UNIT III

Exception handling - Streams and Formatted I/O – File handling – Namespaces - String Objects - Standard Template Library.

EXCEPTION HANDLING

- Exceptions are **run-time anomalies or unusual conditions** that are encountered while executing a program.
- For example, the run-time anomalies include:
  - Division by zero.
  - Access to an array outside its bounds
  - Running out of memory and disk space.
- Exceptions are of two kinds:
  1. Asynchronous exceptions
     - These types of exceptions are caused by events beyond the control of the program. For example: keyboard interrupts, hardware failure etc.,
  2. Synchronous exceptions
     - These types of exceptions are caused due to the abnormal conditions occurring in a program. For example: out-of-range index, division by zero etc.,

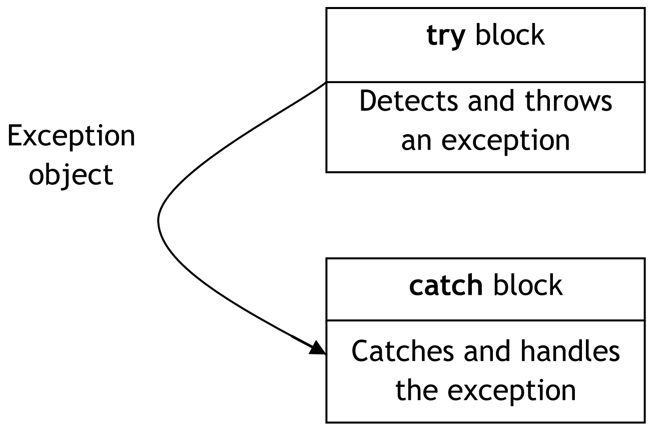
Basics of Exception Handling:

The exception handling mechanism 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:

- The C++ Exception Handling mechanism operates on three keywords.
  1. Try
  2. Throw
  3. Catch


try block
Detects and throws an exception
Exception object
catch block
Catches and handles the exception

Try:

- The try block contains a block of statements which may generate exceptions.

Throw:

- When an exception is detected by the try block, it is thrown to the catch block using the throw statement using the one of the following syntax:

  throw exception;

  throw; *// used for rethrowing an exception.*

Catch:

- The catch block contains the code for handling the exceptions. The catch block has the following syntax:

```
catch (datatype  arg)
{
//Statements for managing exceptions
}
```

- The datatype of argument is mandatory and argument name is optional.
- When the try block throws an exception, the program control leaves the try block and enters the catch block.

The general form of these blocks is stated below:

```
try
{
…..
throw exception;
…..
}
catch( type arg)
{
……
}
```

EXAMPLE 1: THROW OUTSIDE THE TRY BLOCK

```
#include<iostream>
using namespace std;
void divide(int x, int y, int z)
{
        if ((x-y)!=0)
        {
                int res=z/(x-y);
                cout<<"Result ="<<res<<endl;
        }
        else
        {
                throw (x-y);                    // throwing integer
        }
}
int main()
{
        try
        {
                divide(10,20,30);               //Invokes divide() function
                divide(10,10,20);               //Invokes divide() function
        }
        catch( int i)
        {
                cout<<"Caught an exception"<<endl;
        }
return 0;
}
```

OUTPUT:

```
Result= -3

Caught an exception
```

MUTLIPLE CATCH STATEMENTS:

A program segment may raise more than one exception during its execution. In such cases, multiple catch statements should be used as shown below.

Syntax for multiple catch statements:

```
try
{
}
catch (type1 arg)
{
}
catch(type2 arg)
{
}
……
catch( typeN arg)
{
}
```

- When an exception is thrown, the appropriate exception handlers (catch blocks) are searched for an appropriate match.
- The first handler (catch block) that yields the exact match will be executed.
- When no match is found the program is terminated.

EXAMPLE FOR MULTIPLE CATCH STATEMENTS:

```
#include<iostream>
using namespace std;
void test(int  x)
{
        try
        {
                if(x==1)
                        throw x;                    // throws an integer
                else if( x==0)
                        throw 'x';                  // throws a character
                else if(x== -1)
                        throw 1.0;                     // throws a float value
        }
        catch( char c)                                  // Catch block1
        {
                cout<<"Caught a character"<<endl;
        }
        catch( int m)                                   // Catch block2
        {
                cout<<"Caught an integer"<<endl;
        }
        catch (float f)                                 // Catch block3
        {
                cout<<" Caught a float value"<<endl;
        }
}
int main()
{
        cout<<"x==1"<<endl;
        test(1);                                                // Invokes test() function
        cout<<"x==0"<<endl;;
        test(0);                                                // Invokes test() function
        cout<<"x==-1"<<endl;
        test(-1);                                               // Invokes test() function
return 0;
}
```

OUTPUT:

```
x==1
Caught an integer
x==0
Caught a character
x==-1
Caught a float value
```

Catch all exceptions:

The catch block with the following syntax catches all the exceptions instead of a certain type alone.

```
catch(…)
{
// Statements for processing all the exceptions.
}
```

EXAMPLE: CATCH ALL EXCEPTIONS

```
#include<iostream>
using namespace std;
void test(int  x)
{
        try
        {
                if(x==1)
                        throw x;                // throwing integer
                else if( x==0)
                        throw 'x';              // throwing character
                else if(x== -1)
                        throw 1.0;              // throwing  float
        }
        catch( …)                               // Catch block to catch all exceptions
        {
                cout<<"Caught an exception"<<endl;
        }
}
int main()
{
        cout<<"x==1"<<endl;
        test(1);                                // Invokes test() function
        cout<<"x==0"<<endl;;
        test(0);                                // Invokes test() function
        cout<<"x==-1"<<endl;
        test(-1);                               // Invokes test() function
return 0;
}
```

OUTPUT:

```
x==1
Caught an exception
x==0
Caught an exception
x== -1
Caught an exception
```

RETHROWING AN EXCEPTION

The catch block may decide to rethrow the exception caught without processing it. In such cases the throw statement takes the following form:

Syntax for rethrow:

```
        throw;                  //Throw without arguments
```

EXAMPLE: RETHROWING AN EXCEPTION

```
#include<iostream>
using namespace std;
void divide(int x, int y)
{
    try
    {
    if(y==0)
        throw y;                // Throwing integer
    else
        cout<<"Result="<<x/y<<endl;
    }
    catch(int)
    {
        cout<<"Caught integer inside divide()"<<endl;
        throw;                          // Rethrowing exception
    }
}
int main()
{
    try
    {
        divide(10,5);           // Invokes divide() function
        divide(20,0);           // Invokes divide() function
    }
    catch(int)
    {
        cout<<"Caught integer inside main()"<<endl;
    }
return 0;
}
```

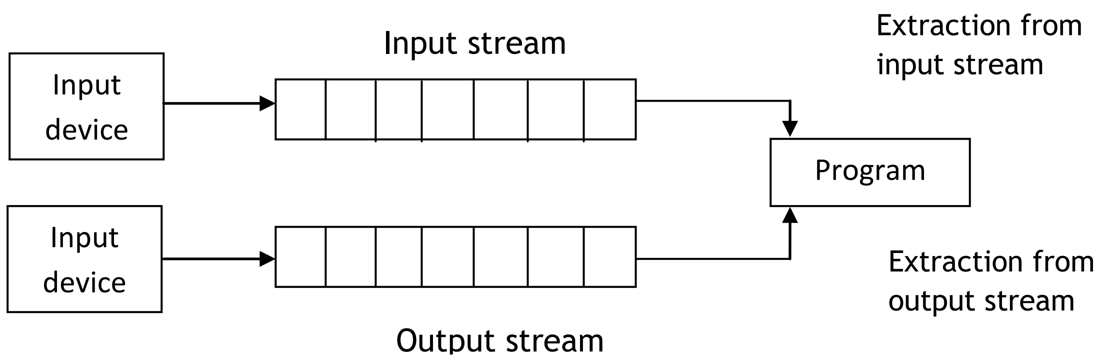
OUTPUT:

```
Result=2
Caught integer inside main
```

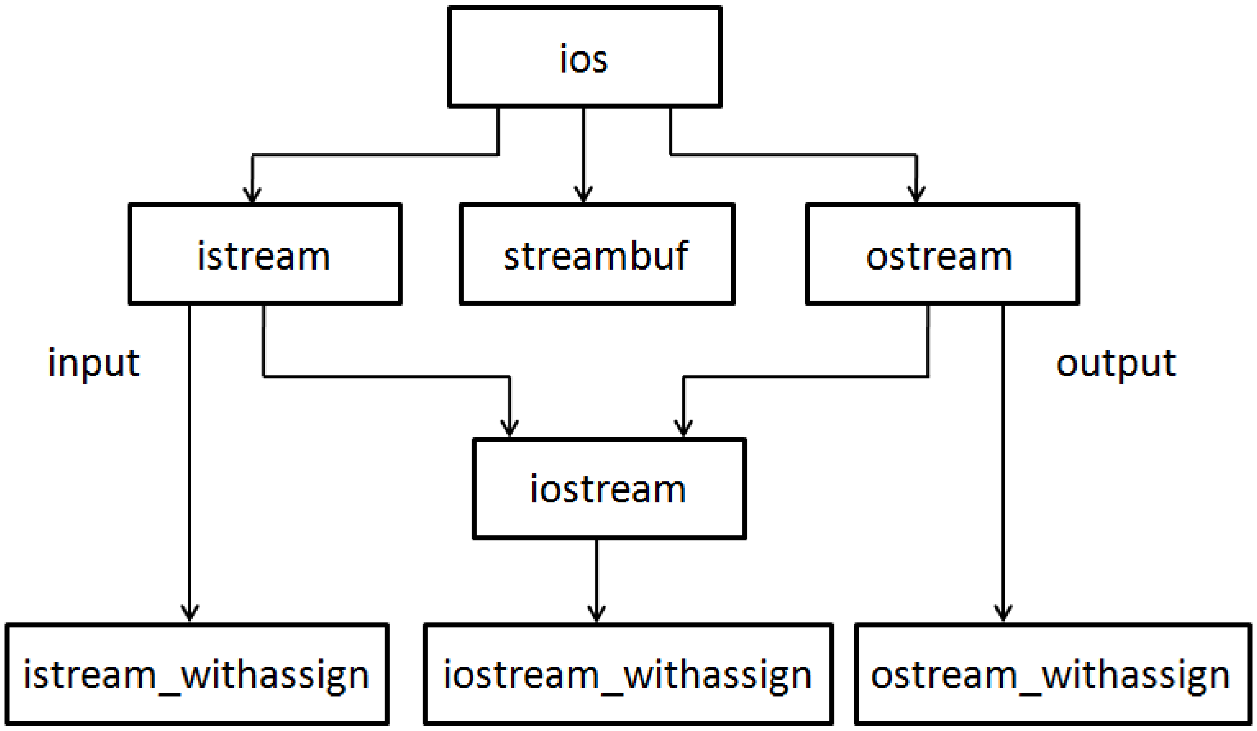
===

STREAMS

- A stream is a sequence of bytes.
- It acts either as a source from which input data can be obtained or as a destination to which the output data can be sent.
- The source stream is called as input stream and the destination stream is called as output stream.



C++ STREAM CLASSES



Stream classes for console operations:

1. ios
   - Contains basic facilities that are used by all other input and output classes.
   - Also contains a pointer to a streambuf object.
   - Declares functions for handling formatted input and output operations.

2. istream (input stream)
    - Inherits the properties of ios
    - Declares input functions such as get(), getline() and read().
    - Contains overloaded extraction operator >>.
3. ostream (output stream)
    - Inherits the properties of ios
    - Declares input functions such as put()and write().
    - Contains overloaded insertion operator <<.
4. iostream (input/output stream)
    - Inherits the properties of ios istream and ostream classes through multiple inheritance and thus contains the input and output functions.
5. streambuf
    - Provides an interface to physical devices through buffers.
    - Acts as a base for filebuf class used ios files.

FORMATTED I/O OPERATIONS

C++ supports a number of features that could be used for formatting the output. These features include:

1. ios class functions and flags
2. Manipulators
3. User-defined manipulators.

i). ios CLASS FUNCTIONS AND FLAGS:

The ios class contains a large number of member functions that would help to format the output. Since these functions are member functions of ostream class, it can be accessed only by using the object of ostream class (cout).

1. **width()**
    - Specifies the required field size for displaying an output value.

    Syntax:

    cout.width(w);

    Where, w is the field width.

2. **precision()**
    - Specifies the number of digits to be displayed after the decimal point of a floating point number.

    Syntax:
    cout.precision(d);

    where, d is the number of digits to the right of the decimal point.

3. **fill()**
   - Specifies the character that is used to fill the unused portion of a field.

   Syntax:

   - **cout.fill('ch');**

   where, ch represents the character which is used for filling the unused positions of the field.

4. **setf()**
   - setf() is a member function of ios class, to format the output by setting the formatting flags.

   Syntax:

   - **cout.setf(flag, bitfield);**

5. **unsetf()**
   - unsetf() is a member function of ios class, to clear the format flags

   Syntax:

   - **cout.unsetf(flag, bitfield);**

FORMATTING FLAGS AND BIT FIELDS

Flags and bitfields for format function:

Format Required	**Flag**	**Bitfield**
Left-justified output	ios::left	ios::adjustfield
Right-justified output	ios::right	ios::adjustfield
Padding after sign	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	Ios::floatfield
Decimal Base	ios::dec	ios::basefield
Octal Base	ios::oct	ios::basefield
Hexadecimal Base	ios::hex	ios::basefield

Flags that have no bit fields:

Flag	**Meaning**
ios::showbase	Use base indicator on output
ios::showpos	Print + before positive numbers
ios::showpoint	Show trailing decimal point and zeroes
ios::uppercase	Use uppercase letters for hex output
ios::skipus	Skip white space on input
ios::unitbuf	Flush all streams after insertion
ios::stdio	Flush stdout and stderr after insertion

EXAMPLE PROGRAM USING FORMAT FUNCTIONS

```
#include<iostream.h>

#include<conio.h>

void main()

{

        cout.fill('#');

        cout.precision(3);

        cout.setf(ios::internal, ios::adjustfield);

        cout.setf(ios::scientific, ios::floatfield);

        cout.width(15);

        cout<<-12.34567<<endl;

        getch();

}
```

OUTPUT:

```
- #####1.236E+01
```

ii). MANIPULATORS

- The manipulators are format functions that can be used in conjunction with the << and >> operators.
- The header file **<iomanip.h>** provides a set of functions called as manipulators to manipulate the output format.

ios member functions	**Equivalent manipulators**
width()	setw(int w)
precision()	setprecision(int d)
Fill	setfill(char c)
setf()	setiosflags(long f)
unsetf()	Resetiosflags(long f)

EXAMPLE FOR FORMATTING OUTPUT WITH MANIPULATORS:

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
void main()
{
double term, sum=0;
cout.setf(ios::showpoint);
cout<<setw(5)<<"N";
cout<<setw(15)<<"Inverse";
cout<<setw(15)<<"Sum of terms";
for(int n=1; n<=5; n++)
{
        term=1.0/float(n);
        sum=sum+term;
        cout<<setw(5)<<n;
        cout<<setw(15)<<setprecision(4)<<setiosflags(ios::scientific)<<term;
        cout<<resetiosflags(ios::scientific)<<sum<<endl;
}
getch();
}
```

OUTPUT:

```
N               Inverse             Sum of terms
1               1.0000e+000         1.0000
2               5.0000e-001         1.5000
3               3.3333e-001         1.8333
4               2.5000e-001         2.0833
5               2.0000e-001         2.2833
```

iii). USER-DEFINED MANIPULATORS:

For some special purposes,new manipulators can be created by the users. The general form for creating a manipulator is:

```
        ostream & manipulator(ostream &output)
        {
        ….(code)
        return output;
        }
```

here, the manipulator is the name of the manipulator being created.

EXAMPLE FOR USER-DEFINED MANIPULATORS:

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
ostream &currency(ostream &output)          // user-defined manipulator 1
{
        output<<"Rs.";
        return output;
}
ostream &form(ostream & output)             // user-defined manipulator 2
{
        output.setf(ios::showpoint);
        output<<setfill(' # ')<<setprecision(2)<<setw(15);
        return output;
}
void main()
{
        cout<<currency<<form<<7865.5;
        getch();
}
```

OUTPUT:

```
Rs.#####7865.50
```

UNFORMATTED I/O OPERATIONS

get() and put()

- get() and put() are used to handle single character input/output operations

Syntax:

```
cin.get(c);

cout.put(c);
```

Example:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        int count=0;
        char c;
        cout<< "Enter the text\n";
        cin.get(c);
        cout<< "\nThe Entered Text is\n";
        while(c!='\n')
        {
        cout.put(c);
        count++;
        cin.get(c);
        }
        cout<<"\n\nNumber of Characters in Given Text = "<<count<<"\n";
        getch();
}
```

OUTPUT:

```
Enter the text
WELCOME TO EEE
The Entered Text is
WELCOME TO EEE
Number of Characters in Given Text = 14
```

getline() & write()

- getline() and write() are used to perform input/output operations on a line of text.

Syntax:

```
cin.getline(line,size);

cout.write(line,size);
```

Example:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

void main ()
{
        clrscr();
        int size=20;
        char *string1;
        char *string2;
        cout<<"\nEnter the string 1:";
        cin.getline(string1,size);
        cout<<"\nEnter the string 2:";
        cin.getline(string2,size);

        int m=strlen(string1);
        int n=strlen(string2);
        for(int i=1;i<n;i++)
        {
                cout.write(string2,i);
                cout<<"\n";
        }
        for(i=n;i>0;i--)
        {
                cout.write(string2,i);
                cout<<"\n";
        }
        cout.write(string1,m).write(string2,n);          //Concatenating strings
        cout<<"\n";
        getch();
}
```

OUTPUT:

```
Enter the string 1:Hello
Enter the string 2: EEE
E
EE
EEE
EE
E
HelloEEE
```

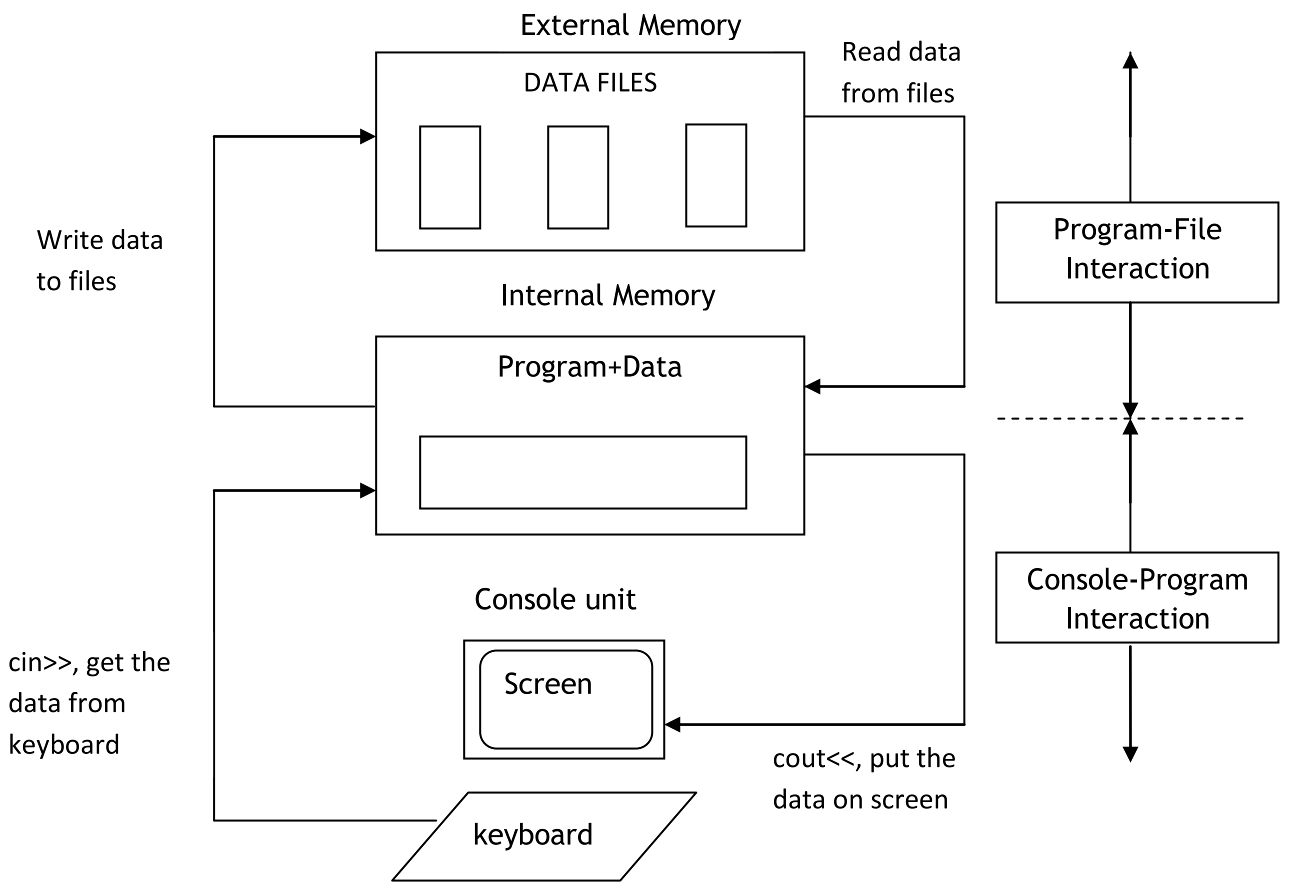
==

FILE HANDLING

Definition:

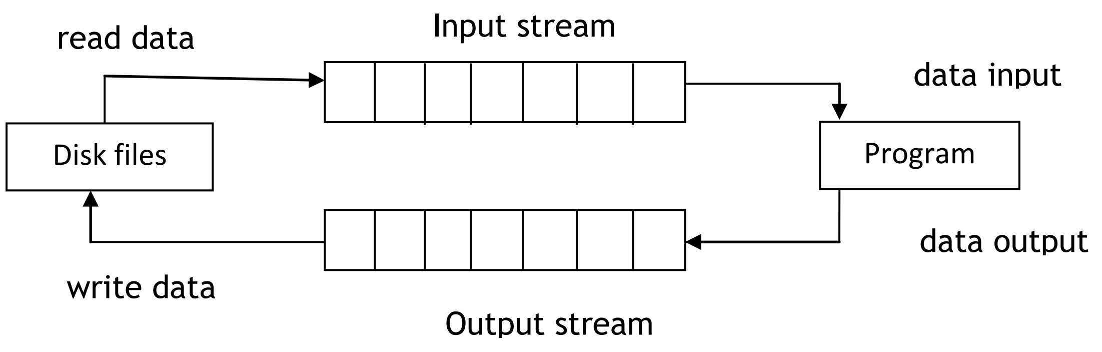
- A file is a collection of related data stored in a particular area on the disk.
- Large volumes of data are stored in devices such as CD, DVD, floppy disk or hard disk using the concept of files.
- The data of a file is stored in either readable form called as text file or data can be stored as binary code called as binary file.

PROGRAM-CONSOLE-FILE INTERACTION

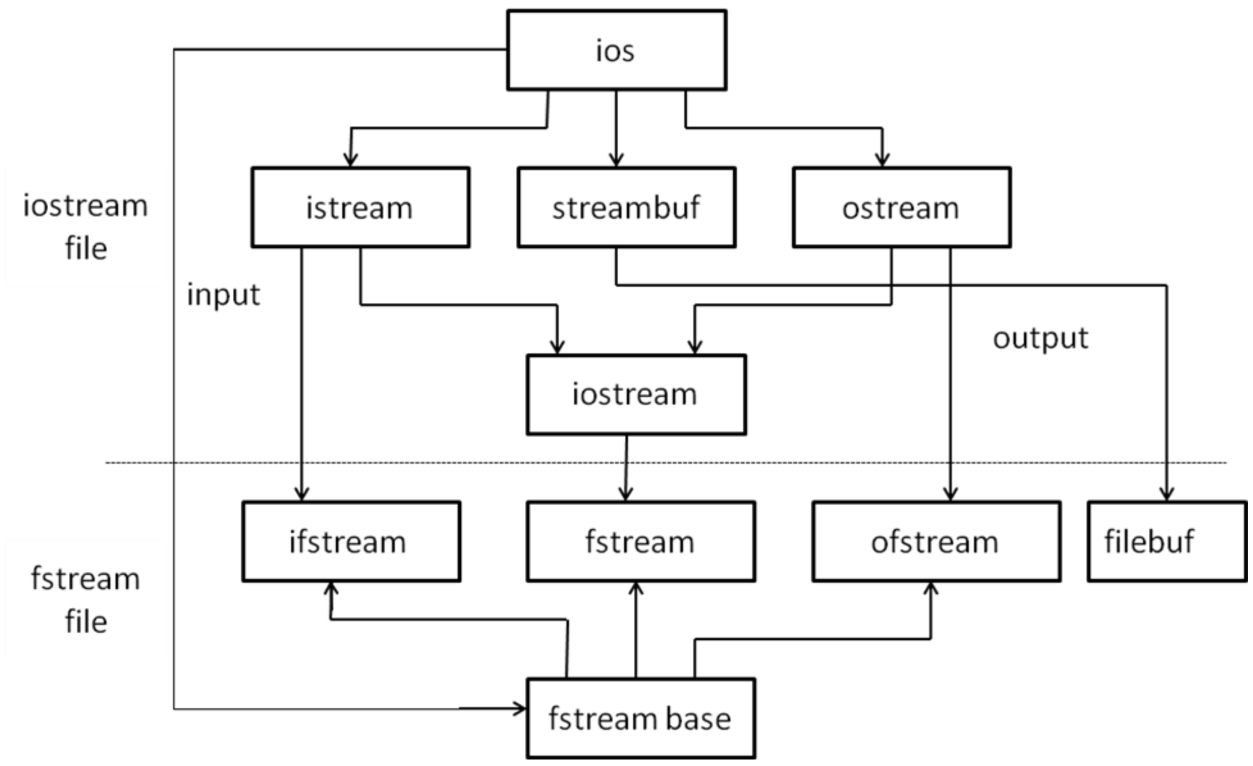


FILE INPUT AND OUTPUT STREAMS

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



FILE STREAM CLASSES



DETAILS OF FILE STREAM CLASSES

1. **filebuf**
    - It sets the file buffers to read and write.
    - Contains close() and open() as members
2. **fstreambase**
    - Provides operations common to the file streams.
    - Serves as a base for fstream, ifstream and ofstream class.
    - Contains open() and close() functions.
3. **ifstream**
    - Provides input operations.
    - Contains open() with default input mode.
    - Inherits the functions **get(), getline(),read(), seekg() and tellg()** from **istream** class.
4. **ofstream**
    - Provides output operations.
    - Contains open() with default output mode.
    - Inherits the functions **put(), write(),seekp(), tellp()** from **ostream** class.
5. **fstream**
    - Provides support for simultaneous input and output operations.
    - Contains open() with default input mode.
    - Inherits all the functions from istream and ostream classes through iostream class.

OPENING FILES

A file can be opened in two ways:

1. using the constructor function of the class
2. using the member function open() of the class.

OPENING FILES USING CONSTRUCTOR

Syntax:

```
filestream_class  stream_object(“filename”);
```

Eg:

```
ifstream inf(“result”);          // opens a file in input mode.

ofstream outf(“result);          // opens a file in output mode.
```

OPENING FILES USING OPEN()

Syntax:

```
filestream_class stream_object;

stream_object.open(“Filename”);
```

Eg:

```
ifstream inf;                    // opens a file in input mode

inf.open(“result”);


ofstream outf;                   // opens a file in output mode.

outf.open(“result”);
```

CLOSING FILES

Syntax:

```
streamobject . close()
```

Eg:

```
inf.close();                     //disconnects the file from input mode

outf.close()                     //disconnects the file from output mode
```

INPUT/OUTPUT OPERATIONS ON TEXT FILES:

WORKING WITH MULTIPLE FILES

```
// creating files with open() function

#include<iostream.h>

#include<fstream.h>

void main()

{
ofstream fout;                    // Create outputstream object

fout.open(“country”);             //opens country file in output mode

fout<<”United States of America”<<endl;

fout<<”United Kingdom”<<endl;

fout.close()                      // disconnects country file from output mode


fout.open(“capital”);             //opens capital file in output mode

fout<<”Washington”<<endl;

fout<<”London”<<endl;

fout.close();                     // disconnects capital file from output mode

// Reading the files

char line[80];

ifstream fin;                     //Create inputstream object

fin.open(“country”);              // opens country file in input mode

 cout<<”CONTENTS OF COUNTRY FILE”<<endl;

while(fin)                        //Check end-of-file

{

        fin.getline(line,80);

        cout<<line;

}
fin.close();                    //disconnect country file from input mode.

fin.open("capital");            // opens capital file in input mode

 cout<<"CONTENTS OF CAPITAL FILE"<<endl;

while(fin)                      //Check end-of-file

{

        fin.getline(line,80);

        cout<<line;

}

fin.close();                    //disconnect capital file from input mode.

getch();

}
```

OUTPUT:

```
CONTENTS OF COUNTRY FILE

United States of America

United Kingdom

CONTENTS OF CAPITAL FILE

Washington

London
```

READING FROM TWO FILES SIMULTANEOUSLY

```
//This program reads contents of the files country and capital created in the previous program

#include<iostream.h>

#include<conio.h>

#include<fstream.h>

#include<stdlib.h>                      // for exit() function

void main()

{

        char line[80];
        ifstream fin1,fin2;
        fin1.open("country");
        fin2.open("capital");
        for(int i=1; i<=10; i++)
        {
                if(fin1.eof()!=0)                   //End-of file is true
                {
                        cout<<"Exit from country";
                        exit(1);
                }
                fin1.getline(line,80);
                cout<<"Capital of"<<line;
                if(fin2.eof()!=0)                   //End-of file is true
                {
                        cout<<"Exit from capital";
                        exit(1);
                }
                fin2.getline(line,80);
                cout<<"is"<<line<<endl;
        }
getch();
}
```

OUTPUT:

```
Capital of United States of America is Washington
Capital of United Kingdom is London
```

FILE MODES

The general form of open() with two arguments is as follows:

stream_object.open("filename", mode);

Name	**Description**
ios::in	Open file to read
ios::out	Open file to write
ios::app	Append to end-of-file. It calls ios::out
ios::ate	Goto the end-of-file on opening
ios::trunc	Deletes all previous content in the file. (empties the file)
ios::nocreate	If the file does not exist, opening it with the open() function fails
ios::noreplace	If the file exists, trying to open it with the open() function, returns an error.
ios::binary	Opens the file in binary mode.

EXAMPLE FOR FILE MODES:

```
#include<iostream.h>

#include<fstream.h>

#include<iomanip.h>

void main()

{
float f1=123.45, f2=34.65, f3=56;

/*Open a file "pay.txt" in output mode and truncate its contents if exists.*/

        ofstream fout("pay.txt", ios::trunc);    // File opened using constructor function

        fout<<setprecision(2)<<setiosflags(ios::showpoint);

        fout<<setw(10)<<f1<<endl;
        fout<<setw(10)<<f2<<endl;

        fout<<setw(10)<<f3<<endl;

        getch(); }
```

```
OUTPUT:

  123.45

   34.65

   56.00
```

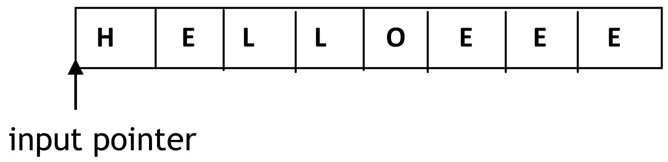
FILE POINTERS AND MANIPULATORS

- The file management system associates two pointers with each file, called as file pointers.
- The input pointer is called as *get pointer*.
- The output pointer is called as *put pointer*.

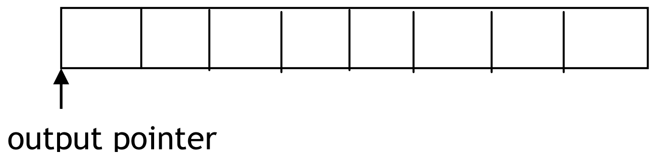
DEFAULT ACTIONS

- The action on file pointers while opening a file is given below:

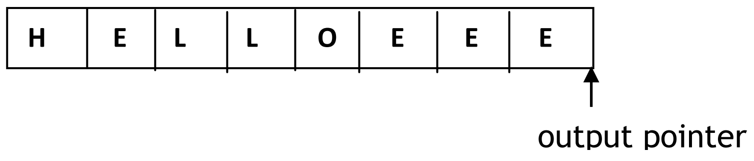
1) **READ MODE:**



2) **WRITE MODE:**



3) **APPEND MODE:**



FUNCTIONS FOR MANIPULATION OF FILE POINTERS

Function	Member of the class	Action performed
seekg()	ifstream	Moves get pointer to specific location
seekp()	ofstream	Moves put pointer to specific location
tellg()	ifstream	Returns current position of get pointer
tellp()	ofstream	Returns current position of put pointer

- seekg() and seekp() can also be used with two arguments:

```
seekg(offset,refposition);      //Moves get pointer
seekp(offset,refposition);      //Moves put pointer
```

The parameter *offset* represents the number of bytes the file pointer is to be moved from the location specified by the *refposition*. *refposition* can be one of the following:

• ios::beg	-	start of the file
• ios::cur	-	current position of the pointer
• ios::end	-	end of the file.

SEEK CALLS AND THEIR ACTIONS

SEEK CALL	ACTION PERFORMED
fout.seekg(0,ios::beg)	Goto the beginning of the file
fout.seekg(0,ios::cur)	Stay at the current file
fout.seekg(0,ios::end)	Goto the end of the file
fout.seekg(n,ios::beg)	Move to n+1 byte location from beginning
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.seekg(-n,ios::end)	Move backward by n bytes from end.
fout.seekp(n,ios::beg)	Move put pointer to n+1 byte location
fout.seekp(-n,ios::cur)	Move put pointer backward by n bytes

ACCESS TO FILES

- C++ file stream system supports a variety of functions to perform the input-output operations on files.
- The functions **get() and put()** are used to manage single character at a time.
- The functions **read() and write()** are used to manipulate blocks of characters
- The contents of the file can be accessed in two ways:
  - Sequential Access
  - Random Access.

1.SEQUENTIAL ACCESS:

- A sequential file has to be accessed sequentially.
- To access the particular data in the file, all the preceding data items have to be read and discarded.

Example:Sequential access to file

```
#include<iostream.h>

#include<fstream.h>

#include<string.h>

void main()

{

char c;

char str[75];

int len;

fstream file("student.txt", ios::in|ios::out);

cout<<"Enter the string:";

cin.getline(str,75);

len=strlen(str);

for (int i=0; i<len; i++)

{

        file.put(str[i]);               // write a character to the file

}

file.seekg(0);                          // Goto the start of file

char ch;

cout<<"The string read from the file is:";

while(file)

{

        file.get(ch);                   // Read a character from the file

        cout<<ch;                       // Display it on the screen

}

getch();

}
```

OUTPUT:

```
Enter the string: C++ Programming

The string read from the file is:  C++ Programming
```

BINARY FILES

- The binary file stores the data in binary code format.

Opening a binary file:

Syntax:

filestream_class stream_object;

stream_object.open("Filename", ios::binary);

INPUT/OUTPUT OPERATIONS ON BINARY FILES

- To store and retrieve data in binary form, the member functions **write()** and **read()** can be used.
- The read and write functions have the following syntax:
    1. **streamobject.read((char *) &var, sizeof(var));**
    2. **streamobject.write((char*) &var, sizeof(var));**

Example : Input/Output Operations on Binary Files

```
#include<iostream.h>

#include<conio.h>

#include<fstream.h>

void main()

{

int num1=530;

float num2=105.25;

// open file in write binary mode, write integer and close.

        ofstream fout("number", ios::binary);


        fout.write( (char *) &num1, sizeof(num1));

        fout.write( (char *) &num2, sizeof(num2));

        fout.close();

// open file in read binary mode, read integer and close.

        ifstream fin("number",ios::binary);
        fin.read( (char *) &num1, sizeof(num1));
        fin.read( (char *) &num2, sizeof(num2) );
        cout<<"Number1="<<num1 <<endl;
        cout<< "Number2="<<num2<<endl;
        fin.close();
        getch();
```

```
OUTPUT:
Number1= 530
Number2=1050.25
```

2.RANDOM ACCESS TO FILE

- A random file allows access to the specific data without the need for accessing its preceding data items. It can also be accessed sequentially.

Example: Random access to file

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
#define size 6
void main()
{
char str[size+1];
// Open file in binary input and output mode
        fstream file("test",ios::binary|ios::in|ios::out);
//Write the numbers 1 to 10 to the file
        for(int i=1; i<=10; i++)
        {
        file<<i;
        }
// Set the put pointer
        file.seekp(2);
        file<<"HELLO";
// Set the get pointer
        file.seekg(4);
        file.read(str,size);
        str[size]=0;                // End  of string
        cout<<str<<endl;
getch();
}
```

OUTPUT:

```
LLO789
```

In this program,

- The ASCII codes of the digits 1 to 10 are written to the file "test".
- put pointer is moved by an offset 2 from the beginning of the file and overwrites the numbers 3 through 6 with the string HELLO.
- Then, it reads 6 characters from the offset 4 into the character array 'str'

}

==

NAMESPACE

- ANSI C++ standard has added a new keyword namespace to define a scope to hold the global identifiers.
- All the standard library classes, functions and templates are defined within the namespace named **std.** The std namespace can be included in the program by the following syntax.

  using namespace std;

Definition:

The namespace provides an enclosure of logical nature which helps libraries to have *separate existence and solves the name-conflict problem*

Syntax for defining a namespace:

```
        namespace namespace_name
        {
        //declaration of variables, functions or classes.
        }
```

NAMESPACE CAN BE USED IN TWO WAYS:

1. Using Directive
2. Using Declaration

USING DIRECTIVE:

- The defined namespace must be included in the current scope of the program. This is done by the keyword 'using'.
- **Syntax:**

  using namespace namespace_name;
- This using keyword is used to include the namespace to the current scope of the program.

Example:

```
#include<iostream>
using namespace std;              // Namespace directive is used
void main()
{
cout<<"HELLLO";                   // cout and insertion operator is in the scope of std.
}
```

USING DECLARATION:

- If the namespace is not included as directive, the members of the namespace can be accessed by the name of the namespace and the Scope Resolution Operator('::').
- **Syntax:**

  using namespace_name :: member_name;

Example:

```
#include<iostream>
void main()
{
using std::cout;                  // cout is declared as the member of std namespace
using std:: operator <<;          // <<  is declared as the member of std namespace
cout<<"HELLLO";
}
```

NESTING OF NAMESPACES

A namespace can be nested inside another namespace as follows:

```
namespace ns1
{
        ……
        namespace ns2
        {
        …..
        }
}
```

- The members of the namespace can be accessed either by using the directive or by declaration

UNNAMED NAMESPACES

- An unnamed namespace does not have a name. The members of the unnamed namespace occupy the global scope.
- A common use of unnamed namespace is to shield global data from potential name classes between files. Every file may have a unique unnamed namespace.

EXAMPLE FOR NESTING OF NAMESPACES AND UNNAMED NAMESPACE

```
#include<iostream>
using namespace std;
/*Namespace definitions*/
namespace ns1                           // Nesting of namespace ns1 and ns2
{
        double x=4.56;
        int m=100;
        namespace ns2
        {
                double y=1.23;
        }
}
```

```
namespace ns3                          // Defining ns3 namespace
{
        int m=200;
        int n=100;
}
namespace                              // Defining unnamed namespace
{
        int k=10;
}
int main()
{
using namespace ns1;    // Including directive, brings the members of ns1 to current scope.
        cout<<"Members of namespace1 and namespace2"<<endl;
        cout<<"x="<<x<<endl;
        cout<<"m="<<m<<endl;
        cout<<"y="<<ns2::y<<endl;
//without including ns3 namespace directive.
        cout<<"Members of namespace3"<<endl;
        cout<<"m="<<ns3::m<<endl;
        cout<<"n="<<ns3::n<<endl;
// Accessing the members of unnamed namespace
        cout<<"Members of unnamed namespace"<<endl;
        cout<<"k="<<k<<endl;
return 0;
}
```

OUTPUT:

```
Members of namespace1 and namespace2
x=4.56
m=100
y=1.23
Members of namespace3
m=200
n=100
Members of unnamed namespace
k=10
```

FUNCTIONS INSIDE NAMESPACE SCOPE:

Functions can also be declared and defined in the namespace.

```
#include<iostream>

using namespace std;

namespace ns1                        // Defining namespace

{
        int divide(int x, int y)     // Function definition

        {

        return (x/y);

        }

        int prod(int x, int y);      // Function declaration only.

}

int ns1::prod(int x, int y)

{
        return (x*y);

}

int main()

{

        cout<<”Without including directive”<<endl;

        cout<<”Division=”<<ns1::div(10,2)<<endl;

        cout<<”Product=”<<ns1::prod(10,5)<<endl;

        using namespace ns1;         // Namespace directive included

        cout<<”Including Directive”<<endl;

        cout<<”Division=”<<div(10,5)<<endl;

        cout<<”Product=”<<prod(20,4)<<endl;

return 0;

}
```

OUTPUT:

```
Without including directive
Division=5
Product=50
Including Directive
Division=2
Product=80
```

CLASSES INSIDE NAMESPACE SCOPE:

```
#include<iostream>
using namespace std;
namespace ns1                                   // Namespace definition
{
        class test
        {
        public:
                int m;
                test( int a)                    // Parameterized constructor
                {
                m=a;
                }
                void display()
                {
                cout<<"m="<<m<<endl;
                }
        };
}
int main()
{
        // Without using directive
                ns1::test t1(100);
                t1.display();
        // Including the directive
                using namespace ns1;
                test t2(200);
                t2.display();
return 0;

}
```

OUTPUT:

```
m=100
n=200
```

===

ANSI STRING OBJECTS 0R MANIPULATING STRINGS

- A string is a sequence of characters.
- C++ does not support the built-in string datatype.
- So, null-terminated character arrays and character pointers are used for storing and manipulating strings.
- Operations with character arrays and character pointers are complex and inefficient.

String class:

- To overcome these difficulties, ANSI standard C++ provides a new class called string.
- For using the string class, we must include <string> in the program.
- The string class is very large and includes member functions and operators. Using the string class, the following operations can be performed.
    1. Creating string objects
    2. Reading string objects from the keyboard.
    3. Displaying the string objects to the keyboard.
    4. Modifying string objects
    5. Comparing string objects.
    6. Adding string objects etc.,

Important functions supported by string class:

Function	Task
append()	Appends a part of string to another
assign()	Assigns a partial string
capacity()	Gives the total elements that can be stored
compare()	Compares a string against another string
empty()	Returns true if the string is empty, otherwise returns false
erase()	Removes characters as specified

insert()	Inserts characters at a specified location
length()	Gives the number of elements in a string
max_size()	Gives the maximum possible size of string object in given system
replace()	Replace specified characters with a given string
size()	Gives the number of characters in the string
swap()	Swaps the given string with the other

CREATING STRING OBJECTS

The following program illustrates the different ways of creating the string objects.

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
// Creating string objects
        string s1;                    // Empty string object
        string s2("New");             // Using string constant
// Assigning value to string objects
        s1=s2;
        cout<<"s1="<<s1<<endl;
// Using another object
        string s3(s1);
        cout<<"s3="<<S3<<endl;
// Reading through keyboard
        string s4;               // Creating empty string object
        cout<<"Enter a string:";
        cin>>s4;
        cout<<"s4="<<s4<<endl;
// Concatenating strings
        s1=s3+s4
        cout<<"Updated s1="<<s1<<endl;
return 0;   }
```

OUTPUT:

```
s1=New
s3=New
Enter a string:
s4=Delhi
Updated s1=NewDelhi
```

MANIPULATING STRING OBJECTS:

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
        string s1(“12345”);
        string s2(“abcde”);
        cout<<”Original strings are:”<<endl;
        cout<<”s1=”<<s1<<endl;
        cout<<”s2=”<<s2<<endl;
// Inserting a string into another
        cout<<”Place s2 inside s1”<<endl;
        s1.insert(4,s2);
        cout<<”Modified s1=”<<s1<<endl;
// Removing characters in a string
        cout<<”Removing 5 characters from s1”<<endl;
        s1.erase(4,5);
        cout<<”Now s1=”<<s1<<endl;
// Replacing characters in a string
        cout<<”Replace 3 characters in s2 with s1”<<endl;
        s2.replace(1,3,s1);
        cout<<”Now s2=”<<s2<<endl;
return 0;
}
```

OUTPUT:

```
Original strings are:
s1=12345
s2=abcde
Place s2 inside s1
Modified s1=1234abcde5
Remove 5 characters from s1
Now s1=12345
Replace 3 characters in s2 with s1
Now s2=a12345e
```

RELATIONAL OPERATIONS ON STRING OBJECTS AND SWAPPING STRING OBJECTS

- The **compare()** function is used to perform relational operations on strings.

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
string s1("ABC");
string s2("XYZ");
string s3=s1+s2;
if(s1!=s2)
        cout<<"s1 is not equal to s2"<<endl;
if(s1>s2)
        cout<<"s1 is greater than s2"<<endl;
else
        cout<<"s2 is greater than s1"<<endl;
if(s3 == s1+s2)
        cout<<"s3 is equal to s1+s2"<<endl;
int x=s1.compare(s2);
if(x==0)
        cout<<"s1==s2"<<endl;
else if(x>0)
        cout<<"s1>s2"<<endl;
else
        cout<<"s1<s2"<<endl;
// Swapping two strings
        cout<<"Before swap:"<<endl;
        cout<<"s1="<<s1<<endl;
        cout<<"s2="<<s2<<endl;
```

OUTPUT:

```
s1 is not equal to s2
s2 is greater than s1
s3 is equal to s1+s2
s1<s2
Before swap:
s1=ABC
s2=XYZ
After swap:
s1=XYZ
s2=ABC
```

```
        s1.swap(s2);
        cout<<"After swap:"<<endl;
        cout<<"s1="<<s1<<endl;
        cout<<"s2="<<s2<<endl;
return 0;
}
```

STRING CHARACTERISTICS:

- The string class supports functions that could be used to obtain the characteristics of strings such as size(), length(), capacity(), max_size(), empty().

```
#include<iostream>
#include<string>
using namespace std;
void display(string str)
{
        cout<<"Size="<<str.size()<<endl;
        cout<<"Length="<<str.length()<<endl;
        cout<<"Capacity="<<str.capacity()<<endl;
        cout<<"Maximum size="<<str.max_size()<<endl;
        cout<<"Empty?"<<(str.empty() ? "yes" : "no")<<endl;
}
int main()
{
        string str;                     // Empty string object
        cout<<"Initial status:"<<endl;
        display(str);
        cout<<"Enter a word of string:"<<endl;
        cin>>str;                       // String initialized through keyboard
        cout<<"Updated status:"<<endl;

        display(str);

return 0;

}
```

OUTPUT:

```
Initial status:
Size=0
Length=0
Capacity=31
Maximum size=4294967293
Empty? yes

Enter a word of string:
HELLOEEE
Updated status:
Size=8
Length=8
Capacity=31
Maximum size=4294967293
Empty? no
```

==

STANDARD TEMPLATE LIBRARY

- The collection of generic classes and functions is called the standard template library.
- STL contains most useful algorithms and data structures.

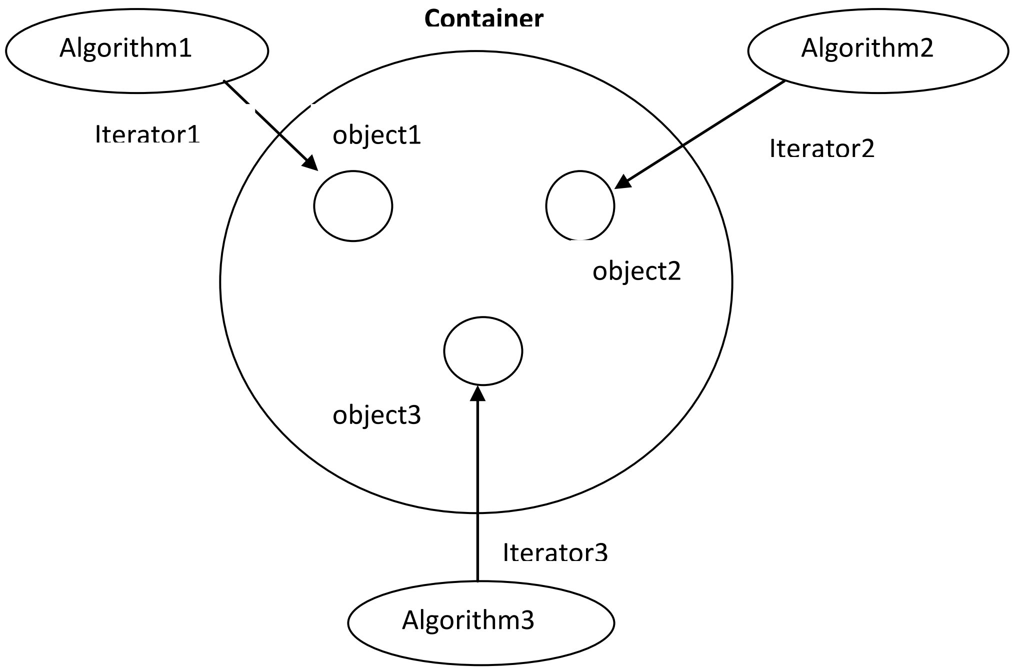
COMPONENTS OF STL

The STL contains several components. The three components are:

1. Containers
2. Algorithms
3. Iterators

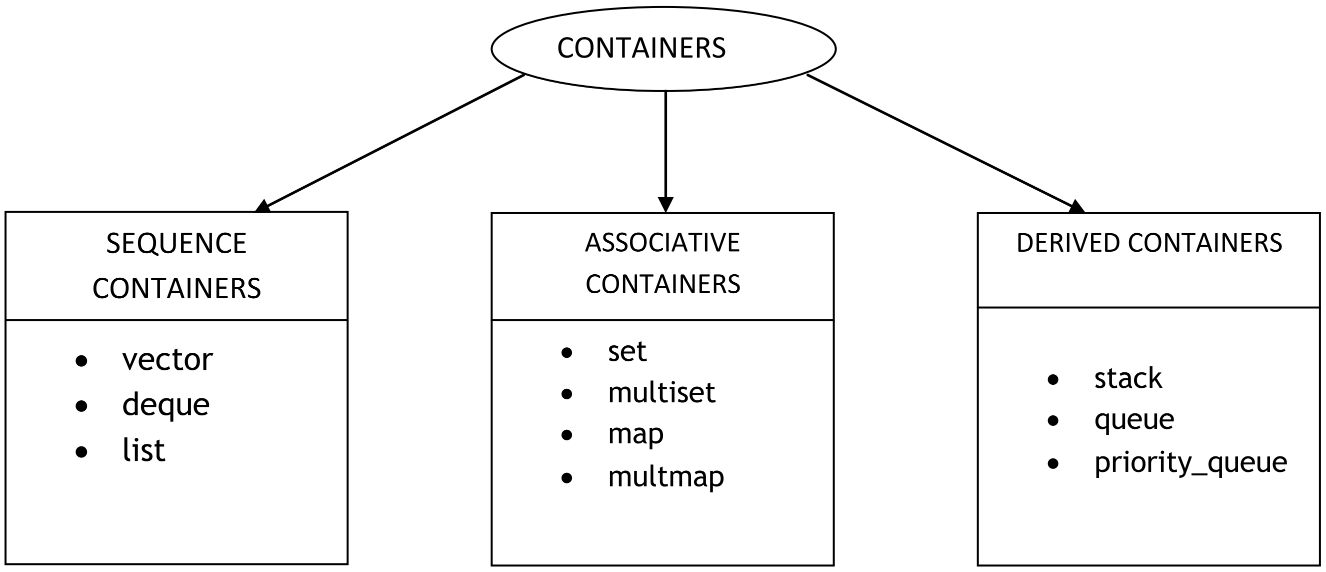
The algorithm employs iterators to perform operations stored in containers.

Relationship between the three STL Components:



CONTAINERS

- A container is an object that actually stores the data. It is the way the data is organized in memory.
- The STL containers are implemented by template classes and therefore can hold different types of data.



CONTAINERS SUPPORTED BY STL

1. SEQUENCE CONTAINERS:

- Sequence containers store elements in a linear sequence. The three types of sequence containers are explained as below:

CONTAINER	DESCRIPTION	HEADER FILE	ITERATOR
vector	A dynamic array. Allows insertions and deletions at back	<vector>	Random access
list	A bidirectional linear list. allows insertions and deletions anywhere	<list>	bidirecitonal
deque	A double ended queue. allows insertions and deletions at both the ends	<deque>	Random access

2. ASSOCIATIVE CONTAINERS:

- Associative containers are designed to support direct access to elements using keys.

CONTAINER	DESCRIPTION	HEADER FILE	ITERATOR
set	An associative container for storing unique sets	<set>	bidirectional
multiset	An associative container for storing non-unique sets	<set>	bidirectional
map	An associative container for storing unique key/value pairs. Each key is associated with one value	<map>	bidirectional
multimap	An associative container for storing key/value pairs in which one key is associated with more than one value	<map>	Bidirectional

3. DERIVED CONTAINERS

- The STL provides three derived containers namely stack, queue and priority_queue. They are also called as container adapters.

CONTAINER	DESCRIPTION	HEADER FILE	ITERATOR
stack	A standard stack. Last-in-first-out(LIFO)	<stack>	No iterator
queue	A standard queue. First-in-first-out(FIFO)	<queue>	No iterator
priority_queue	A priority queue. The first element is always the highest priority element	<queue>	No iterator

ALGORITHMS

- An algorithm is a procedure that is used to process the data contained in the containers.
- STL includes different kinds of algorithms to provide support for initializing, searching, copying, sorting, merging etc.,
- Algorithms are implemented by template functions.
- STL algorithms are categorized as:
    1. Retrieve and non-mutating algorithms
    2. Mutating algorithms
    3. Sorting algorithms
    4. Set algorithms
    5. Relational algorithms
    6. Numeric algorithms

1. NON-MUTATING ALGORITHMS:

Operations	Description
count()	Counts occurrence of a value in sequence
count_if()	Count number of elements that matches a predicate
equal()	True if two ranges are same
find()	Finds first occurrence of a value in a sequence
find_end()	Finds last occurrence of a value in a sequence
search()	Finds a subsequence within a sequence

2. MUTATING ALGORITHMS:

Operations	Description
copy()	Copies a sequence
copy_backward()	Copies a sequence from the end
fill()	Fills a sequence with a specified value
fill_n()	Fills first n elements with a specified value
generate()	Replaces all elements with the result
generate_n()	Replaces first n elements with the result
remove()	Deletes elements of specified value
replace()	Replaces elements with specified value
reverse()	Reverses the order of elements
rotate()	Rotate elements
swap()	Swaps two elements

3. SORTING ALGORITHMS:

Operations	Description
binary_search()	Conducts a binary search on an ordered sequence
make_heap()	Makes a heap from a sequence
merge()	Merges two sorted sequences
partial_sort()	Sorts a part of a sequence
partition()	Places elements matching a predicate first
pop_heap()	Deletes the top element
push_heap()	Adds a element to the heap
sort()	Sorts a sequence
sort_heap()	Sorts a heap

4. SET ALGORITHMS:

Operations	Description
includes()	Finds whether a sequence is a subsequence of another
set_difference()	Constructs a sequence which is the difference of two ordered sets
set_intersection()	Constructs a sequence which is the intersection of two ordered sets
set_symmetric_difference()	Produces a set which is the symmetric difference between two ordered sets
set_union()	Produces sorted union of two ordered sets

5. RELATIONAL ALGORITHMS

Operations	**Description**
equal()	Finds whether two sequence are the same
lexicographical_compare()	Compares alphabetically one sequence with the other
max()	Gives the maximum of two values
max_element()	Finds the maximum element in the sequence
min()	Gives the minimum of two values
min_element()	Finds the minimum element in the sequence

6. NUMERIC ALGORITHMS

STL contains few numeric algorithms in the header file called <numeric>

Operations	**Description**
accumulate()	Accumulates the results of operation on a sequence
adjacent_difference()	Produces a sequence from another sequence
inner_product()	Accumulates the results of operation on a pair of sequences
partial_sum()	Produces a sequence of operation on a pair of sequences

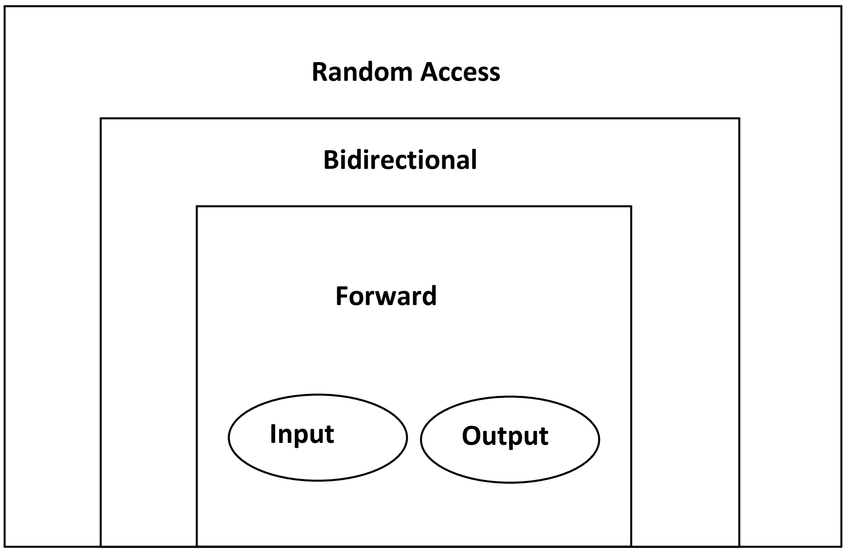
ITERATORS:

- An iterator is an object that points to an element in a container.
- Iterators are used to move through the contents of container.
- It can be incremented or decremented.
- Iterators connect the algorithms to containers and play a key role in the manipulation of data stored in the containers.

Iterators and their characteristics:

Iterator	**Access method**	**Direction of movement**	**I/O capability**
Input	Linear	Forward only	Read only
Output	Linear	Forward only	Write only
Forward	Linear	Forward only	Read/Write
Bidirectional	Linear	Forward and Backward	Read/Write
Random	Random	Forward and Backward	Read/Write

Functionality Venn Diagram of Iterators:



- The **input and output iterators** support the least functions. They can be used only to traverse in a container.
- The **bidirectional iterators** supports all forward iterator operations and provides the ability to move in the backward direction in the container.
- The **random access iterator** combines the functionality of birectional iterator with the capability to jump to a specified location.

SIMPLE STL PROGRAM:

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

void main()
{
  list <int> l;
  int j;
  l.push_back(10);
  l.push_back(20);
  l.push_back(30);
  l.push_back(40);
  l.push_back(50);
  int m=l.size();
  cout<<"Total Elements : "<<m;
  cout<<"\n They are : ";
  cout<<"\nForward Direction : ";
  for(j=0;j<m;++j)
  {
    cout<<l.front()<<" \t";
    l.pop_front();
  }
}
```

OUTPUT

```
Total Elements : 5
They are :
Forward Direction : 10      20      30      40      50
```