UNIT-III

Abstract Class

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. Here is the syntax for a pure virtual function,
virtual void f() = 0;

```
class Base          //Abstract base class
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};

class Derived:public Base
{
 public:
 void show()
 { cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
 Base obj;       //Compile Time Error
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Pure Virtual definitions

- Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.
- Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

```
class Base          //Abstract base class
{
 public:
 virtual void show() = 0;           //Pure Virtual Function
};

void Base :: show()        //Pure Virtual definition
{
 cout << "Pure Virtual definition\n";
}

class Derived:public Base
{
 public:
 void show()
 { cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
 Base *b;
 Derived d;
 b = &d;
 b->show();
}
```

Output :

```
Pure Virtual definition
Implementation of Virtual Function in Derived class
```

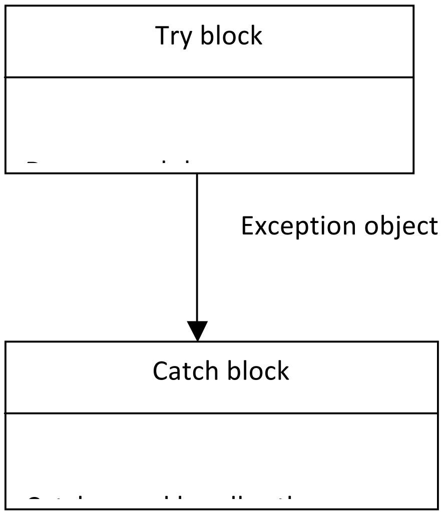
Exception handling

Exceptions are run time anomalies. They include conditions like division by zero or access to an array outside to its bound etc.

Types: Synchronous exception
Asynchronous exception.

- Find the problem (Hit the exception)
- Inform that an error has occurred. (Throw exception)
- Receive error information (Catch exception)
- Take corrective action (Handle exception)

C++ 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 appropitely.



- try block throwing an exception
- invoking function that generates exception
- throwing mechanism
- catching mechanism
- multiple catch statements
- catch all exceptions
- Rethrowing an exception

General form

```
try
   {
     …..
    throw exception;
     …….
   }
```

```
        Catch ( type arg)
       {
         ……
        }
```

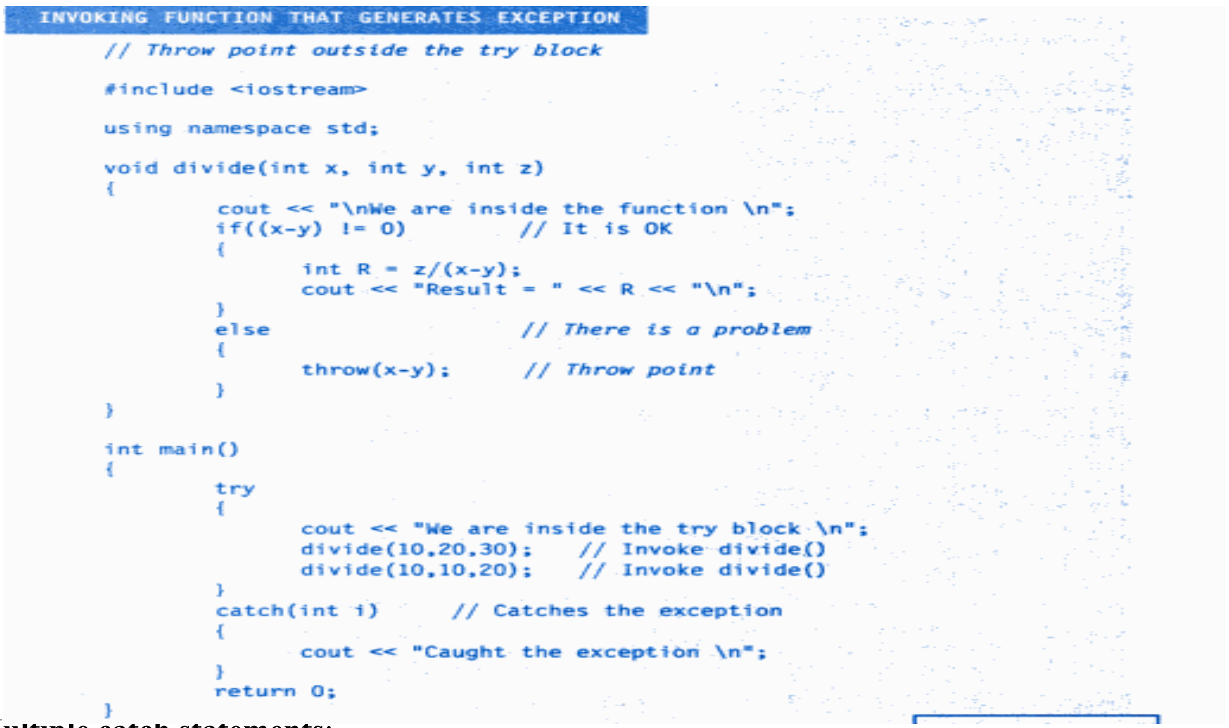
Example:

```
#include<iostream.h>
#include<conio.h>
class MyException
{
int n;
public:
MyException(int a)
{
n=a;
}
int Divide(int x)
{
try
{
if(x==0)
{
throw x;
cout<<"This statement will not be executed\n";
}
else
{
cout<<"Trying division succeeded\n";
cout<<"\n Quotient = "<<n/x;
}
}
catch(int)
{
cout<<"Exception : Division by zero. Check the value";
}
return 0;
}
};
void main()
{
int a;
cout<<"\n Enter a Number : ";
cin>>a;
MyException ex(10);
ex.Divide(a);
}
```

Exceptions that has to be caught when functions are used- The form is as follows:

```
Type function (arg list)
{
 ……
  Throw (object)
  …..
}
try
    {
      …..
     Invoke function here;

      …….
    }
    Catch ( type arg)
   {
    Handles exception here
   }
```



INVOKING FUNCTION THAT GENERATES EXCEPTION

```
// Throw point outside the try block

#include <iostream>

using namespace std;

void divide(int x, int y, int z)
{
        cout << "\nWe are inside the function \n";
        if((x-y) != 0)          // It is OK
        {
              int R = z/(x-y);
              cout << "Result = " << R << "\n";
        }
        else                    // There is a problem
        {
              throw(x-y);       // Throw point
        }
}

int main()
{
        try
        {
              cout << "We are inside the try block \n";
              divide(10,20,30);    // Invoke divide()
              divide(10,10,20);    // Invoke divide()
        }
        catch(int i)      // Catches the exception
        {
              cout << "Caught the exception \n";
        }
        return 0;
}
```

Multiple catch statements:

```
try
{
  …..
 throw exception;
```

```
          …….
         }
         Catch ( type arg)
        {
         ……// catch block1
        }

           Catch ( type arg)
        {
         ……//catch block2
        }

   Catch ( type arg)
        {
         ……//catch block n
        }
```

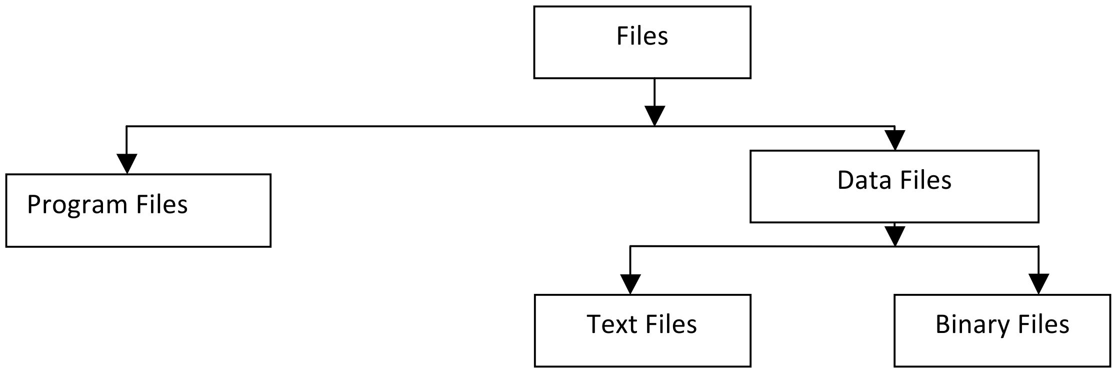
Generic exception handling is done using ellipse as follows:

```
   Catch ( . . .)
       {
         ……
        }
```

Streams and formatted I/O

File handling

Files are required to save our data for future use, as Ram is not able to hold our data permanently.



The Language like C/C++ treat every thing as a file , these languages treat keyboard , mouse, printer, Hard disk , Floppy disk and all other hardware as a file.

The Basic operation on text/binary files are : Reading/writing ,reading and manipulation of data stored on these files. Both types of files needs to be open and close.

How to open File

Using member function Open( )	**Using Constructor**

Syntax
 	
Filestream object;
 	
Object.open(“filename”,mode);
 	
Example
 	
ifstream fin;
 	
fin.open(“abc.txt”)
	Syntax
	
	Filestream object(“filename”,mode);
	
	Example
	
	ifstream fin(“abc.txt”);

NOTE: (a) Mode are optional and given at the end .

(a) Filename must follow the convention of 8.3 and it’s extension can be anyone

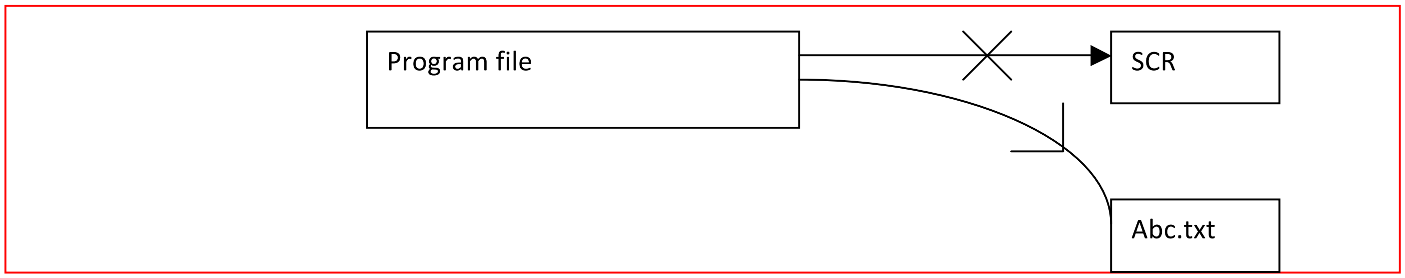
How to close file

All types of files can be closed using **close( )** member function

Syntax

```
fileobject.close( );
```

Objective : To insert some data on a text file



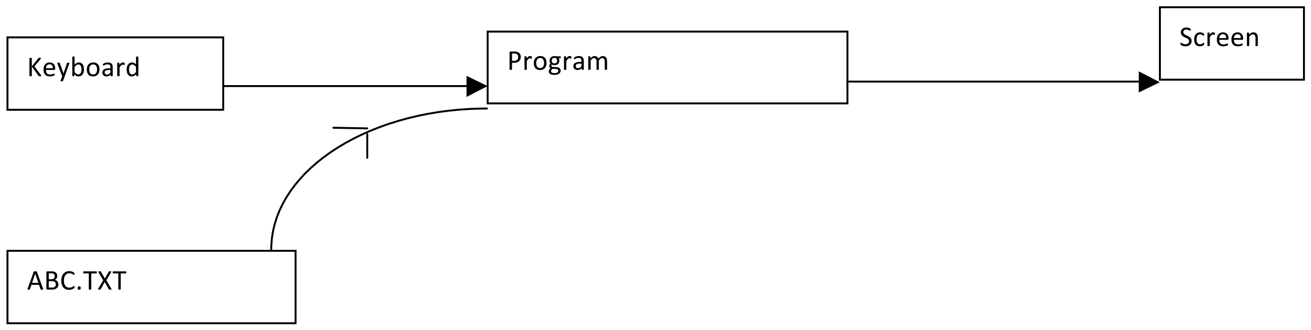
Program

Program	ABC.txt file contents
`#include<fstream>` `using namespace std;` `int main()` `{` `ofstream fout;` `fout.open("abc.txt");` `fout<<"This is my first program in file handling";` `fout<<"\n Hello again";` `fout.close();` `return 0;` `}`	This is my first program in file handling Hello again

```
#include<fstream>
using namespace std;
int main()
{
   ofstream fout;
   fout.open("abc.txt");
   fout<<"This is my first program in file
handling";
   fout<<"\n Hello again";
   fout.close();
   return 0;
}
```

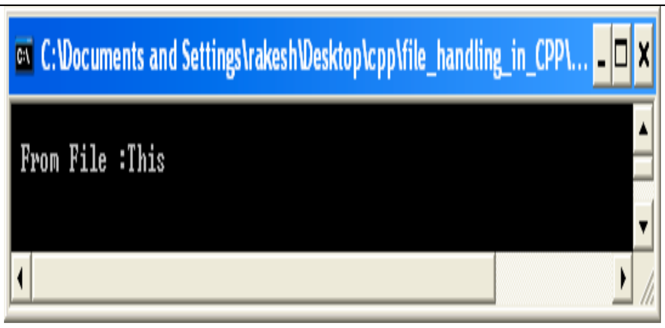
```
This is my first program in file handling
 Hello again
```

Reading data from a Text File



```
#include<fstream>
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
   ifstream fin;
   char str[80];
   fin.open("abc.txt");
   fin>>str;     // read only first //string
from file
   cout<<"\n From File :"<<str;   // as
//spaces is treated as termination point
   getch();
   return 0;
}
```

NOTE : To overcome this problem use



C:\Documents and Settings\rakesh\Desktop\cpp\file_handling_in_CPP\...

```
From File :This
```

fin.getline(str,79);	

Detecting END OF FILE

Using **EOF( ) member function**	Using **filestream object**

Syntax
 	
Filestream_object.eof( );
 	
Example
 	
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
int main()
{
char ch;
ifstream fin;
fin.open("abc.txt");
while(!fin.eof()) // using **eof()**
// function
{
fin.get(ch);
cout<<ch;
}
fin.close();
getch();
return 0;
}
	Example
	
	// detectting end of file
	#include<iostream>
	#include<fstream>
	#include<conio.h>
	using namespace std;
	int main()
	{
	char ch;
	ifstream fin;
	fin.open("abc.txt");
	while(fin) // file object
	{
	fin.get(ch);
	cout<<ch;
	}
	fin.close();
	getch();
	return 0;
	}

Example **: To read the contents of a text file and display them on the screen**.

Program (using **getline** member function)	Program (using **get( )** member function)

#include<fstream>
#include<conio.h>
#include<iostream>
using namespace std;
int main()
	#include<fstream>
	#include<conio.h>
	#include<iostream>
	using namespace std;
	int main()

```
{
   char str[100];
   ifstream fin;
   fin.open("c:\\abc.txt");
   while(!fin.eof())
    {
       fin.getline(str,99);
       cout<<str;
       }
   fin.close();
   getch();
   return 0;
}
```

```
{
   char ch;
   ifstream fin;
   fin.open("file6.cpp");
   while(!fin.eof())
    {
       fin.get(ch);
       cout<<ch;
       }
   fin.close();
   getch();
   return 0;
}
```

Standard template library

STL – standard template library:

The general purpose templatized classes (data st) and functions (algo) that could be used as a standard approach for storing and processing of data. collection of these generic classes and functions is called STL. STL save time and effort by reusing the well written and well tested components.

- STL components are part of standard C++ library and defined in namespace std as follows
  using namespace std;
  Components of STL
- 1.containers 2. algorithms 3.iterators

STL algorithms

- Retrieving or non-mutating algorithms
- Mutating algorithms
- Sorting algorithms
- Set algorithms
- Relational algorithms

Retrieving or non-mutating algorithms

- adjacent_find()   count()
- Count_if()   equal()
- Equal()   find()
- Find_end()   for_each()
- Mismatch()   search()
- Search_n()

Mutating algorithms

- Copy()   copy_backward()
- Fill()   fill_n()
- Generate()   generate_n()
- Iter_swap()   remove()
- Remove_copy()   replace()
- Swap()   reverse()
- Rotate()   transform()
- Unique()

Sorting algorithms

- Binary_search()   equal_range()
- Inplace_merge()   merge()
- Make_heap()   nth_element()
- Partial_sort()   partition()
- Pop_heap()   sort()
- Sort_heap()   stable_sort()
- Upper_bound()   sort_heap()

Set algorithms

- Set_difference()   set_intersection()
- Set_symmetric_difference()
- Set_union()

Relational algorithms

- Equal()
- Lexicographical_compare()
- Max()
- Max_element()
- Min_element()
- Mismatch()

Iterators

Iterators are like pointers and are used to access container elements.

5 types of iterators

- Input
- Output
- Forward
- Bidirectional
- Random

```
#include<iostream>                    //program using list container
#include<list>
#include<stdlib>   // for using rand()
Using namespace std;
Void display(list<int> &lst)
{  list<int> :: iterator p;
   for(p = lst.begin(); p!=lst.end(); ++p)
      cout<< *p;
}
Void main()
{
List<int> list1;                //empty list of 0 size
List<int> list2(5);   //empty list of 5 size
For(p=list2.begin(); p!=list2.end; ++p)
      *p = rand()/100;
Display(list1);
Display(list2);
List1.push_front(100);                //add 2 elements at both the ends
List1.push_back(200);
 list2.pop_front();
List<int> listA
List1.merge(list2);
Display(list1);
```

```
      listA=list1;
      listA.sort();
      listA.reverse();
      Display(listA);
      }
```

Multiset –

```
#include<iostream>
#include<vector>
#include<string>
#include<set>
Using namespace std;
Class student
{
            int rollno;
            String name;
Public : student(int trollno.string tname)
                          { rollno=trollno;   name= tname;  }
          ostream &operator <<( ostream & tempout, student tstu)
                         {  return tempout<<tstu.rollno<<tstu.name<<endl; }
         bool operator <(student tstu) const   // it is must for multiset
       {  return(rollno<tstu.rollno);  }
};
Void main()
{
Multiset<student>multisetstudent;
Student s1(123,”mohit”);
Student s2(124,”rohit”);
Student s3(125,”kumar”);
Student s4(126,”raj”);
Multisetstudent.insert(s1);
Multisetstudent.insert(s2);
Multisetstudent.insert(s3);
Multisetstudent.insert(s1);   // second time insert the same data
Multisetstudent.insert(s2);
Multisetstudent.insert(s3);
Multiset<student>::iterator multisetstudentindex;
For(multisetstudent=multisetstudent.begin();
      multisetstudent != multisetstudent.end();
      multisetstudentindex++)
{  cout<<*multisetstudentindex; }
}
```

Output

```
      123 mohit
123 mohit
124 rohit
124 rohit
125 kumar
      124 kumar
```

CLASS TEMPLATES:

Multiple Types,

Using default Arguments,

Subscript Operator Overloading,(All in one Program)

- It has only one argument, the index. int & operator [] (int Index)
- The call SafeArray[0] is similar to **SafeArray.operator[](0).**
- This IndexValue (0 in above case) is checked to see in the overloaded operator function if it is goes out of boundaries.

```
#include <iostream.h>

template <class T=int,int Size=10>
class SafeGenericArray
{
        T Array[Size];
public:

        T & operator [](int Index) //Subscript([]) operator overloading
        {
                if ((Index < 0) || (Index > Size - 1))
                {
                        cout << "Subscript Out of Range !";
                        exit(1);
                }
                else return Array[Index];
        }
};

void main()
{
        SafeGenericArray <int , 5> SafeIntArray1; // T = int and Size = 5
        SafeGenericArray <char> SafeCharArray1;// T = char ,Size = 10
        SafeGenericArray <> SafeIntArray2; // T = int and Size = 10
        SafeGenericArray<char,5> SafeCharArray2;
```

```
                //SafeGenericArray SafeIntArray;
                SafeIntArray1[0] = 5;

        }
```

FUNCTION TEMPLATES

A template is a new concept which enables us to define generic classes and functions that thus provides support **for generic programming**.

A template can be considered as a kind of **macro**.

Instantiation:

When the template function like Genericbubblesort<int>(Array2); ia called two things happen

a. The function is generated.i.e instantiation from the template definition for the respective type int.
b. The call is compiled

Syntax:

```
template <class T>

return_type  function_name (arguments_of_type_T)

{

        //;;;;;;;;;;;;;

        //Body of function

        // with type T

        // wherever appropriate

        //;;;;;;;;;;;;;;;;;;;

}
```

Program:

```
#include<iostrem.h>

tempale < class T>
```

```
void swap(T &x, T &y)

{

            T temp=x;

            x=y;

            y=temp;

}

void fun(int m, int n, float a, float b)

{

            cout<<" m and n before swap:"<<m<<" "<<n<<"\n";

            swap(m,n);

            cout<<"m and n after swap;"<<m<<" "<<n<<"\n";

           cout<<"a and b before swap:"<<a<<" "<<b<<"\n";

           swap(a,b);

           cout<<"a and b after swap:"<<a<<" "<<b<<"\n";

   }

  int main()

{

 fun(100,200,11.22,33.44);

return 0;

}
```

The output would be:

```
m and n before swap: 100 200

m and n after swap: 200 100

a and b before swap:11.22 33.439999

a and b after swap:33.439999 11.22
```

1. **WRITE NOTES ON STANDARD TEMPLATE LIBRARY**

- STL is Standard Template Library, a collection of generic software component (generic containers) and generic algorithms, glued by objects called Iterators.
- STL is a different type of library.
- It has quite a large number of non member functions designed to work on multiple classes of container types.
- Feasible to implement all operations in most efficient manner
- many useful algorithms like find(), replace(), merge(), sort() are implemented in STL
- Being non member function they can be used with any container even a newly designed one
- These containers are classes which can in turn contain other objects.
- Sequence containers
    - Vector, List, DeQueue
- Sorted associative containers
    - Set, Map, Multi-set, Multi-map
- Adapted containers
    - Stack , Queue

The Iterators

- These are pointer like *objects*
- They are categorized into various categories unlike pointers.
- The generic algorithms are written to work on iterators objects rather then any data structure.
- Containers are also designed in terms of iterators rather then normal pointers.

Types of Iterators

- Find algorithm works on input Iterators
- sort is defined to require Random Access Iterators
- list (the doubly linked list) container is defined to have Bi-directional Iterator
- Thus sort can not be used with list!

Generic programming

- Idea of Generic Programming has nothing to do with C++!
- It is a mechanism of designing generic software components like Vector, list, deque and so on.
- Unlike normal programming practice, Generic algorithms are not designed having any software component in mind.

The Containers

- STL provides tested and debugged software components available readily.
- They are reusable in the sense that we can use them as a building block for any other software development projects.