
  - **count(const_n)**: This set method checks for the occurrence of the passed value 'n' and returns 0 or 1 if the element is found or not found respectively.
  - **clear()**: It removes all the elements present in a set.

```
#include<iostream>
#include<set>
using namespace std;
int main()
{
 set <int> s1;
 set <int, greater <int>> s2;
 for (int i = 0; i < 5; i++)
 {
  s1.insert(i + 1);
 }
 for (int i = 0; i < 5; i++)
 {
  s2.insert((i + 1) * 10);
 }
 set <int> ::iterator it;
 cout << "Set1 ";
 for (it = s1.begin(); it != s1.end(); it++)
  cout << * it << " ";
 cout << '\n';
 cout << "Set2 ";
 for (it = s2.begin(); it != s2.end(); it++)
  cout << * it << " ";
 cout << '\n';
```

```
s1.erase(1);
s2.erase(s2.begin(), s2.find(10));
cout << "After erasing element, size of set1 is: " << s1.size() << '\n';
int val = 4;
if (s1.find(val) != s1.end())
 cout << "The set1 contains " << val << endl;
else
 cout << "The set1 does not contains " << val << endl;
cout << "Elements of set1: ";
for (it = s1.begin(); it != s1.end(); it++)
 cout << * it << " ";
cout << '\n';
cout << "Elements of set2: ";
for (it = s2.begin(); it != s2.end(); it++)
 cout << * it << " ";
cout << '\n';
s1.clear();
if (s1.empty() == true)
{
 cout << "set1 is empty now!";
}
```

```
 s2.insert(20);
 s2.insert(30);
 s2.insert(40);
 s2.insert(10);
 s2.insert(10);
 cout << "\nElements of set2: ";
 for (it = s2.begin(); it != s2.end(); it++)
  cout << * it << " ";
 cout << '\n';
 return 0;
}
```

```
After erasing element, size of set1 is: 4
The set1 contains 4
Elements of set1: 2 3 4 5
Elements of set2: 10
set1 is empty now!
Elements of set2: 40 30 20 10
```

Standard Template Library – Cont'd

Associative containers

- **Maps** :
  - Maps are containers in STL that stores the elements as **key-value pairs**.
  - Each element has a **unique key value** and the **elements are stored in sorted order based on their key value**.
  - Keys cannot be changed, once they are added to the map, they can only be inserted or deleted, but the values can be altered.
  - They are implemented as **Red-Black Trees in memory**.
  - The map is present in **#include<map>** header file.
  - The elements inside the map can be accessed using iterators.

Standard Template Library – Cont'd

Associative containers

- **Map Declaration**
  - **Syntax:**
  - **map < key_data_type, value_data_type > name { initial_values };**
    - **key_data_type:** data type of the key to be stored inside the map
    - **value_data_type:** data type of the value to be stored inside the map
    - **initial_values:** optional parameter which initializes the map with the given values
  - **Note:** By default the map stores values in non-decreasing order sorted according to key. To store the values in non-increasing order, we use an inbuilt comparator function
  - **map < key_data_type, value_data_type, greater < data_type >> name;**
  - **Example:**
  - map < int, string > m;
    - initializes a map of size 0 which have key elements as integers and values as strings arranged in non-decreasing order based on keys
  - map < int, string > m = { { 1, "a" }, { 2, "ab" }, { 3, "abc" }
    - initializes a map having initial key-value pairs as {1, a}, {2, ab}, {3, abc}
  - map < int, string, greater < int >> m;
    - initializes a map of size 0 which which have key elements as integers and values as strings arranged in non-increasing order

Standard Template Library – Cont'd

Associative containers

- **Functions on maps**
  - **begin():** Returns an iterator to the first element of the map.
  - **end():** Returns an iterator to the element past the last element of the map.
  - **[key]:** Returns the value associated with the given key.
  - **size():** It tells us the size of the map.
  - **insert(pair):** Insert a pair in the map.
  - **erase(key) or erase(pos_iterator):** Delete an element from the map.
  - **find(key):** Returns an iterator pointing to the element, if the key is found else returns an iterator pointing to the end of the map.
  - **clear():** It deletes all the elements from the map
  - **empty():** It tells us whether the map is empty or not.

```cpp
#include<map>
using namespace std;
int main()
{
 map < int, string > m1;
 m1.insert(pair < int, string > (1, "apple"));
 map < int, string, greater < int >> m2;
 m2.insert(pair < int, string > (1, "apple"));
 m1[2] = "grapes";
 m1[3] = "banana";
 m1[4] = "guava";
 m2[2] = "grapes";
 m2[3] = "banana";
 m2[4] = "guava";

 map < int, string > ::iterator it;
 cout << "Map1\n";
 for (it = m1.begin(); it != m1.end(); it++)
  cout << it -> first << " " << it -> second << '\n';
 cout << '\n';
 cout << "Map2\n";
 for (it = m2.begin(); it != m2.end(); it++)
  cout << it -> first << " " << it -> second << '\n';
 cout << '\n';
```

```
Map1
1 apple
2 grapes
3 banana
4 guava

Map2
4 guava
3 banana
2 grapes
1 apple
```

```
 m1.erase(1);
 m2.erase(m2.find(1));
 cout << "After erasing element, size of map1 is " << m1.size() << '\n';
 cout << "After erasing element, size of map2 is " << m2.size() << '\n';
 int val = 3;
 if (m1.find(val) != m1.end())
  cout << "The map1 contains " << val << " as key" << endl;
 else
  cout << "The map1 does not contains " << val << " as key" << endl;
 cout << "Elements of map1\n";
 for (it = m1.begin(); it != m1.end(); it++)
  cout << it -> first << " " << it -> second << '\n';
 cout << '\n';
 cout << "Elements of map2\n";
 for (it = m2.begin(); it != m2.end(); it++)
  cout << it -> first << " " << it -> second << '\n';
 cout << '\n';

 m1.clear();
 if (m1.empty() == true)
 {
  cout << "Map1 is empty now!";
 }
 return 0;
}
```

```
After erasing element, size of map1 is 3
After erasing element, size of map2 is 3
The map1 contains 3 as key
Elements of map1
2 grapes
3 banana
4 guava

Elements of map2
4 guava
3 banana
2 grapes

Map1 is empty now!
```

Standard Template Library – Cont'd

Associative containers

- **Multisets :**
  - Multisets in C++ are **containers** that are very **similar to sets**.
  - Unlike sets, multisets **can store duplicate elements in a sorted manner**.
  - The **elements inside the multiset cannot be changed**, once they are added to the multiset, they can only be inserted or deleted.
  - A multiset is present in **#include<set>** header file.
  - The elements inside the multiset can be accessed using iterators.

Standard Template Library – Cont'd

Associative containers

- **Multiset Declaration**
  - **Syntax:**
  - **multiset <data_type> name = { initial_values };**
  - **data_type:** data type of the elements to be stored inside the multiset
  - **initial_values:** optional parameter which initializes the multiset with the given values
  - **Note:** By default the multiset stores values in non-decreasing order. To store the values in non-increasing order, we use an inbuilt comparator function
  - multiset <data_type, greater <data_type>> name;
- **Example:**
- **multiset <int> s;**
  - initializes a multiset of size 0 which stores integer values arranged in non-decreasing order
- **multiset <int> s = { 10, 20, 30 };**
  - initializes a multiset having initial values as 10,20,30
- **multiset <int, greater <int>> s;**
  - initializes a multiset of size 0 which stores integer values arranged in non-increasing order

Standard Template Library – Cont'd

Associative containers

- **Functions on multisets**
  - **begin():** Returns an iterator to the first element of the multiset.
  - **end():** Returns an iterator to the element past the last element of the multiset.
  - **size():** It tells us the size of the multiset.
  - **insert(element):** Inserts an element in the multiset.
  - **erase(value) or erase(start_iterator,end_iterator):** Delete elements from the multiset.
  - **find(element):** Returns an iterator pointing to the element, if the element is found else returns an iterator pointing to the end of the multiset.
  - **clear():** It deletes all the elements from the multiset
  - **empty():** It tells us whether the multiset is empty or not.

```
#include<iostream>
#include<set>
using namespace std;
int main()
{
 multiset <int> s1;
 multiset <int, greater <int>> s2;
 for (int i = 0; i < 5; i++)
 {
  s1.insert(i + 1);
 }
  for (int i = 0; i < 5; i++)
  {
     s2.insert((i + 1) * 10);
   }
 set <int> ::iterator it;
 for (it = s1.begin(); it != s1.end(); it++)
  cout << * it << " ";
 cout << '\n';
 for (it = s2.begin(); it != s2.end(); it++)
  cout << * it << " ";
 cout << '\n';
```

```
1 2 3 4 5
50 40 30 20 10
```

```
 s1.erase(1);
 s2.erase(s2.begin(), s2.find(10));
 cout << "After erasing element, size of set1 is " << s1.size() << '\n';
 int val = 4;
 if (s1.find(val) != s1.end())
  cout << "The set1 contains " << val << endl;
 else
  cout << "The set1 does not contains " << val << endl;
 cout << "New elements of set1 are ";
 for (it = s1.begin(); it != s1.end(); it++)
  cout << * it << " ";
 cout << '\n';
 s1.insert(5);
 s1.insert(4);
 s1.insert(31);
 s1.insert(1);
 cout << "New elements of set1 are ";
 for (it = s1.begin(); it != s1.end(); it++)
  cout << * it << " ";
 cout << '\n';
 s1.clear();
 if (s1.empty() == true)
 {
  cout << "set1 is empty!";
 }
 return 0;
}
```

```
After erasing element, size of set1 is 4
The set1 contains 4
New elements of set1 are 2 3 4 5
New elements of set1 are 1 2 3 4 4 5 5 31
set1 is empty!
```

Standard Template Library – Cont'd

Associative containers

- **Multimaps**
  - Multimaps are containers in STL that are **very similar to maps**, they store the elements as key-value pairs.
  - Unlike maps, multimaps can have duplicate keys.
  - The elements are stored in sorted order based on their key value.
  - Keys cannot be changed, once they are added to the map, they can only be inserted or deleted, but the values can be altered.
  - They are **implemented as Red-Black Trees in memory**.
  - The multimap is present in **#include<map>** header file. The elements inside the multimap can be accessed using iterators.

Standard Template Library – Cont'd

Associative containers

- **Multimap Declaration**
- **Syntax:**
- **multimapmap <key_data_type, value_data_type> name {initial_values};**
  - **key_data_type:** data type of the key to be stored inside the map
  - **value_data_type:** data type of the value to be stored inside the map
  - **initial_values:** optional parameter which initializes the map with the given values
- **Note:** By default the map stores values in non-decreasing order sorted according to key. To store the values in non-increasing order, we use an inbuilt comparator function
- multimap <key_data_type, value_data_type, greater <data_type>> name;
- **Example:**
- **multimap <int, string> m;**
  - initializes a multimap of size 0 which have key elements as integers and values as strings arranged in non-decreasing order based on keys
- **multimap <int, string> m = {{1, "a"}, {1, "b"}, {2, "ab"}, {3, "abc"}}**
  - initializes a multimap having initial key-value pairs as {1, a}, {1, b}, {2, ab}, {3, abc}
- **multimap <int, string, greater <int>> m;**
  - initializes a multimap of size 0 which which have key elements as integers and values as strings arranged in non-increasing order

Standard Template Library – Cont'd

Associative containers

- **Functions on multi maps:**
  - **begin():** Returns an iterator to the first element of the multimap.
  - **end():** Returns an iterator to the element past the last element of the multimap.
  - **size():** It tells us the size of the multimap.
  - **insert(pair):** Insert a pair in the multimap.
  - **erase(key_val) or erase(pos_iterator):** Delete all elements from the multimap with key_val as the key value or delete the element at the specified position.
  - **find(key):** Returns an iterator pointing to the element, if the key is found else returns an iterator pointing to the end of the multimap. If there are duplicate keys, returns an iterator to the first one.
  - **clear():** It deletes all the elements from the multimap
  - **empty():** It tells us whether the multimap is empty or not.

```
#include<iostream>
#include<map>
using namespace std;
void print1(multimap <int,string> m)
{
   multimap<int,string>::iterator itr;
   for (itr = m.begin(); itr != m.end(); itr++)
   cout << itr -> first << " " << itr -> second << '\n';
  cout << '\n';
}
void print2(multimap <int,string,greater<int>> m)
{
   multimap<int,string>::iterator itr;
   for (itr = m.begin(); itr != m.end(); itr++)
   cout << itr -> first << " " << itr -> second << '\n';
  cout << '\n';
}
int main()
{
 multimap <int, string> m1;
 m1.insert(pair <int, string> (1, "apple"));
 m1.insert(pair <int, string> (1, "banana"));
 m1.insert(pair <int, string> (2, "grapes"));
 m1.insert(pair <int, string> (3, "guava"));
 m1.insert(pair <int, string> (4, "kiwi"));
 cout<<"\nSize of multimap1:"<<m1.size();
  cout << "\nMultimap1\n";
  print1(m1);
multimap <int, string, greater <int>> m2;
m2.insert(pair <int, string> (1, "apple"));
m2.insert(pair <int, string> (1, "banana"));
m2.insert(pair <int, string> (2, "grapes"));
m2.insert(pair <int, string> (3, "guava"));
m2.insert(pair <int, string> (4, "kiwi"));
cout<<"\nSize of multimap2:"<<m2.size();
cout << "\nMultimap2\n";
print2(m2);
```

```
Size of multimap1:5
Multimap1
1 apple
1 banana
2 grapes
3 guava
4 kiwi


Size of multimap2:5
Multimap2
4 kiwi
3 guava
2 grapes
1 apple
1 banana
```

```
m1.erase(1); // delete all occurrences of 1
cout << "After erasing element, size of multimap1 is " << m1.size() << '\n';
 cout << "\nMultimap1 after erasing 1\n";
 print1(m1);

m2.erase(m2.find(1)); // Finds returns an iterator, so only value will be deleted i.e value at specified position
cout << "After erasing element, size of multimap2 is " << m2.size() << '\n';
cout << "\nMultimap2 after erasing 1\n";
 print2(m2);
int val = 3;
if (m1.find(val) != m1.end())
 cout << "The multimap1 contains " << val << " as key" << endl;
else
 cout << "The multimap1 does not contains " << val << " as key" << endl;
cout << "Elements of multimap1\n";
print1(m1);
cout << "Elements of multimap2\n";
print2(m2);
m1.clear();
if (m1.empty() == true)
 {
 cout << "Multimap1 is empty now!";
}
  if (m1.empty() == true)
   {
   cout << "Multimap1 is empty now!";
  }
  cout << "\nElements of multimap1\n";
  print1(m1);
  return 0;
 }
```

```
After erasing element, size of multimap1 is 3

Multimap1 after erasing 1
2 grapes
3 guava
4 kiwi

After erasing element, size of multimap2 is 4

Multimap2 after erasing 1
4 kiwi
3 guava
2 grapes
1 banana

The multimap1 contains 3 as key
Elements of multimap1
2 grapes
3 guava
4 kiwi

Elements of multimap2
4 kiwi
3 guava
2 grapes
1 banana
```

```
Multimap1 is empty now!
Elements of multimap1
```

Standard Template Library – Cont'd

Unordered associative containers

- The four unordered associative containers are,
  - unordered_set
  - unordered_map
  - unordered_multiset
  - unordered_multimap
- Note that the meanings implied by containers are same except one is strict about maintaining order among the objects stored whereas another one is not.
- This means that ordered ones store the keys in a sorted order, but the unordered ones have no such order.
- Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched (O(1) amortized, O(n) worst-case complexity).

Standard Template Library – Cont'd

Unordered associative containers

- **Unordered_sets :**
  - Unordered_sets in C++ are containers that **similar to sets**.
  - They **store unique elements in any order**.
  - The elements inside the unordered_set cannot be changed, once they are added to the set, they can only be inserted or deleted. Insertion in unordered_set is randomized.
  - They are implemented as a hash table in memory.
  - The element in the unordered_set acts as both key and value in the hash table.
  - All operations take O(1) time on average and O(N) time in the worst case.
  - Unordered_set is present in **#include<unordered_set>** header file.
  - The elements inside the unordered_set can be accessed using iterators.

Standard Template Library – Cont'd

Unordered associative containers

- **Unordered_set Declaration**
  - **Syntax:**
  - **unordered_set <data_type> name = {initial_values};**
  - **data_type:** data type of the elements to be stored inside the unordered_set
  - **initial_values:** optional parameter which initializes the unordered_set with the given values
- **Example:**
  - **unordered_set <int> s;**
    - initializes a unordered_set of size 0 which stores integer values
  - **unordered_set <int> s = {10, 20, 30};**
    - initializes a unordered_set having initial values as 10,20,30

Standard Template Library – Cont'd

Unordered associative containers

- **Functions on unordered_sets**
  - **begin():** Returns an iterator to the first element of the unordered_set.
  - **end():** Returns an iterator to the element past the last element of the unordered_set.
  - **size():** It tells us the size of the unordered_set.
  - **insert(element):** Inserts an element in the unordered_set.
  - **erase(value) or erase(start_iterator,end_iterator):** Delete elements from the unordered_set.
  - **find(element):** Returns an iterator pointing to the element, if the element is found else returns an iterator pointing to the end of the unordered_set.
  - **clear():** It deletes all the elements from the unordered_set.
  - **empty():** It tells us whether the unordered_set is empty or not.

```cpp
#include<iostream>
#include<unordered_set>
using namespace std;
void print(unordered_set<int> s)
{
  unordered_set <int> ::iterator it;
  for (it = s.begin(); it != s.end(); it++)
    cout << * it << " ";
  cout << '\n';
}
int main()
{
 unordered_set <int> s;
 for (int i = 0; i < 5; i++)
 {
  s.insert(i + 1);
 }
 cout << "\nUnordered_set: ";
 print(s);
 s.insert(0);
 s.insert(4);
 s.insert(3);
 s.insert(7);
 cout << "\nUnordered_set after insert: ";
 print(s);
```

```
Unordered_set: 5 4 3 1 2

Unordered_set after insert: 7 2 1 3 4 5 0
```

```
 s.erase(1);
 cout << "\nUnordered_set after erase(1): ";
 print(s);
  cout << "After erasing element, size of unordered_set is " << s.size() << '\n';
 s.erase(s.find(2));
 cout << "\nUnordered_set after erase(s.find(2)): ";
 print(s);
 cout << "After erasing element, size of unordered_set is " << s.size() << '\n';
 int val = 4;
 if (s.find(val) != s.end())
   cout << "The unordered_set contains " << val << endl;
 else
   cout << "The unordered_set does not contains " << val << endl;
 cout << "Elements of unordered_set ";
print(s);
 s.clear();
 if (s.empty() == true)
   {
   cout << "set is empty now!";
 }
 return 0;
}
```

```
Unordered_set after erase(1): 7 2 3 4 5 0
After erasing element, size of unordered_set is 6

Unordered_set after erase(s.find(2)): 7 3 4 5 0
After erasing element, size of unordered_set is 5
The unordered_set contains 4
Elements of unordered_set 7 3 4 5 0
set is empty now!
```

Standard Template Library – Cont'd

Unordered associative containers

- **Unordered_maps :**
  - Unordered_maps are containers in STL that **stores the elements as key-value pairs in any order**.
  - Each element has a unique key value.
  - Keys cannot be changed, once they are added to the map, they can only be inserted or deleted, but the values can be altered.
  - They are implemented as hash-table in memory.
  - The map is present in #include<unordered_map> header file.
  - The elements inside the map can be accessed using iterators.

Standard Template Library – Cont'd

Unordered associative containers

- **Unordered_map Declaration**
  - **Syntax:**
  - **unordered_map <key_data_type, value_data_type> name {initial_values};**
    - **key_data_type:** data type of the key to be stored inside the map
    - **value_data_type:** data type of the value to be stored inside the map
    - **initial_values:** optional parameter which initializes the map with the given values
- **Example:**
- **unordered_map <int, string> m;**
  - initializes a unordered_map of size 0 which have key elements as integers and values as strings
- **unordered_map <int, string> m = {{1, "a"}, {2, "ab"}, {2, "ab"}, {3, "abc" } }**
  - initializes a unordered_map having initial key-value pairs as {1, a}, {2, ab}, {3, abc}

Standard Template Library – Cont'd

Unordered associative containers

- **Functions on unordered_maps**
  - **begin():** Returns an iterator to the first element of the unordered_map.
  - **end():** Returns an iterator to the element past the last element of the unordered_map.
  - **[key]:** Returns the value associated with the given key.
  - **size():** It tells us the size of the unordered_map.
  - **insert(pair):** Insert a pair in the unordered_map.
  - **erase(key) or erase(pos_iterator):** Delete an element from the unordered_map.
  - **find(key):** Returns an iterator pointing to the element, if the key is found else returns an iterator pointing to the end of the unordered_map.
  - **clear():** It deletes all the elements from the unordered_map
  - **empty():** It tells us whether the unordered_map is empty or not.

```cpp
#include<iostream>
#include<unordered_map>
using namespace std;
void print(unordered_map <int, string> m)
{
   unordered_map <int, string> ::iterator it;
    for (it = m.begin(); it != m.end(); it++)
      cout << it -> first << " " << it -> second << '\n';
   cout << '\n';
}
int main()
{
 unordered_map <int, string> m;
 m.insert(pair <int, string> (1, "apple"));
 m[2] = "banana";
 m[3] = "guava";
 m[4] = "grapes";
 cout << "Unordered_map contains\n";
 print(m);
 m[1]="mango";
 m[2]="kiwi";
cout << "Unordered_map contains\n";
 print(m);
```

```
Unordered_map contains
4 grapes
3 guava
1 apple
2 banana

Unordered_map contains
4 grapes
3 guava
1 mango
2 kiwi
```

```
 m.erase(1);
 cout << "After erase,Unordered_map contains\n";
 print(m);
 m.erase(m.find(2));
 cout << "After erase(m.find(2)),Unordered_map contains\n";
 print(m);
 cout << "After erasing element, size of unordered_map is " << m.size() << '\n';
 int val = 3;
 if (m.find(val) != m.end())
  cout << "The unordered_map contains " << val << " as key" << endl;
 else
  cout << "The unordered_map does not contains " << val << " as key" << endl;
 cout << "Elements of unordered_map\n";
 print(m);

 m.clear();
 if (m.empty() == true) {
  cout << "Unordered_map is empty now!";
 }
 return 0;
}
```

```
After erase,Unordered_map contains
4 grapes
3 guava
2 kiwi

After erase(m.find(2)),Unordered_map contains
4 grapes
3 guava

After erasing element, size of unordered_map is 2
The unordered_map contains 3 as key
Elements of unordered_map
4 grapes
3 guava

Unordered_map is empty now!
```

Standard Template Library – Cont'd

Unordered associative containers

- **Unordered_multisets :**
  - Unordered_multisets in C++ are containers that are the **same as unordered_sets except for the fact that they can store duplicate values**.
  - They store elements in any order but duplicated elements come together.
  - The elements inside the unordered_multiset cannot be changed, once they are added to the set, they can only be inserted or deleted.
  - They are **implemented as a hash table in memory.**
  - The elements inside the unordered_multisets are organized into buckets.
  - The elements inside these buckets are inserted using hashing.
  - Also, the count of each value is stored, thus taking extra space.
  - When a duplicate key is added to the set, the count of the key is incremented in the hash table.
  - All operations take O(1) time on average and O(N) time in the worst case.
  - Unordered_multiset is present in **#include<unordered_set>** header file.
  - The elements inside the unordered_multiset can be accessed using iterators.

Standard Template Library – Cont'd

Unordered associative containers

- **Unordered_multiset Declaration**
  - **Syntax:**
  - **unordered_multiset <data_type> name = {initial_values};**
    - **data_type:** data type of the elements to be stored inside the unordered_set
    - **initial_values:** optional parameter which initializes the unordered_set with the given values
  - **Example:**
  - **unordered_multiset <int> s;**
    - initializes a unordered_multiset of size 0 which stores integer values
  - **unordered_multiset <int> s = {10, 20, 20, 30};**
    - initializes a unordered_multiset having initial values as 10,20,20,30

Standard Template Library – Cont'd

Unordered associative containers

- **Functions on unordered_multisets**
  - **begin():** Returns an iterator to the first element of the unordered_multiset.
  - **end():** Returns an iterator to the element past the last element of the unordered_multiset.
  - **size():** It tells us the size of the unordered_multiset.
  - **bucket_count():** It tells us the total number of buckets in the unordered_multimap.
  - **count(element):** It tells us the total number of occurrences of the element in the unordered_multiset.
  - **insert(element):** Inserts an element in the unordered_multiset.
  - **erase(value) or erase(start_iterator,end_iterator):** Delete elements from the unordered_multiset.
  - **find(element):** Returns an iterator pointing to the element, if the element is found else returns an iterator pointing to the end of the unordered_multiset.
  - **clear():** It deletes all the elements from the unordered_multiset.
  - **empty():** It tells us whether the unordered_multiset is empty or not.

```
#include<iostream>
#include<unordered_set>
using namespace std;
void print(unordered_multiset <int> s)
{
   unordered_multiset <int> ::iterator it;
   for (it = s.begin(); it != s.end(); it++)
   cout << * it << " ";
 cout << '\n';
}
int main() {
 unordered_multiset <int> s;
 for (int i = 0; i < 5; i++) {
   s.insert(i + 1);
 }
 cout << "\nUnordered_multiset contains:";
 print(s);
 s.insert(4);
 s.insert(5);
 s.insert(0);
cout << "\nAfter insert,Unordered_multiset contains: ";
 print(s);
```

```
s.erase(4);
cout << "\nAfter erase(4),Unordered_multiset contains: ";
print(s);
s.erase(s.find(2));
cout << "\nAfter erase(find(2)),Unordered_multiset contains: ";
print(s);
cout << "After erasing the element, the size of unordered_multiset is " << s.size() << '\n';
int val = 4;
if (s.find(val) != s.end())
  cout << "The unordered_multiset contains " << val << endl;
else
  cout << "The unordered_multiset does not contain " << val << endl;
cout << "Elements of unordered_multiset ";
print(s);
cout << "No of occurrences of 5 in unordered_multiset is " << s.count(5) << "\n";
int n = s.bucket_count();
cout << "set has " << n << " buckets.\n";
for (int i = 0; i < n; ++i)
  {
  cout << "bucket " << i << " contains:";
  for (auto it = s.begin(i); it != s.end(i); ++it)
    cout << " " << * it;
  cout << "\n";
 }
 s.clear();
 if (s.empty() == true)
 {
   cout << "set is empty now!";
 }
 return 0;
}
```

```
After erase,Unordered_map contains
4 grapes
3 guava
2 kiwi

After erase(m.find(2)),Unordered_map contains
4 grapes
3 guava

After erasing element, size of unordered_map is 2
The unordered_map contains 3 as key
Elements of unordered_map
4 grapes
3 guava

Unordered_map is empty now!
```

Standard Template Library – Cont'd

Unordered associative containers

- **Unordered_multimaps :**
  - Unordered_multimaps are containers in STL that are very **similar to unordered_map** **except for the fact that they store duplicate keys.**
  - The elements inside unordered_multimaps are stored as key-value pairs in any order.
  - Keys cannot be changed, once they are added to the map, they can only be inserted or deleted, but the values can be altered.
  - They are implemented as hash-table in memory.
  - When a duplicated key-value pair is inserted into the map, the count of the key-value pair is incremented.
  - The map is present in **#include<unordered_map>** header file.
  - The elements inside the map can be accessed using iterators.

Standard Template Library – Cont'd

Unordered associative containers

- **Unordered_multimap Declaration**
  - **Syntax:**
  - **unordered_multimap <key_data_type, value_data_type> name { initial_values };**
    - **key_data_type:** data type of the key to be stored inside the map
    - **value_data_type:** data type of the value to be stored inside the map
    - **initial_values:** optional parameter which initializes the map with the given values
  - **Example:**
  - **unordered_multimap <int, string> m;**
    - initializes a unordered_multimap of size 0 which have key elements as integers and values as strings
  - **unordered_multimap <int, string> m = {{1, "a"}, {2, "b"}, {2, "ab"}, {2, "ab"}, {3, "abc"}**
    - initializes a unordered_multimap having initial key-value pairs as {1, a}, {2, b}, {2, ab}, {3, abc}

Standard Template Library – Cont'd

Unordered associative containers

- **Functions on unordered_multimaps:**
  - **begin():** Returns an iterator to the first element of the unordered_multimap.
  - **end():** Returns an iterator to the element past the last element of the unordered_multimap.
  - **size():** It tells us the size of the unordered_multimap.
  - **insert(pair):** Insert a pair in the unordered_multimap.
  - **erase(value) or erase(start_iterator,end_iterator):** Delete elements from the unordered_multimap.
  - **find(key):** Returns an iterator pointing to the element, if the key is found else returns an iterator pointing to the end of the unordered_multimap.
  - **bucket_count():** It tells us the total number of buckets in the unordered_multimap.
  - **count(key):** It tells us the total number of occurrences of the key in the unordered_multimap.
  - **clear():** It deletes all the elements from the unordered_multimap
  - **empty():** It tells us whether the unordered_multimap is empty or not.

```cpp
#include<iostream>
#include<unordered_map>
using namespace std;
void print(unordered_multimap <int, string> m)
{
    unordered_multimap <int, string> ::iterator it;
   for (it = m.begin(); it != m.end(); it++)
   cout << it -> first << " " << it -> second << '\n';
 cout << '\n';
}
int main()
{
 unordered_multimap <int, string> m;
 m.insert(pair <int, string> (1, "apple"));
 m.insert(pair <int, string> (2, "banana"));
 m.insert(pair <int, string> (2, "grapes"));
 m.insert(pair <int, string> (3, "guava"));
 m.insert(pair <int, string> (4, "kiwi"));
 m.insert(pair<int,string>(2,"mango"));
 cout << "Unordered_multimap contains\n";
 print(m);
```

```
Unordered_multimap contains
4 kiwi
3 guava
1 apple
2 mango
2 grapes
2 banana

After erase(1),Unordered_multimap contains
4 kiwi
3 guava
2 mango
2 grapes
2 banana
```

```
m.erase(1);
cout << "After erase(1),Unordered_multimap contains\n";
print(m);
m.erase(m.find(2));
cout << "After erase(find(2)),Unordered_multimap contains\n";
print(m);
cout << "After erasing element, size of unordered_multimap is " << m.size() << '\n';
int val = 3;
if (m.find(val) != m.end())
 cout << "The unordered_multimap contains " << val << " as key" << endl;
else
 cout << "The unordered_multimap does not contains " << val << " as key" << endl
cout << "Elements of unordered_multimap\n";
print(m);
cout << "No of occurrences of 5 in unordered_multimap is " << m.count(5) << "\n";
int n = m.bucket_count();
cout << "map has " << n << " buckets.\n";
for (int i = 0; i < n; ++i)

 cout << "bucket " << i << " contains:";
 for (auto it = m.begin(i); it != m.end(i); ++it)
  cout << " " << it -> first << " " << it -> second;
 cout << "\n";
}
```

```
 m.clear();
 if (m.empty() == true)
{
  cout << "Unordered_multimap is empty now!";
 }
 return 0;
}
```

```
After erase(1),Unordered_multimap contains
4 kiwi
3 guava
2 mango
2 grapes
2 banana

After erase(find(2)),Unordered_multimap contains
4 kiwi
3 guava
2 grapes
2 banana

After erasing element, size of unordered_multimap is 4
The unordered_multimap contains 3 as key
Elements of unordered_multimap
4 kiwi
3 guava
2 grapes
2 banana
```

```
No of occurrences of 5 in unordered_multimap is 0
map has 7 buckets.
bucket 0 contains:
bucket 1 contains:
bucket 2 contains: 2 grapes 2 banana
bucket 3 contains: 3 guava
bucket 4 contains: 4 kiwi
bucket 5 contains:
bucket 6 contains:
Unordered_multimap is empty now!
```

Standard Template Library – Cont'd

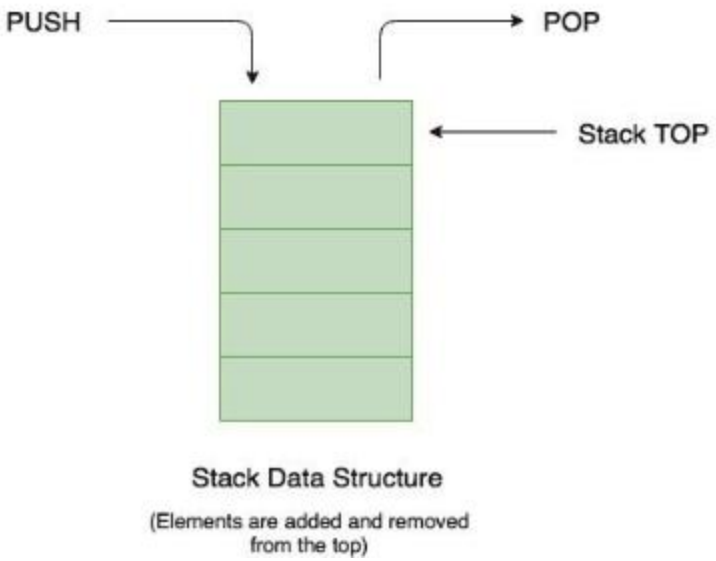
Container adapters

- **Container adapters :**
  - Container adapters provide a different interface for sequential containers.
    - **stack:** Adapts a container to provide stack (LIFO data structure) (class template).
    - **queue:** Adapts a container to provide queue (FIFO data structure) (class template).
    - **priority_queue:** Adapts a container to provide priority queue (class template).

Standard Template Library – Cont'd

Container adapters

- **Stacks :**
  - Stacks are a type of container adaptors with **LIFO(Last In First Out)** type of working, where a new element is added at one end (top) and an element is removed from that end only.
  - Container objects hold data of a similar data type.
  - We can create a stack from various sequence containers.
  - If no container is provided, the deque container will be used by default.
  - Container adapters don't support iterators, so it can't be used to manipulate data.



Stack Data Structure

(Elements are added and removed from the top)

Standard Template Library – Cont'd

Container adapters

- **Definition of std::stack from <stack> header file**
  - template <class T, class Container = deque<T> > class stack;
    - where,
      T = Type of the element of any type(user-defined / in-built).
    - Container = Type of the underlying container object.
- **Creation of Stack Objects**
- stack <data_type> object_name;
  - To create a stack, we must include the **<stack> header file** in our code.
- **Example:**
- stack<int> integer_stack;
  - create a stack of integers
- stack<string> string_stack;
  - create a stack of strings

Standard Template Library – Cont'd

Container adapters

- **Functions on queues**
  - **push(element):** It pushes the value at the top of the stack.
  - **pop():** It deletes the topmost element of the stack.
  - **size():** It tells us the size of the stack.
  - **top():** It returns the topmost element of the stack.
  - **swap(stack_name)**: It swaps the elements of the two stacks.
  - **empty():** It tells us whether the stack is empty or not.
  - **emplace():** Function inserts a new element into the stack container, the new element is added on top of the stack.

Standard Template Library – Cont'd

Container adapters

- **Copying one stack to another**
  - stack < int > stack2 = stack1;
    - copies stack1 into stack2 Any change in stack2 will not affect stack1.
  - The same can be done using reference but it will not create a copy, it will contain a reference to the original stack.
  - stack < int > & stack2 = stack1;
    - stack2 references stack1

```
#include <iostream>
#include <stack>
using namespace std;
void displayStack(stack <int> s)
{
    while (!s.empty())
    {
        cout << s.top() << ' ';
        s.pop();
    }
}
int main()
{
    std::stack <int> s1;
    s1.push(50);
    s1.push(40);
    s1.push(30);
    s1.push(20);
    s1.push(10);

    cout << "\nS1 stack : \n";
    displayStack(s1);
    cout << "\nS1 size() : " << s1.size();
    cout << "\nS1 top() : " << s1.top();
    cout << "\nS1 pop() operation done.";
    s1.pop();
```

```
S1 stack :
10 20 30 40 50
S1 size() : 5
S1 top() : 10
S1 pop() operation done.
```

```
cout << "\nS1 stack (after pop): \n";
displayStack(s1);
cout << "\nIs S1 empty() : ";
bool t_ = s1.empty();
cout<<boolalpha<<t_;
cout << "\nS1 emplace() operation done.";
s1.emplace(10);
cout << "\nS1 stack (after emplacing element : 10): \n";
displayStack(s1);

stack <int> s2;
s2.push(500);
s2.push(400);
s2.push(300);
s2.push(200);
s2.push(100);

cout << "\nS2 stack : \n";
displayStack(s2);
cout << "\nS2 size() : " << s2.size();
s2 = s1;
cout << "\nS2 stack (using = operator overloading): \n";
displayStack(s2);
```

```
S1 stack (after pop):
20 30 40 50
Is S1 empty() : false
S1 emplace() operation done.
S1 stack (after emplacing element : 10):
10 20 30 40 50
S2 stack :
100 200 300 400 500
S2 size() : 5
S2 stack (using = operator overloading):
10 20 30 40 50
```

```
    stack <int> s3;
    s3.push(55);
    s3.push(44);
    s3.push(33);
    cout << "\nStack s1 Elements before swapping : \n";
    displayStack(s1);
    cout << "\nStack s3 Elements before swapping : \n";
    displayStack(s3);
    s3.swap(s1);
    cout << "\nStack s1 Elements after swapping : \n";
    displayStack(s1);
    cout << "\nStack s3 Elements after swapping : \n";
    displayStack(s3);
}
```

```
Stack s1 Elements before swapping :
10 20 30 40 50
Stack s3 Elements before swapping :
33 44 55
Stack s1 Elements after swapping :
33 44 55
Stack s3 Elements after swapping :
10 20 30 40 50
```

Standard Template Library – Cont'd

Container adapters

- **Queue:**
  - Queues are a type of container adaptors that operate in a **first in first out (FIFO)** type of arrangement.
  - Elements are inserted at the back (end) and are deleted from the front.
  - Queues use an encapsulated object of deque or list (sequential container class) as its underlying container, providing a specific set of member functions to access its elements.

Standard Template Library – Cont'd

Container adapters

- **Definition in <queue>**
  - **std::queue** is defined as follows under <queue> header file:

```
template <class T, class Container = deque<T> > class qu
```

- **Syntax:**
  - queue < data_type > name;
  - **Example:**
  - queue < int > q;
    - initializes a queue of size 0 which stores integer values;

Standard Template Library – Cont'd

Container adapters

- **Functions on queues**
  - **push(element):** It pushes the value at the end of the queue.
  - **pop():** It deletes the first element from the queue.
  - **size():** It tells us the size of the queue.
  - **front():** It returns the first element of the queue.
  - **back():** It returns the last element of the queue.
  - **swap(queue_name):** It swaps the elements of the two queues.
  - **empty():** It tells us whether the queue is empty or not.
  - **emplace(x)**: **Places** the element **x** at the back of the queue.

Standard Template Library – Cont'd
Container adapters

- **Copying one queue to another**
  - queue < int > q2 = q1;
    - copies queue1 into queue2 Any change in q2 will not affect q1.
  - The same can be done using reference but it will not create a copy, it will contain a reference to the original queue.
  - queue < int > & q2 = q1;
    - q2 references q1

```
#include <iostream>
#include <queue>
using namespace std;
void display(queue <int> q)
{
   queue <int> c = q;
   while (!c.empty())
   {
      cout << " " << c.front();
      c.pop();
   }
   cout << "\n";
}
int main()
{
   queue <int> a;
   a.push(10);
   a.push(20);
   a.push(30);
   cout << "The queue a is :";
   display(a);
   cout << "is a.empty() :" << boolalpha<<a.empty() << "\n";
   cout << "a.size() : " << a.size() << "\n";
   cout << "a.front() : " << a.front() << "\n";
   cout << "a.back() : " << a.back() << "\n";
   cout << "a.pop() : ";
   a.pop();
   display(a);
   a.push(40);
   cout << "The queue a is :";
   display(a);
   return 0;
}
```

```
The queue a is : 10 20 30
is a.empty() :false
a.size() : 3
a.front() : 10
a.back() : 30
a.pop() :  20 30
The queue a is : 20 30 40
```

Standard Template Library – Cont'd

Container adapters

- **priority queue :**
- A **C++ priority queue** is a type of container adapter, specifically designed such that the first element of the queue is either the greatest or the smallest of all elements in the queue, and elements are in non-increasing or non-decreasing order (hence we can see that each element of the queue has a priority {fixed order}).
- In C++ STL, the top element is always the greatest by default.
- We can also change it to the smallest element at the top.
- Priority queues are built on the top of the max heap and use an array or vector as an internal structure.
- In simple terms, **STL Priority Queue** is the implementation of Heap Data Structure.

Standard Template Library – Cont'd

Container adapters

- **Definition in <queue>**
  - **priority_queue** is defined as follows under <queue> header file:

```
template <class T, class Container = vector<T>, class
  Compare = less<typename Container::value_type>
  > class priority_queue;
```

- **Declaration**
  - A **priority queue** with name **a** storing integers is declared as follows:

```
priority_queue<int> a;
```

Standard Template Library – Cont'd

Container adapters

- Heaps are of two types:
- **Max-heap**: When the elements are stored in non-increasing order and the greatest element will be the one that will be deleted first, i.e largest element has the highest priority. By default, C++ creates a max-heap.
  Syntax:
  priority_queue < data_type > name;
- **Example:**
- priority_queue < int > pq; //initializes a priority queue (max-heap) of size 0 which stores integer values
- **Min-heap**: When the elements are stored in non-decreasing order and the smallest element will be the one that will be deleted first, i.e smallest element has the highest priority.
  Syntax:
  priority_queue<data_type,vector<int>,greater<int>> name;
- **Example:**
- priority_queue < int, vector < int > , greater < int >> pq; //initializes a priority queue (min-heap) of size 0 which stores integer values

Standard Template Library – Cont'd

Container adapters

- **Functions on priority_queues**
  - **push(element):** It pushes the value at the end of the queue. After that, the elements are reordered according to the priority.
  - **pop():** It deletes the highest priority element from the queue. If it's a min-heap, the smallest element gets deleted otherwise maximum element gets deleted.
  - **size():** It tells us the size of the queue.
  - **top():** It returns the highest priority element of the queue.
  - **swap(queue_name):** It swaps the elements of the two queues.
  - **empty():** It tells us whether the queue is empty or not.

```
#include <iostream>
#include <queue>
using namespace std;
void display(priority_queue <int> a)
{
   priority_queue <int> c = a;
   while (!c.empty())
   {
      cout << '\t' << c.top();
      c.pop();
   }
   cout << '\n';
}
int main ()
{
   priority_queue <int> a;
   a.push(10);
   a.push(30);
   a.push(20);
   a.push(5);
   a.push(1);
   cout << "\nThe priority queue a is : ";
   display(a);
   cout << "\na.size() : " << a.size();
   cout << "\na.top() : " << a.top();
   cout << "\na.pop() : ";
   a.pop();
   display(a);
   return 0;
}
```

```
The priority queue a is :       30      20      10      5       1

a.size() : 5
a.top() : 30
a.pop() :       20      10      5       1
```

	Array	Vector	Deque	List	Forward List	Associative Containers	Unordered Containers
Available since	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Typical internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree	Hash table
Element type	Value	Value	Value	Value	Value	Set: value Map: key/value	Set: value Map: key/value
Duplicates allowed	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)	Forward (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere	Everywhere
Random access available	Yes	Yes	Yes	No	No	No	Almost
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast	Very fast
Inserting/removing invalidates iterators	—	On reallocation	Always	Never	Never	Never	On rehashing
Inserting/removing references, pointers	—	On reallocation	Always	Never	Never	Never	Never
Allows memory reservation	—	Yes	No	—	—	—	Yes (buckets)
Frees memory for removed elements	—	Only with `shrink_to_fit()`	Sometimes	Always	Always	Always	Sometimes
Transaction safe (success or no effect)	No	Push/pop at the end	Push/pop at the beginning and the end	All insertions and all erasures	All insertions and all erasures	Single-element insertions and all erasures if comparing doesn't throw	Single-element insertions and all erasures if hashing and comparing don't throw

Standard Template Library – Cont'd

Containers

Container	Description	Header file	iterator
vector	vector is a class that creates a dynamic array allowing insertions and deletions at the back.	<vector>	Random access
list	list is the sequence containers that allow the insertions and deletions from anywhere.	<list>	Bidirectional
deque	deque is the double ended queue that allows the insertion and deletion from both the ends.	<deque>	Random access
set	set is an associate container for storing unique sets.	<set>	Bidirectional
multiset	Multiset is an associate container for storing non- unique sets.	<set>	Bidirectional

Standard Template Library – Cont'd

Containers

Container	Description	Header file	iterator
map	Map is an associate container for storing unique key-value pairs, i.e. each key is associated with only one value(one to one mapping).	<map>	Bidirectional
multimap	multimap is an associate container for storing key- value pair, and each key can be associated with more than one value.	<map>	Bidirectional
stack	It follows last in first out(LIFO).	<stack>	No iterator
queue	It follows first in first out(FIFO).	<queue>	No iterator
Priority-queue	First element out is always the highest priority element.	<queue>	No iterator