SATHYABAMA UNIVERSITY FACULTY OF ELECTRICAL AND ELECTRONICS

OBJECT ORIENTED PROGRAMMING (SCS1202)

UNIT 5 I/O AND LIBRARY ORGANIZATION

I/O Stream - File I/O - Exception Handling - Templates - STL - Library Organization and Containers - Standard Containers - Overview of Standard Algorithms-Iterators and Allocators.

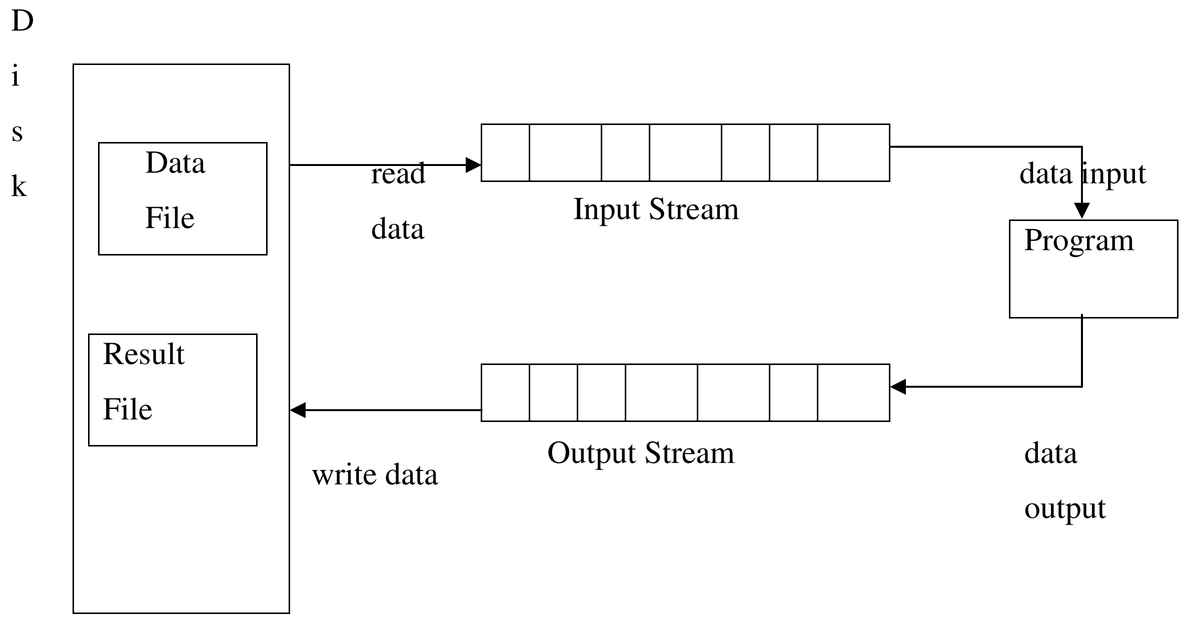
FILE HANDLING

File handling is an important part of all programs. Most of the applications need to save some data to the local disk and read data from the disk again. C++ File I/O classes simplify such file read/write operations for the programmer by providing easier to use classes.

The I/O system of C++ uses *file streams* as an interface between the program and the files during file operations.

- The stream that supplies data to the program is known as *input stream*
- The one that receives data from the program is known as *output stream*.

In other words, the input stream reads data from the file and the output stream writes data to the file. This is illustrated in figure



File Input and Output Streams

The input operation involves the creation of an *input stream* and linking the input file with the program. The input stream extracts (reads) the data from the file and supplies to the program. Similarly, the output operation involves establishing an *output stream* and making necessary links with the program and the output file. The output stream receives data from the program and stores or inserts (writes) the data into the file.

NOTE: File is a collection of related data stored in a particular area on the disk

DATA FILES

To accomplish the task of storing large amounts of data**data files** are used.There are two types of data files:

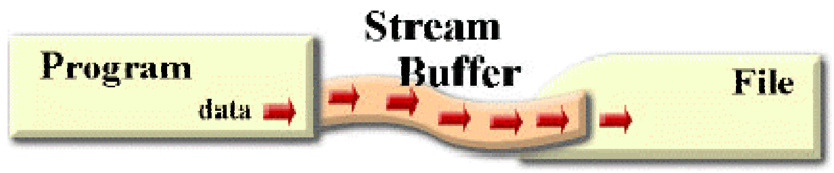
Sequential Access Files: These files must be accessed in the same order in which they were written. This process is analogous to audio cassette tapes where you must fast forward or rewind through the songs sequentially to get to a specific song. In order to access data from a sequential file, you must start at the beginning of the file and search through the entire file for the data that you want.

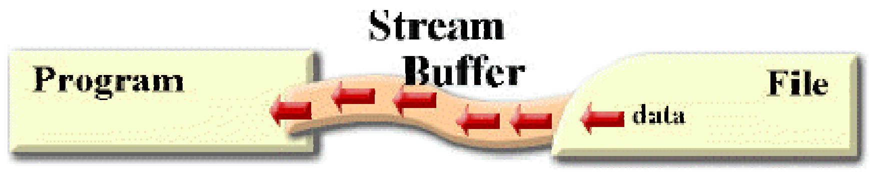
Random Access Files: These files are analogous to audio compact disks where you can easily access any song, regardless of the order in which the songs were recorded. Random access files allow instant access to any data in the file. Unfortunately, random access files often occupy more disk space than sequential access files.

<u>File Streams:</u>

C++ program views input (or output) as a **stream** of bytes. On input, a program extracts (>>) bytes from an input stream. On output, a program inserts (<<) bytes into the output stream. The stream acts as a mediator between the program and the stream's source or destination.

A buffer is a block of memory used as an intermediate, temporary storage area for the transfer of information between a program and a device





Creating a Sequential Access File

A file is a container for data. which needs to be opened and closed. Before you can put something into a file, or take something out, you must open the file (the drawer). When you are finished using the file, the file (the drawer) must be closed.

<u>File Stream Classes:</u>

C++ provides the following classes to perform input and output of characters to / from files:

- **ifstream** : Stream class to *read* from files
- **ofstream** : Stream class to *write* on files
- **fstream** : Stream class to both *read* and *write* from/to files

These file stream classes are designed exclusively to manage the disk files and their declaration exists in the header file *fstream.h,* therefore we must include this header file in any program that uses files.

NOTE: To use the classes, include the following statement in the program

```
# include< fstream.h >
```

These classes are derived directly or indirectly from the classes *istream*, and *ostream*. The actions performed by the stream classes related to file management are:

ifstream: The class ifstream supports **input** operations. It contains open( ) with default input mode and inherits get( ) , getline( ) , read( ) , seekg( ) , and tellg( ) functions from *istream*.

ofstream: The class ofstream supports **output** operations. It contains open( ) with default output mode and inherits put( ) , write( ) , seekp( ) , and tellp( ) functions from o*stream*

fstream: The class fstream supports simultaneous **input** and **output** operations. It contains open ( ) with default input mode and inherits all functions from *istream and ostream* classes through *iostream.*

Opening and Closing of Files :

Manipulation of a file involves the following steps:

- Name the File on the disk
- Open the File
- Process the File (Read / Write)
- Check for Errors while processing
- Close the File

File name:

The Filename is a string of characters, with which a file is logically identified by the user. The number of characters used for the file name depends on the Operating system. Normally a filename contains two parts, a *name* and an *extension*. The extension is optional.

In MS-DOS systems, the maximum size of a file name is eight characters and that of an extension is three characters. In UNIX based systems, the file name can be up to 31 characters and any number of extensions separated by a dot.

For example :

```
result.data
name.doc
salary              are some valid file names.
```

```
// Basic file operations

#include <iostream.h>
#include <fstream.h>

Void  main ()                                    [file example.txt]
  {
    ofstream myfile;                             Good Morning India
    myfile.open ("example.txt");
      myfile << "Good Morning India.\n";
    myfile.close();
  }
```

This code creates a file called **example.txt** and inserts a sentence into it in the same way we are used to do with cout, but using the file stream myfile instead.

<u>opening a file</u>

A file can be opened either in Read, Write or Append mode. For opening a file, we must first create a file stream and then link it to the filename. A file can be opened in two ways:

1. Using **Constructor function** of the class.
2. Using the member function **open ( )** of the class**.**

The first method is useful when we use only <u>*one file*</u> in the stream. The second method is used when we want to mange <u>*multiple files*</u> using one stream.

<u>Opening files using Constructors:</u>

We know that a constructor is used to initialize an object while it is being created. Here, the constructor can be utilized to initialize the file name to be used with the file stream object. The creation and assignment of file name to the file stream object involves the following steps:

1. Create a file stream object using the appropriate class. For example, o*fstream* can be used to create the output stream, *ifstream* for input stream and *fstream* can be used to create input and output stream.
2. Initialize the object with the desired filename

Opening File in Write Mode:

Example:

```
        #include <fstream.h>

         void main()
           {
                 ofstream myfile("example.txt");
                 myfile<< "Hello World!";
                 myfile.close( );
           }
```

This code creates a file called example.txt and inserts a sentence "Hello World" into it using the file stream myfile.

In the above example,

ofstream myfile ("example.txt");

1. **ofstream m**eans "output file stream". It creates an object for a file stream to **write in a file**.
2. myfile –the name of the object. The object name can be any valid C++ name.

3. **("example.txt");** - opens the file example.txt, which should be placed in the directory from where you execute the program. If such a file does not exist, it will be created for you.

We are familiar with the cout statement. For instance,

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

prints the message Hello World! on the screen. Whereas, the statement

```
myfile<< "Hello World!”
```

prints the message Hello World! into the file pointed by the file pointer myfile.

If we want to print variables instead of text, just write as myfile<<variable name.
For example the following statement

```
Myfile<<salary
```

writes the content of the variable salary to the output file.

In order to open a file with a stream object we use its member function
open():

Syntax: open (filename, mode);

Where filename is a null-terminated character sequence of type const char * (the same type that string literals have) representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

ios::in	**Open for input operations.**
ios::out	**Open for output operations.**
ios::binary	**Open in binary mode.**
ios::ate	**Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.**
ios::app	**All output operations are performed at the end of the file, appending the content to the current content of the file. This flag can only be used in streams open for output-only operations.**
ios::trunc	**If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.**

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function

<u>open():</u>

ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);

Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	**default mode parameter**
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectivelly assumed, even if a mode that does not include them is passed as second argument to the open() member function.

<u>Closing a file</u>

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function close(). This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

myfile.close();

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

```
#include<iostream.h>
#include<fstream.h>
Void main()
{
Ofstream outf("item");
Cout<<"enter the item name";
Char  name[30];
Cin>>name;
Outf<<name;
Cout<<"enter item cost";
float cost;
cin>>cost;
outf<<cost;
outf.close();
ifstream inf("item");
inf>>name;
inf>>cost;
cout<<name;
cout<<cost;
inf.close();
}
```

INPUT/OUTPUT OPERATIONS ON FILES:

- The functions, put(), and get(), are designed to manage a single character at a time.
- The other functions, read(), write(), are designed to manipulate blocks of character data.

put( ), and get( ) functions:

- The function get( ) is a member function of the file stream class fstream, and is used to read a single character from the file.

- The function put( ) is a member function of the output stream class fstream, and is used to write a single character to the output file.

```
#include <fstream.h>
void main()
  {
        char c, string[75];

        //open a file to write a string (character by character) into it

        fstream file(“student.txt”, ios::out);
        cout<<”Enter string:”;
        file.getline(string, 74);
        for(int i=0;string[i]; i++)
                file.put(string[i]);
        file.close( );

        // open a file to read a string (character by character) from it

        fstream file(“student.txt”, ios::in);
        cout<< <<”Output string”;
        while(!file.eof())
        {
                file.get(c);
                cout<<c;
        }
        file.close( );
 }
```

Run:

```
        Enter string: object oriented programming
        Output string: object oriented programming
```

In the above program a new member function called **eof()** that returns true in the case that the end of the file has been reached. We have created a while loop that finishes when indeed **file.eof()** becomes true(i.e. the end of the file has been reached).

Write( ) and read( ) functions:

- The functions **write ( )** and **read()**, unlike the function put() and get(), handle the data in binary form.
- This means that the values are stored in the disk file in the same format in which they are stored in the internal memory.
- For example an int value 2435 can be stored either as int or char formats.
- An int takes 2 bytes to store its value in the binary form, irrespective of its size but the character will take 4 bytes to store the same int value.
- The binary format is more accurate for storing the numbers as they are stored in the exact internal representation. There are no conversions while saving the data and therefore saving is much faster.
- The binary input and output functions takes the following syntax:

infile.read((char*)&V,sizeof(V));
outfile.write(char*)&V,sizeof(V));

these functions take two arguments.

- The first is the address of the variable V, it must be cast to type char*(ie. pointer to character type)
- The second is the length of that variable V in bytes.

```
#include <fstream.h>
void main()
  {
        int num1=2344;
        float num2=45.45;
        // open file in write binary mode, write int and close

        ofstream outfile("num.bin", ios::binary);
        outfile.write((char*)&num1,sizeof(num1));
        outfile.write((char*)&num2,sizeof(num2));
        outfile.close( );

        // open file in read binary mode, read int and close

        ifstream infile("num.bin", ios::binary);
        infile.read((char*)&num1,sizeof(num1));
        infile.read((char*)&num2,sizeof(num2));
        cout<<"number1:"<<num1<<"  number2:"<<num2;

        infile.close( );
  }
```

Run:

```
number1: 2344  number2: 45.45
```

Note: in main(), the statement

```
        outfile.write((char*)&num1,sizeof(num1));
```

writes the contents of the integer variable num1 to the disk file. The number of bytes to be written can be computed by sizeof(num1).

the statement

```
        infile.read((char*)&num1,sizeof(num1));
```

reads the sizeof(num1) gives the number of bytes from the file and stores in the memory location pointed by the 2nd parameter.

Exception Handling:-

Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing. Anomalies might include conditions such as division by zero, access to an array outside of its bounds, or running out of memory or disk space.

Exceptions are of two kinds, namely, synchronous exceptions and asynchronous exceptions. Errors such as “out-of-range” and “overflow” belong to the synchronous type exceptions. The errors that are caused by events beyond the control of the programs(such as keyboard interrupts) are called asynchronous exceptions. The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions.

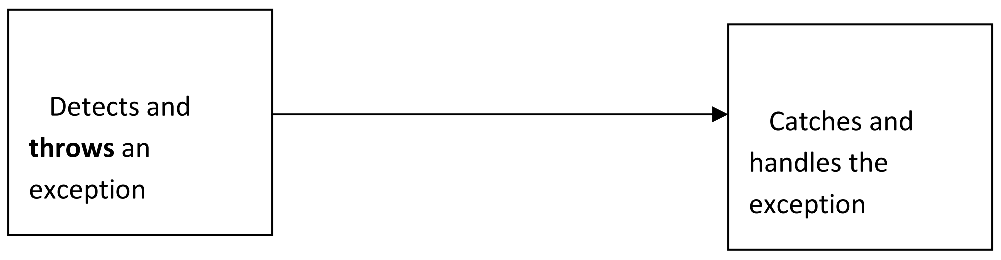
The purpose of the exception handling mechanism is to provide means to detect and report an “exceptional circumstance” so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks.

1. Find the problem(Hit the exception)
2. Inform that an error has occurred(throw the exception)
3. Receive the error information(catch the exception)
4. Take corrective actions(Handle the exception)

Exception handling mechanism is basically built upon three keywords, namely, try,throw, and catch. The keyword try is used to preface a block of statements which may generate exceptions. This block of statements is known as try block. When an exception is detected , it is thrown using a throw statement in the try block. A catch block defined by the keyword catch, catches the exception thrown by the throw statement in the try block, and handles it appropriately. The catch block that catches an exception must immediately follow the try block that throws the exception.

Exception Handling Model:

The exception handling mechanism uses three blocks try, throw, catch. The relationship of those three exception handling model shown below


Detects and **throws** an exception
Catches and handles the exception

try block   **catch** block

Exception

- The keyword try is used to preface a block of statements (surrounded by braces) which may generate exceptions. This block of statements known as try bock.
- When an exception is detected, it is thrown using a **throw** statement in the try block.
- A catch block defined by the keyword **catch** catches the exception thrown by the throw statement in the try block, and handle it appropriately.
- **General format:**

```
…………
…………
try
{
        ……….                 // block of statements which detects and
        throw   exception     // throws an exception
        ……….
        ……….
}
catch ( type arg)
{
        ………                    // block of statements that handles the exception
        ………
        ………
    }
……..
……..
```

- When the **try** block throws an exception, the program control leaves the try block and enters the catch statement of the catch block.
- Exceptions are objects or variables used to transmit information about a problem.
- If the type of object thrown matches the arg type in the catch statement, then catch block is executed for handling the exception.
- If they do not match, the program is aborted with the help of the **abort()** function which is invoked automatically.
- The catch block may have more than one catch statements, each corresponding to a particular type of exception.
- For example if the throw block is likely to throw two exception, the catch block will have two catch statements one for each type of exception. Each catch statement is called is exception handler.
- When no exception is detected and thrown, the control goes to the statement immediately after the catch block. That is catch block is skipped.
- Example: divide by zero

Program:-

```
#include<iostream.h>
void main()
{
        int a,b;
        cout <<"Enter values of a and b"<<endl;
        cin>>a;
        cin>>b;

        int x=a-b;

        try

        {

                if(x!=0)

                {

                        cout<<”Result(a/x)”<<a/x<<endl;

                }

                else// there is an exception

                {

                        throw(x);//throws int object

                }

        }

        catch(int i)

        {

                cout<<”Exception caught: x=”<<x<<endl;

        }

        cout<<”End”;

}
```

Output:-

```
        Enter values of a and b

        20 15

        Result(a/x)=4

        End
```

Output2:-

```
        Enter values of a and b
```

```
10 10
Exception caught: x=0
End
```

The program detects and catches a division-by-zero problem. The output of first run shows a successful execution. When no exception is thrown, the catch block is skipped and execution resumes with the first line after the catch. In the second run, the denominator x becomes zero and therefore a division-by-zero situation occurs. This exception is thrown using the object x. Since the exception object is an int type, the catch statement containing int type argument catches the exception and displays necessary message.

Invoking function that generates exceptions:

```
        #include<iostream.h>
        void divide(int x, int y, int z)
        {
                cout<<"We are inside the function";
                if(x-y!=0)
                {
                        int R = z/(x-y);
                        cout<<"Result = "<<R<<endl;
                }
                else
                {
                        throw(x-y)
                }
        }
void main()
{
        try
        {
                cout<<"We are inside the try block"<<endl;
                divide(10,20,30);
                divide(10,10,20);
        }
        catch(int i)
        {
                cout<<"caught the exception"<<endl;
        }
}
```

Output:-

```
        We are inside the try block
        We are inside the function
        Result=-3
        We are inside the function
        Caught the exception
```

Multiple Catch Statements:-

It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one catch statement with a try(much like the conditions in a switch statement)as shown below:

```
        try
        {
                //try block
        }
        catch(type1 arg)
        {
                //catch block1
        }
        catch(type2 arg)
        {
                //catch block2
        }
        --------
        --------
        catch(typeN arg)
        {
                //catch blockN
        }
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try. When no match is found, the program is terminated.

Program

```
#include<iostream.h>
void test(int x)
{
          try
          {
                    if(x == 1) throw x;
                    else
                              if(x == 0) throw 'x';
                    else
                              if(x == 1) throw 1.0;
                    cout<<"End of try-block"<<endl;
```

```
        }
        catch(char c)
        {
                cout<<”Caught a character”<<endl;
        }
        catch(int m)
        {
                cout<<”Caught an integer”<<endl;
        }
        catch(double d)
        {
                cout<<”Caught an double”<<endl;
        }

void main()
{
        cout<<”Testing Multiple catches”<<endl;
        cout<<”x==1”<<endl;
        test(1);
        cout<<”x==0”<<endl;
        test(0);
        cout<<”x== -1”<<endl;
        test(-1);
        cout<<”x== 2”<<endl;
        test(2);
}
```

Output:-

```
Testing Multiple catches

x==1

Caught an integer

End of try-block

x==0

Caught a character

End of try-block

x== -1

Caught an double

End of try-block

x== 2

Caught an integer

End of try-block
```

Catch All Exceptions:

In some situations, we may not be able to anticipate all possible types of exceptions and therefore may not be able to design independent catch handlers to catch them, IN such circumstances, we can force a catch statement to catch all exceptions instead of a certain type alone. This could be achieved by defining the catch statement using ellipses as follows.

```
catch(……)

        {

                //Statement for processing all exceptions

        }
```

Program

```
#include<iostream.h>

void test(int x)

{
        try
        {
                if(x == 0) throw x;
                if(x == -1) throw ‘x’;
                if(x == 1) throw 1.0;
                cout<<”End of try-block”<<endl;
        }
        catch(….)
        {
                cout<<”Caught an exception”<<endl;
        }
void main()
{
        cout<<”Testing Generic catch”<<endl;
        test(-1);
        test(0);
        test(1);
}
```

Output:-

```
Testing Generic catch
Caught an exception
Caught an exception
Caught an exception
```

Rethrowing an Exception:-

A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any arguments as shown below.

```
Throw;
```

This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

Program:-

```
    #include<iostream.h>
    void divide(double x,double y)
    {
        cout<<"Inside  function"<<endl;
        try
        {
            if(y == 0.0)
                throw y;
            else
                cout<<"Division ="<<x/y<<endl;
        }
        catch(double e)
        {
            cout<<"Caught double inside function"<<endl;
            throw;
        }
        cout<<"End of function"<<endl;
    }
void main()
{
    cout<<"Inside main"<<endl;
    try
    {
        divide(10.5,2.0);
```

```
                divide(20.0,0.0);
        }
        catch(double i)
        {
                cout<<"caught double inside main"<<endl;
        }
        cout<<"End of main"<<endl;
}
```

Output:-

```
        Inside main
        Inside function
        Division = 5.25
        End of function
        Inside function
        Caught double inside function
        Caught double inside main
        End of main
```

When an exception is rethrown, it will not be caught by the same catch statement or any other catch in that group. Rather, it will be caught by an appropriate catch in the outer try/catch sequence only. A catch handler itself may detect and throw an exception. Here again, the exception thrown will not be caught by any catch statements in that group. It will be passed on to the next outer try/catch sequence for processing.

Templates

Templates in C++ programming allows function or class to work on more than one data type at once without writing different codes for different data types. Templates are often used in larger programs for the purpose of code reusability and flexibility of program. The concept of templetes can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function templates work in similar manner as function but with one key difference. A single function template can work on different types at once but, different functions are needed to perform identical task on different data types. If you need to perform identical operations on two or more types of data then, you can use function overloading. But better approach would be to use function templates because you can perform this task by writing less code and code is easier to maintain.

A function template starts with keyword **template** followed by template parameter/s inside **< >** which is followed by function declaration.

```
template <class T>
 return type some_function(T arg)
{
   .... ... ....
}
```

In above code, *T* is a template argument and **class** is a keyword. You can use keyword **typename** instead of class in above example. When, an argument is passed to some_function( ), compiler generates new version of some_function() to work on argument of that type.

Example of Function Template

```
template <typename T>
void Swap(T &n1, T &n2)
{
```

```
          T temp;
          temp = n1;
          n1 = n2;
          n2 = temp;
}

void main()
{
          int i1=1, i2=2;
          float f1=1.1, f2=2.2;
          char c1='a', c2='b';
          cout<<"Before passing data to function template.\n";
          cout<<"i1 = "<<i1<<"\ni2="<<i2;
          cout<<"\nf1 = "<<f1<<"\nf2="<<f2;
          cout<<"\nc1 = "<<c1<<"\nc2="<<c2;

          Swap(i1, i2);
          Swap(f1, f2);
          Swap(c1, c2);

     cout<<"\n\nAfter passing data to function template.\n";
          cout<<"i1 = "<<i1<<"\ni2="<<i2;
          cout<<"\nf1 = "<<f1<<"\nf2="<<f2;
          cout<<"\nc1 = "<<c1<<"\nc2="<<c2;
}
```

Output:

```
Before passing data to function template.
i1 = 1
i2 = 2
```

```
f1 = 1.1
f2 = 2.2
c1 = a
c2 = b

After passing data to function template.
i1 = 2
i2 = 1
f1 = 2.2
f2 = 1.1
c1 = b
c2 = a
```

Class Template

Templates are a feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

```
template <class T>
class class-name
{

.....
//class member specification

};
```

The class template is very similar to an ordinary class definition except the prefix **template <class T>** and use of type T.The prefix tells the compiler that we are going to declare a template and use T as a type name in the declaration.T may be substituted by any data type including the user defined types.

Class Templates with Multiple Parameter:

```
Template<class T1 ,class T2>
Class classname
{
............
............
};
```

Example:

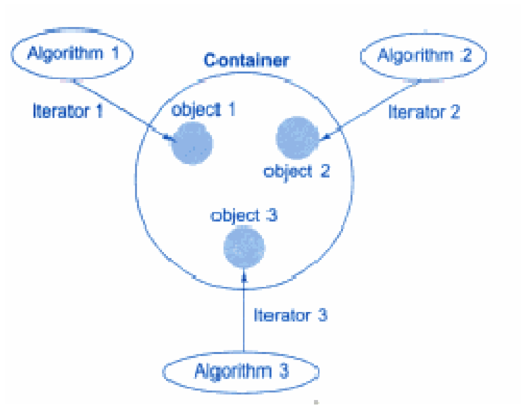
```
#include<iostream.h>
Template<class T1,class T2>
Class Test
{
T1 a;
T2 b;
Public:
Test(T1 x ,T2 y)
{
a=x;
b=y;
}
Void show()
{
Cout<<a<<b;
}
};
Void main()
{
Test<float, int >test1(1.2,123);
Test<int,char>test2(100,’w’);
test1.show();
test2.show();
}
```

STANDARD TEMPLATE LIBRARY

Most computer programs exist to process data. The data may represent a wide variety of realworld information: personnel records, inventories, text documents, the results of scientific experiments, and so on. Whatever it represents, data is stored in memory and manipulated in similar ways. University computer science programs typically include a course called "DataStructures and Algorithms." The term data structures refers to the ways data is stored in memory,and algorithms refers to how it is manipulated.

C++ classes provide an excellent mechanism for creating a library of data structures. In the past, compiler vendors and many third-party developers offered libraries of container classes to handle the storage and processing of data. However, Standard C++ includes its own builtin container class library. It's called the Standard Template Library (STL), and was developed by Alexander Stepanov and Meng Lee of Hewlett Packard. The STL is part of the Standard C++ class library, and can be used as a standard approach to storing and processing data.

Components of STL:



The STL contains several kinds of entities. The three most important are containers, algorithms,and iterators.

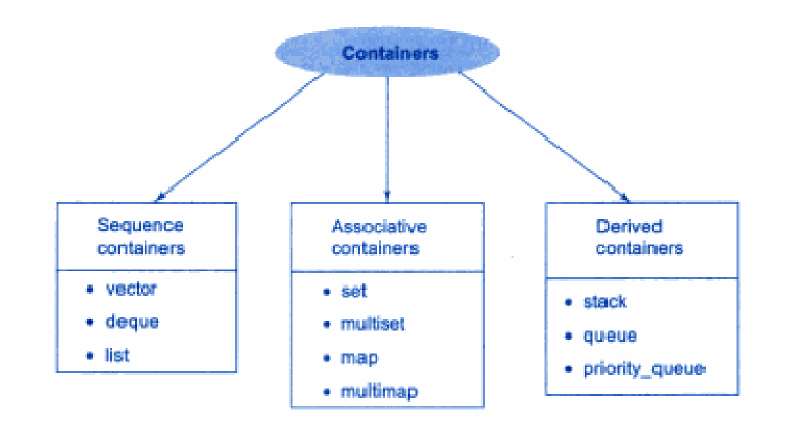
A ***container*** is a way that stored data is organized in memory. There are two kinds of containers: stacks and linked lists. Another container, the array, is socommon that it's built into C++ (and most other computer languages). However, there are many other kinds of containers, and the STL includes the most useful. The STL containers are implemented by template classes, so they can be easily customized to hold different kinds ofdata.

Algorithms in the STL are procedures that are applied to containers to process their data in variousways. For example, there are algorithms to sort, copy, search, and merge data. Algorithms are represented by template functions. These functions are not member functions of the container classes. Rather, they are standalone functions.

Iterators are a generalization of the concept of pointers: they point to elements in a container.we can increment an iterator, as you can a pointer, so it points in turn to each element in a container. Iterators are a key part of the STL because they connect algorithms with containers.

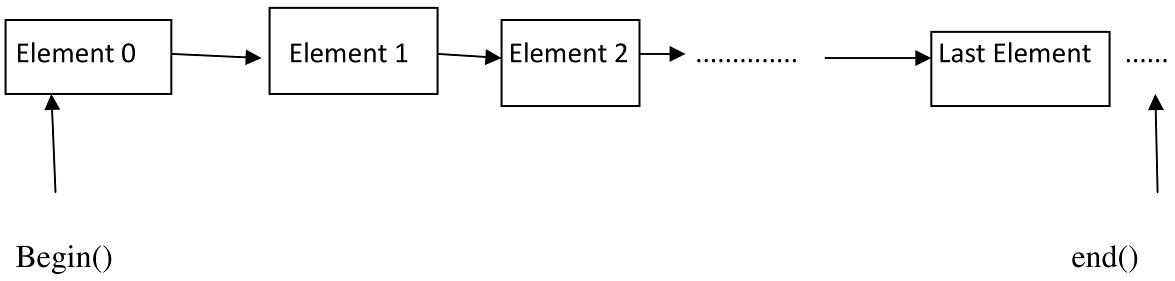
Containers

The STL contains sequence containers and associative containers. The Containers are objects that store data.



Sequence Containers:

A sequence container stores a set of elements in linear sequence.Each element is related to other element by its position along the line.They all expand themselves to allow insertion of element and all of them support a number of operations on them.



Sequence containers maintain the ordering of inserted elements that you specify.

A **vector** container behaves like an array, but can automatically grow as required. It is random access and contiguously stored, and length is highly flexible. **vector** is the preferred sequence container for most applications. When in doubt as to what kind of sequence container to use, start by using a vector!

.

An **array** container has some of the strengths of **vector**, but the length is not as flexible.

A **deque** (double-ended queue) container allows for fast insertions and deletions at the beginning and end of the container. It shares the random-access and flexible-length advantages of **vector**, but is not contiguous.

.

A **list** container is a doubly linked list that enables bidirectional access, fast insertions, and fast deletions anywhere in the container, but you cannot randomly access an element in the container.

A **forward_list** container is a singly linked list—the forward-access version of **list**.

Associative Containers

In associative containers are designed to support direct acces to element using keys.They are not sequential. The associative containers can be grouped into two subsets: maps and sets.

A **map**, sometimes referred to as a dictionary, consists of a key/value pair. The key is used to order the sequence, and the value is associated with that key. For example, a **map** might contain keys that represent every unique word in a text and corresponding values that represent the number of times that each word appears in the text. The unordered version of **map** is **unordered_map**.

A **set** is just an ascending container of unique elements—the value is also the key. The unordered version of **set** is **unordered_set**.

Both **map** and **set** only allow one instance of a key or element to be inserted into the container. If multiple instances of elements are required, use **multimap** or **multiset**. The unordered versions are **unordered_multimap** and **unordered_multiset**.

Ordered maps and sets support bi-directional iterators, and their unordered counterparts support forward iterators.

Derived containers:

The STL provides three Derived containers namely,stack,queue and priority_queue. These are also knows as container adaptors.

stack,queue and priority_queue can be created from different sequence containers.The Derived containers do not support iterators and therefore we cannot use them for data mainpulation.However, they support two member functions pop() and push() for deleting and inserting.

Overview of Standard Algorithms

Algorithms are functions that can be used generally across a variety of containers for processing their content.Although each container provides functions for its basics operations,STL provides more than sixty standared algorithms to support more extended or complex operations.Standard algorithms also permit us to work with two different types of continers at the same time.STL algorithm are not member function or frienfs of containers.They are standalone template functions.
To have access to STL algorithm **<algorithm>** must be include in the program.

STL algorithm,based on the nature of the operations they perform,they are categorised under

- Retrieve or non-mutating algorithm
- mutating algorithm
- Sorting algorithm
- Set algorithm
- Relational algorithm

Non modifying operations:

for_each	Do specified operation for each element in a sequence
find	Find the first occurence of a specified value in a sequence
find_if	Find the first match of a predicate in a sequence

find_first_of	Find the first occurence of a value from one sequence in another
adjacent_find	Find the first occurence of an adjacent pair of values
count	Count occurences of a value in a sequence
count_if	Count matches of a predicate in a sequence
accumulate	Accumulate (i.e., obtain the sum of) the elements of a sequence
equal	Compare two ranges
max_element	Find the highest element in a sequence
min_element	Find the lowest element in a sequence

Modifying operations:

transform	Apply an operation to each element in an input sequence and store the result in an output sequence (possibly the same input sequence)
copy	Copy a sequence
replace	Replace elements in a sequence with a specified value
replace_if	Replace elements matching a predicate
remove	Remove elements with a specified value
remove_if	Remove elements matching a predicate
reverse	Reverses a sequence
random_shuffle	Randomly reorganize elements using a uniform distribution
fill	Fill a sequence with a given value
generate	Fill a sequence with the result of a given operation

Sorting:

sort	Sort elements
stable_sort	Sort maintaining the order of equal elements
nth_element	Put nth element in its place
binary_search	Find a value in a sequence, performing binary search

Numeric operations

Defined in header <numeric>

accumulate	sums up a range of elements (function template)
inner_product	computes the inner product of two ranges of elements (function template)
adjacent_difference	computes the differences between adjacent elements in a range (function template)
partial_sum	computes the partial sum of a range of elements (function template)

Set operations (on sorted ranges)

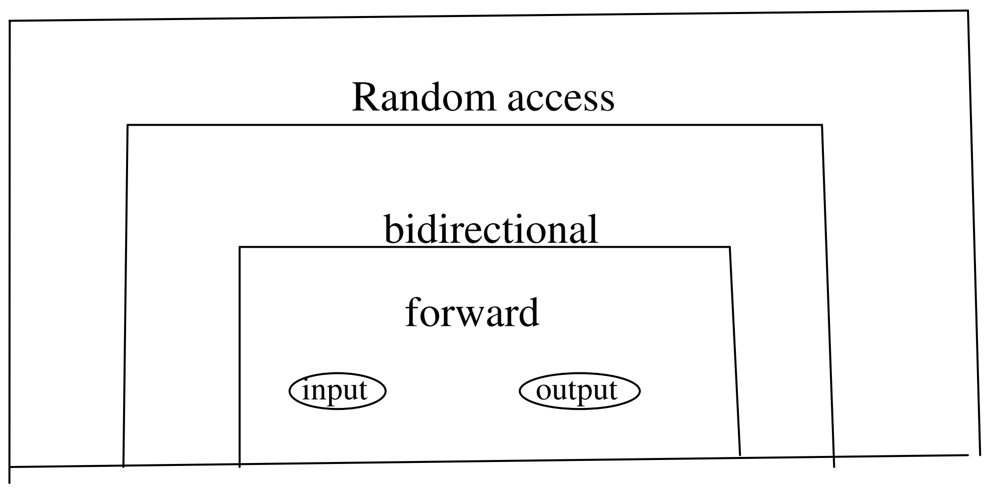
Defined in header <algorithm>

includes	returns true if one set is a subset of another (function template)
set_difference	computes the difference between two sets (function template)
set_intersection	computes the intersection of two sets (function template)
set_symmetric_difference	computes the symmetric difference between two sets (function template)
set_union	computes the union of two sets (function template)

Iterator:

Iterators behaves like pointer and are used to acces container elements.They are often used to traverse from one element to another,a process known as iterating through the container.

Different types of iterator must used with thwe different types of containers
Each type of iterator is used for performing certain functionsthe venn diagram of iterator is as follows



The **input** and **output** iterator supports the least functions.They can be used only to traverse in a container.

The **forward** iterator supports all operations of input and output iterator and also retains its position in the container.

A **bidirectiona**l iterator support all forward iterator operations,provides the ability to move in the backward direction in the container.

A **random access** iterator combines the functionality of a bidirectional iterator with an ability to jump to an arbitrary location.

Iterator	Element access	Read	Write	Increment operation	comparison
Input	->	v=*p		++	==,!+
Output			*p=v	++	
Forward	->	v=*p	*p=v	++	==,!=
Bidirectional	->	v=*p	*p=v	++.--	==,!=
Random access	->,[]	v=*p	*p=v	++,--,+,- +=,-+	==.!=,<,>,<=,>=

Allocators:

STL provides allocator that does all the memory management of the container classes. The concept of allocators was originally introduced to provide an abstraction for different memory models to handle the problem of having different pointer types on certain 16-bit operating systems (such as near, far, and so forth). However, this approach failed. Nowadays, allocators serve as an abstraction to translate the need to use memory into a raw call for memory.

Allocators separate the implementation of containers, which need to allocate memory dynamically, from the details of the underlying physical memory management. Thus, different memory models such as shared memory, garbage collections, and so forth to your containers without any hassle because allocators provide a common interface.

Allocators are integrated into the container classes by

```
vector<T, Alloc>
```

'T' represents the vector's value type—in other words, the type of object that is stored in the vector. 'Alloc' represents the vector's allocator, the method for the internal memory management.

The internal implementation of the allocator is completely irrelevant to the vector itself. It is simply relying on the standardized public interface every allocator has to provide. The vector does not need to care any longer whether it would need to call 'malloc', 'new', and so on to allocate some memory; it simply calls a standardized function of the allocator object named 'allocate()' that will simply return a pointer to the newly allocated memory.