Templates and Exception Handling

Introduction to function templates and class templates:

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- A template is a blueprint or formula for creating a generic class or a function.
- The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.
- You can use templates to define functions as well as classes.
- Template is a new concept which enables us to define generic and functions and thus provides support for generic programming.
- Generic programming as an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.
- A template can be used to create a family of classes or functions.
- For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array.
- similarly, we can define a template for a function, say mul(),that would help us create versions of mul() for multiplying int, float and double type values.
- A template can be considered as a kind of macro.
- When an object of a specific type is define for actual use, the template definition for that class is substitute with the required data type.
- Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized class or functions.

Difference between Function template and class template

- Function template are those functions which can handle different data types without separate code for each of them for a similar operation on several kinds of data types, a programmer need not write different functions.
- Using class template we can write a class whose members use template parameters as type.
- Function overloading allow the definition of more than one function with the same name and provides a means of choosing between the functions based on parameter matching.
- Template functions tell the compiler how to create new functions based on the instantiation data type.
- Functions generated from the template behave like normal functions.
- The function body will differ in function overloading where as the function body will not differ in function template.

Class Template: Just like we can create function templates, we can also create class templates, allowing classes to have members that use template parameters as types. For example:

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type `int` with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

This same class could also be used to create an object to store any other type, such as:

```
mypair<double> myfloats (3.0, 2.18);
```

Class Template: Consider a vector class defined as follows:

```
Class vector
{
int *v ;
int size;
public:
vector(int m ) // create a null vector
{
v=new int[size = m];
for(int i=0;i<size;i++)
v[i]=0;
}
vector(int *a) //create a vector from an array
{
for(int i=0;i<size;i++)
v[i]=a[i];
}
int operator*9vector &y) //scalar product
{
int sum=0;
for(int i=0;i<size;i++)
sum+=this->v[i]*y-v[i];
return sum;
}
};
```

Now suppose we want to define a vector that can store an array of float value. We can do this simply replacing the appropriate int declaration with float in the vector class. This means that we can have to redefine the entire class all over again.
Assume that we want to define a vector class with the data type as a parameter and then use this class to create a vector of any data type instead of defining a new class every time. The template mechanism enables us to achieve this goal. As mentioned earlier, template allows us to define generic classes. It is simple process to create a generic class using a template with an anonymous type.
The general format of a class template is: Template<class T>

```
class classname
{
...
//class member specification
//with anonymous type T
//whenever appropriate
….
….
};
```

The template definition of vector class shown below illutrates the syntax of a template:

```
template<class T>
class vector
{
T* v; // type T vector
```

Remember: The class template definition is very similar to an ordinary class definition except the prefix template<class T> and the use of type T. This prefix tells the complier that we are going to declare a template and use T as a type name in the Declaration. Thus, vector has become a parameterized class with the type T as its parameters. T may be substituted by any data type including the user defined types. Now we can create vectors for holding different data types.

Example:

```
vector<int> v1(10); //10 element int vector
vector<float> v2(30); //30 element float vector
```

The type T may represent a class name as well.

Example:

```
Vector<complex> v3 (5); // vector of 5 complex numbers
```

A class created from a class template is called a template class. The syntax for defining an object of a template class is:

```
Classname<type> objectname (arglist);
```

This process of creating a specific class from a class template is called instantiation. The complier will perform the error analysis only when an instantiating take place. It is, therefore, advisable to create and debug an ordinary class before converting it in to template.

Example of class template

```
 #include <iostream>
using namespace std;
const size=3;
template<class T>
class vector
{
T*v; // type T vector
public:
vector()
{
v=new T[size];
for(int i=0;i<size;i++)
v[i]=0;
}
vector(T* a)
{
for(int i=0;i<size;i++)
v[i]=a[i];
}
T operator*(vector &y)
{
```

```
T sum=0;
for(int  i=0;i<size;i++)
sum+=this->v[i]*y.v[i];
return  sum;
}
};
int  main()
{
int  x[3]={1,2,3};
int  y[3]={4,5,6};
vector<int>  v1;
vector<int>  v2;
v1=x;
v2=y;
int  R= v1*v2;
cout<<"R="<<r<<"\n";
return  0;
}
```

The output would be :
R=32

Another Example:

```
#include  <iostream>
using  namespace  std;
const size=3;
template<classT>
class vector
{
T*v; // type T vector
public:
vector ()
{
v=new  T[size];
for(inti=0;i<size;i++)
v[i]=0;
}
vector(T*  a)
{
for(int  i=0;i<size;i++)
v[i]=a[i];
}
T operator*(vector  &y)
{
T sum=0;
for(int=0;i<size;i++)
sum+=this->v[i]*y.v[i];
return  sum;
}
};
Int main()
```

```
{
float x[3]={1.1,2.2,3.3};
float y[3]={4.4,5.5,6.6};
vector <float> v1;
vector <float>v2;
v1=x;
v2=y;
float R=v1*v2;
cout<<"R="<<R<<"\n";
return 0;
}
```

The output would be:

```
R=38.720001
```

Class Templates with Multiple Parameters : We can use more than one generic data type in a class. They are declared as a comma-separated list within the template specification as shown below:

```
Template<class T1, class t2, …..>
class classname
{
……
……
…….
};
```

Example with two generic data types:

```
 #include <iostream>
using namespace std;
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<<"and"<<<<"\n";
}
};
int main()
{
Test<float,int> test1(1.23,123);
Test<int,char> test2(100,'W');
test1.show();
test2.show();
return 0;
};
```

Output would be:

```
1.23 and 123
100 and W
```

Function Templates: Like class templates, we can also define function templates that could be used to create a family of functions with different argument types. The general format of a function template is: template<class T>

```
returntype functionname (argument of type T)
{
//
//body of function
//with Type T
//whenever appropriate
//……………
}
```

- The function template syntax is similar to that of the class template except that we are defining functions instead of classes.
- We must use the template parameter T as and when necessary in the function body and its argument list.

The following example declares a swap () function template that will swap two values of a given type of data.

```
template <class T>
void swap(T&x , T&y)
{
T temp =x;
x=y;
y=temp;
}
```

- This essential declares a set of overloading functions, one for each type of data.
- We can invoke the swap () function likes any ordinary function .for example, we can apply the swap () function as follows:

```
void f ( int m , int n , float b )
{
swap ( m , n); //swap two integer values
swap ( a , b); //swap two float values
//……..
}
```

- This will generate a swap () function template for each set of argument types.
- Example will show how a template function is defined and implemented.

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

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
```

Another function often used is sort () for sorting arrays of various types such as int and double. The following shows a function template for bubble sort:

```
Template<class T>
bubble( T a[] , int n )
{
for( int i=0 ; i<n-1 ; i++ )
for( int j=n-1 ; i<j ; j-- )
if ( a[j] < a[j-1] )
{
T temp = v[j];
v[j]=v[j-1];
v[j-1]=temp;
}
}
```

Note that the swapping statements

```
T temp = v[j];
v[j]=v[j-1];
v[j-1]=temp;
```

May be replaced by the statement

```
swap( v[j], v[j-1] );
```

Where swap () has been defined as a function template. Here is another example where a function returns a value.

```
template<class T>
T max ( T x, T y)
```

```
{
return x>y ? x : y;
}
```

Program of Bubble Sort demonstrates the use of two template functions in nested from for implementing the bubble sort algorithm.

```
# include <iostream>
using namespace std ;
template<class T>
void bubble (T a[], int n)
{
for (int i=0; i<n-1; i++)
for (int j=n-1; i<j; j--)
i f (a[j] < a[j-1])
{
swap (a [j],a[j-1];// calls template function
}
}
templte<class X>
void swap(X &a, X &b)
{
x temp=a;
a= b;
b= temp;
}
int main()
{
int x[5] = { 10,50,30,40,20,};
float y[5] ={1.1,5.5,3.3,4.4,2.2,};
bubble ( x , 5 ); // calls template function for int values
bubble ( y , 5 ); // calls template function for floate values
cout << “sorted x-array:”;
for ( int i=0; i<5; i + + )
cout << x[i] << “ ” ;
cout << endl ;
cout << “ Sorted y-array : ” ;
for (int j=0; j<5; j + +)
cout << y[j] << “ ”;
cout << endl ;
return 0;
}
```

The output would be:

```
sorted x-array: 10 20 30 40 50
sorted y- array: 1.1 2.2 3.3 4.4 5.5
```

Another example:

```
#include <isotream>
#include <iomanip>
#include <cmath>
using namespace std;
```

```
template <class T>
void rootes(T a,T b,T c)
{
T d = b*b - 4*a*c;
if (d= = 0) // Roots are equal
{
cout << "R1 = R2 = " << -b/(2*a) << endl;
}
else if (d>0) //two real roots
{
cout<<"roots are real \n";
float R =sqrt (d);
float R1 = (-b+R)/(2*a);
float R2 = ( -b+R )/(2*a);
cout<< "R1 = "<< R1 << " and";
cout <<R2 = "<< R2 << endl;
}
else // roots are complex
{
cout <<"roots are complex \n";
float R1 = -b/( 2*a);
float R2 = sqrt( -d )/( 2*a );
cout <<" real part = " << R1 << endl;
cout<< "imaginary part =" << R2;
cout<< endl;
}
}
int main()
{
cout<< "integer coefficients \n";
roots(1,-5 ,6);
cout << "\n float coefficients \n";
roots (1.5, 3.6, 5.0);
return 0;
}
```

Output would be :

```
integer coefficients
roots are real
R1= 3 and R2 =2
float coefficients
roots are complex
real part = -1.2
imaginary part = 1.3757985
```

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
{
  // body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>
inline T const& Max (T const& a, T const& b)
{
   return a < b ? b:a;
}

int main ()
{
    int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;
   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;
    string s1 = "Hello";
    string s2 = "World";
    cout << "Max(s1, s2): " << Max(s1, s2) << endl;
    return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

Function Template with Multiple Parameters: Like template classes, we can use more than one generic data type in the template statement, using a comma-separated list shown below:

```
template<class T1 , class T2, …..>
returntype functionname(arguments of types T1, T2, …)
{
……
……
……
}
```

Example with two generic types in template functions:

```
#include <iostream>
#include<string>
using namespace std;
template<class T1,class T2>
void display( T1 x, T2 y)
{
cout<<x<<" "<<y<<"\n";
}
int main()
```

```
{
display(1999, “EBG”);
display(12.34, “1234);
return 0;
}
```

The output would be:

```
1999 EBG
12.34 1234
```

Overloading of Template Functions:

A template function may be overloaded either by template functions or ordinary functions of its name. In such cases, the overloading resolution is accomplished as follows:

1. Call an ordinary function that has an exact match.
2. Call a template function could be created with an exact match.
3. Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions.

Example for showing how a template function is overloaded with an explicit function.

```
#include <iostream>
#include <string>
using namespace std;
template <class T>
void display(T x)
{
cout<<”template display:” << x<< “\n”;
}
void display ( int x)
{
cout<<”Explicit display: “<< x <<”\n”;
}
int main()
{
display(100);
display(12.34);
display(‘c’);
return 0;
}
```

The output would be:

```
Explict display:100
template display:12.34
template display:c
```

Remember: The call display (100) invokes the ordinary version of display() and not the template version.

Member Function Templates :

- In the class template which we have discussed all the members are defined inline.
- But its not always necessary to define the members of the class inside.

- You can define them outside the class.These functions are defined by function templates.
- The syntax of member function templatesis as follows.
- When we create a class template for vector, all the member functions were defined as inline which was not necessary. We would have defined them outside the class as well. But remember that the member functions of the template classes themselves are parameterized by the type argument and therefore these functions must be defined by the function templates. It takes the following general form:

```
Template <class T>
returntype classname <T> :: functionname(arglist)
{
……
……..
…….
}
```

The vector class template and its member fnctions are redefined as follow:

```
// class template……….

template<class T>
class vector
{
T*v;
int size;
public:
vector(int m);
vector(T* a);
T operator*(vector & y);
};
//member function templates
template<class T>
vector<T> :: vector (int m );
{
v=new T[size=m];
for(int i=0; i<size ; i++)
v[i]= 0;
}
template<class T>
vector <T>::vector(t*a)
{
for(int i=0; i<size ; i++)
v[i]=a[i];
}
template< class T >
T vector < T > :: operator*(vector & y)
{
T sum =0;
for ( int i=0; i< size ; i++)
sum += this -> v[i]*y.v[i];
return sum;
}
```

```
Template<class type>
Returntype classname<Type>::function_name(arg_list)
{
//bode
}
```

Example of Member Function Templates

```
#include<iostream>
using namespace std;
template<class t>
class compare
{
T a,b;
public:
compare(t first, t second);
t max();
};
template<class t>
compare<t>::compare(t first, t second)
{
a=first;
b=second;
}
template<class t>
t compare<t>::max()
{
t val;
if(a>b)
val=a;
else
val=b;
return val;
}
int main()
{
compare<int>obj1(100,60);
compare<char>obj2('p','t');
cout<<"max(100,60)="<<obj1.max();
cout<<"max('p','t')="<<obj2.max();
}
```

Template Arguments:

- Templates can have multiple arguments.
- These arguments are normally denoted by the data type T.But along with T we can also define other data types arguments.
- For example we can define the template function as Compare(T first, T second);

The following program illustrates this concept.

```
#include<iostream>
```

```
using namespace std;
template<class T>
class compare
{
        T a,b;
        public:
        compare(T first, int second);
        T max();
};
template<class T>
compare<T>::compare(T first, int sec)
{
        a=first;
        b=sec;
}
template<class T>
T compare<T>::max()
{
        T val;
        if(a>b)
        val=a;
        else
        val=b;
        return val;
}
int main()
{
        compare<int>obj1(100,160);
        cout<<"maximum of (100,160)="<<obj1.max();
        return 0;
}
```

Non-Type Template Arguments : We have seen that a template can have multiple arguments. It is also possible to use non-type arguments. That is, in addition to the type argument T, we can also use other arguments such as strings, function names, constant expressions and built-in types. Consider the following example:

```
Template<class T, int size>
Class array
{
T a[size]; //automatic array initialization
//…………
//………..
};
```

This template supplies the size of the array as an argument. This implies that the size of the array is known to the complier at the compile time itself. The arguments must be specified whenever a template class is created. Example:

```
Array <int,10> a1; //array of 10 integers
Array <float,5> a2; //array of 5 floats
```

Array <char,20> a3; //string of size 20

The size is given as an argument to the template class.

Function overloading in C++:

- C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.
- An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).
- When you call an overloaded function or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.
- You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.
- Following is the example where same function print() is being used to print different data types:

```
#include <iostream>
using namespace std;
 class printData
{
public:

   void print(int i)
    {
          cout << "Printing int: " << i << endl;
     }

   void print(double  f)
    {
        cout << "Printing float: " << f << endl;
      }
    void print(char* c)
     {
        cout << "Printing character: " << c << endl;
      }
};

int main(void)
{
   printData pd;
    // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
```

```
  // Call print to print character
  pd.print("Hello C++");
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Example for function overloading:

- C++ program for function overloading.
- Function overloading means two or more functions can have the same name but either the number of arguments or the data type of arguments has to be different.
- Return type has no role because function will return a value when it is called and at compile time compiler will not be able to determine which function to call.
- In the first example in our code we make two functions one for adding two integers and other for adding two floats but they have same name and in the second program we make two functions with identical names but pass them different number of arguments. Function overloading is also known as compile time polymorphism.

```
#include <iostream>
 using namespace std;
 /* Function arguments are of different data type */
 long add(long, long);
float add(float, float);
 int main()
{
  long a, b, x;
  float c, d, y;
  cout << "Enter two integers\n";
  cin >> a >> b;
   x = add(a, b);
  cout << "Sum of integers: " << x << endl;
  cout << "Enter two floating point numbers\n";
  cin >> c >> d;
   y = add(c, d);
   cout << "Sum of floats: " << y << endl;
   return 0;
}
 long add(long x, long y)
{
   long sum;
   sum = x + y;
   return sum;
}
 float add(float x, float y)
{
  float sum;
   sum = x + y;
```

```
    return sum;
}
```

EXCEPTION HANDLING

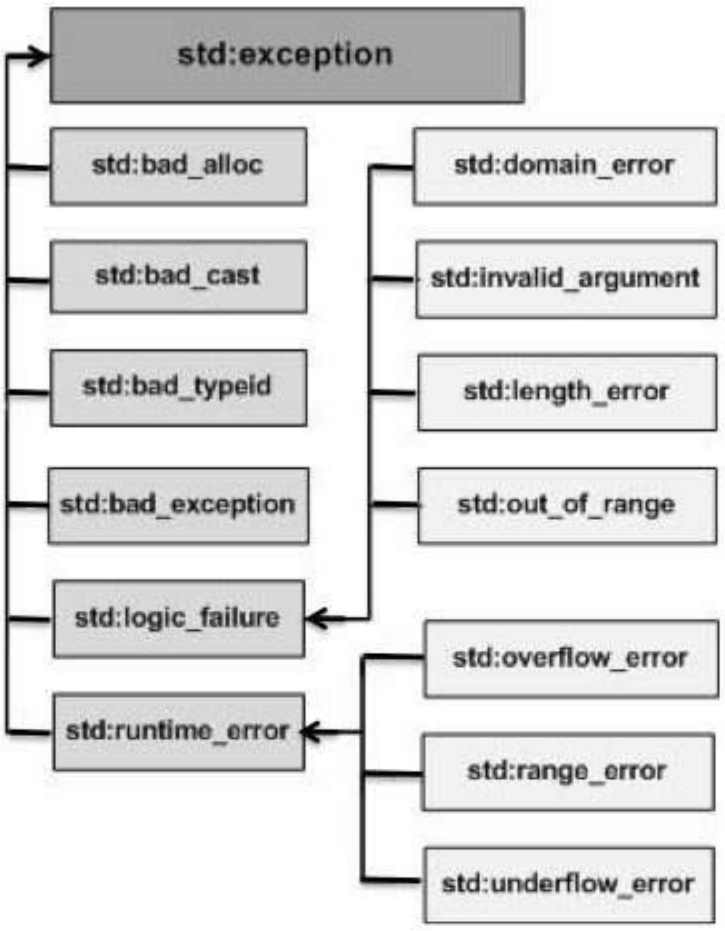
Introduction:

- Usually there are mainly two type of bugs, logical errors and syntactic errors.
- The logical errors occur due to poor understanding of problem and syntactic errors arise due to poor understanding of language.
- There are some other problems called exceptions that are run time anomalies or unused conditions that a program may encounter while executing.
- These anomalies can be division by zero,access to an array outside of its bounds or running out of memory or disk space.
- When a program encounters an exceptional condition it is important to identify it and dealt with it effectively.
- An exception is an object that is sent from the part of the program where an error occurs to that part of program which is going to control the error.
- An exception occurs when an unexpected error or unpredictable behaviors happened on your program not caused by the operating system itself. These exceptions are handled by code which is outside the normal flow of control and it needs an emergency exit. C++ has incorporated three operators to help us handle these situations: try, throw and catch.
- C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below:

Exception	**Description**
std::exception	An exception and parent class of all the standard C++ exceptions.
std::bad_alloc	This can be thrown by **new**.
std::bad_cast	This can be thrown by **dynamic_cast**.
std::bad_exception	This is useful device to handle unexpected exceptions in a C++ program
std::bad_typeid	This can be thrown by **typeid**.
std::logic_error	An exception that theoretically can be detected by reading the code.
std::domain_error	This is an exception thrown when a mathematically invalid domain is used
std::invalid_argument	This is thrown due to invalid arguments.
std::length_error	This is thrown when a too big std::string is created
std::out_of_range	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This is occured when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

- The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions.
- It is called `std::exception` and is defined in the `<exception>` header.
- This class has a virtual member function called `what` that returns a null-terminated

character sequence (of type `char *`) and that can be overwritten in derived classes to contain some sort of description of the exception.



- Other way is to create your own class without inheriting exception class and write your own methods to handle exception as below :

```
#include <iostream>
using namespace std;
class Error
{
   public:
      Error(string s)
      {
         str=s;
      }
      void err_disp()
      {
         cout<<"Error message: "<<str<<endl;
      }
   private:
      string str;
};
int main() {

         cout<<"Enter a number between 1 to 10";
```

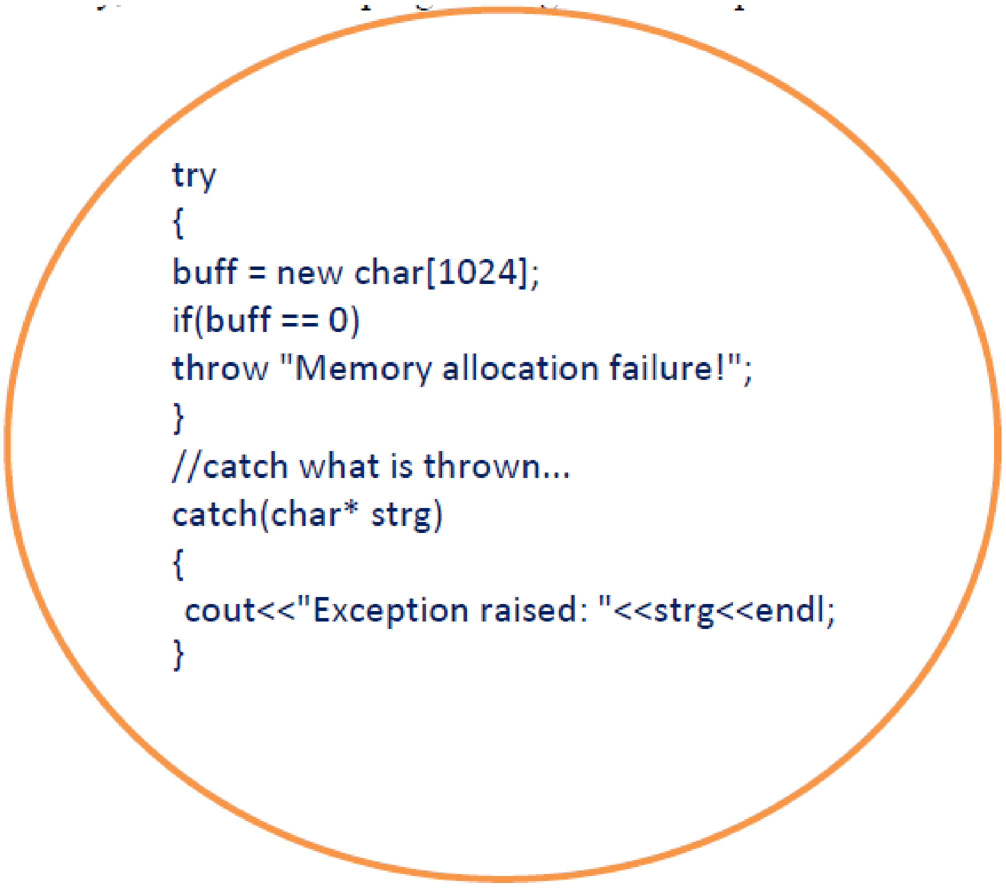
```
        int num;
        cin>>num;
        try
        {
           if(num<1 || num>10)
              throw Error("Value is not between 1 to 10");
          else
              cout<<"You entered "<<num<<endl;

        }

        catch(Error e)
        {
           e.err_disp();
        }
        return 0;
}
```

Principles of Exception handling:-

- Exceptions are basically of two types namely, synchronous and asynchronous exceptions.
- Errors such as "out of range index" and "over flow" belongs to synchronous type exceptions.
- The errors that are caused by the events beyond the control of program(such as keyboard interrupts) are called asynchronous exceptions.
- The purpose of exception handling mechanism is to detect and report an exceptional circumstances so that appropriate action can be taken.
- The mechanism for exception handling is
    - Find the problem(hit the exception).
    - Inform that an error has occurred(throw the exception).
    - Receive the error information(Catch the exception).
    - Take corrective actions(Handle the exception).
- The error handling code mainly consist of two segments, one to detect error and throw exceptions and other to catch the exceptions and to take appropriate actions.
- C++ has incorporated three operators to help us handle these situations: try, throw and catch.
- Syntax:

```
try
{
Compound-statement handler-list
handler-list here
The throw-expression:
throw expression
}
catch (exception-declaration) compound-statement
{
Exception-declaration:
type-specifier-list here
}
```

- The following is the try, throw…catch program segment example:



```
try
{
buff = new char[1024];
if(buff == 0)
throw "Memory allocation failure!";
}
//catch what is thrown...
catch(char* strg)
{
 cout<<"Exception raised: "<<strg<<endl;
}
```

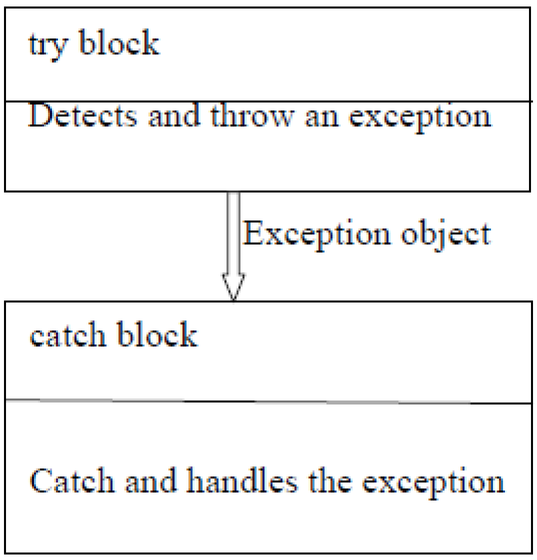
Exception handling mechanism:

- 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 statement is called try block.
- When an exception is detected it is thrown using 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.
- The general form for this is

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

```
            ………………
            …………….
    }
    ………….
    …………..
```

- When the try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block.
- If the type of object thrown matches the arg type in the catch statement,then the catch block is executed for handling the exception.
- If they donot match,the program is aborted with the help of abort() function which is executed implicitly by the compiler.
- When no exception is detected and thrown,the control goes to the statement immediately after the catch block i.e catch block is skipped.
- The below diagram will show the mechanism of exception handling


try block
Detects and throw an exception
Exception object
catch block
Catch and handles the exception

The following program shows the use of try-catch blocks.

```
#include<iostream>
using namespace std;
int main()
{
int a,b;
cout<<"enter the values of a and b";
cin>>a;
cin>>b;
int x = a- b;
try
{
if(x!=0)
{
cout<<"result(a/x) = "<<a/x<<"\n";
}
else
{
throw(x);
}
}
catch(int i)
{
cout<<"exception caught : x = "<<x<<"\n";
}
cout<<"end";
return 0;
}
```

Output:

```
enter value of a and b
20 15
result(a/x)=4
end
Second run
Enter value 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 statement is skipped and execution resumes with the first line after the catch.
- In the second run the denominator x become zero and therefore a division by zero situation occurs.
- This exception is thrown using the object x.
- Since the exception object is of integer type, the catch statement containing int type argument catches the exception and displays necessary message.
- The exceptions are thrown by functions that are invoked from within the try block.
- The point at which the throw is executed is called throw point.
- Once an exception is thrown to catch block ,control cannot return to the throw point.
- The general format of code for this kind of relationship is shown below

```
type function (arg list) //function with exception
{ ………..
……………..
    throw(object); //throw exception
            ………..
            ………..
    }
    ………
    ………….
    try
    { ……….
    ………. Invoke function here
    ………..
    }
    catch(type arg) //catches exception
    {
            …………..
```

```
                …………. Handle exception here
                ………….
        }
        ………..
```

- It is to be noted here that the try block is immediately followed by the catch block irrespective of the location of the throw point.
- The below program demonstrates how a try block invokes a function that generates an exception

```
//Throw point outside the try block
# include <iostream>
using namespace std;
void divide (int x,int y,int z)
{
cout<<"we are outside the function";
if ( ( x-y) != 0)
{ int r=z/(x-y);
cout<<"result = "<<r;
}
else
{
throw(x-y);
}
}
int main()
{
try
{
cout<<"we are inside the try block";
divide(10,20,30);
divide(10,10,20);
}
catch (int i)
{
cout<<"caught the exception";
}
return 0;
}
```

The output of the above program is

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

Throwing mechanism:-

- When an exception is encountered it is thrown using the throw statement in the following form:
  throw (exception);
  throw exception;

throw;

- The operand object exception may be of any type including constants.
- It is also possible to throw objects not intended for error handling.
- When an exception is thrown, it will be caught by the catch statement associated with the try block.
- In other words the control exits the try block and transferred to catch block after the try block.
- Throw point can be in the deep nested scope within the try block or in a deeply nested function call.

Catching mechanism:

- Code for handling exceptions is included in catch blocks.
- The catch block is like a function definition and is of form

```
catch(type arg)
{
        statements for managing exceptions
}
```

- The type indicates the type of exception that catch block handles.
- The parameter arg is an optional parameter name.
- The catch statement catches an exception whose type matches with the type of catch argument.
- When it is caught, the code in the catch block is executed.
- After executing the handler, the control goes to the statement immediately following in catch block.
- Due to mismatch ,if an exception is not caught abnormal program termination will occur.
- In other words catch block is simply skipped if the catch statement does not catch an exception.

Multiple Catch Statements:

- In some situations the program segment has more than one condition to throw an exception.
- In such case more than one catch blocks can be associated with a try block as shown below

```
try
{
        //try block
}
catch(type1 arg)
{
        //catch block1
}
catch(type 2 arg)
{
        //catch block 2
}
        ……………..
        …………….
catch (type N arg)
{
        //catch block N
```

}

- 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.
- If in some case the arguments of several catch statements match the type of an exception, then the first handler that matches the exception type is executed.
- The below program shows the example of multiple catch statements.

MULTIPLE CATCH STATEMENTS

```
#include <iostream>
using namespace std;
void test (int x)
{
try
{
if (x==1) throw x; //int
else
if(x==0) throw ‘x’; //char
else
if (x== -1 ) throw 1.0; //double
cout<<”end of try- block \n”;
}
catch(char c) //Catch 1
{
cout<<”Caught a character \n”;
}
catch (int m) //Catch 2
{ cout <<”caught an integer\n”;
}
catch (double d) //catch 3
{ cout<<”caught a double \n”;
}
cout<<”end of try –catch system \n\n”;
}
int main()
{
cout<<”Testing multiple catches \n”;
cout<<”x== 1 \n”;
test(1);
cout<<”x== 0 \n”;
test(0);
cout<<”x == -1 \n”;
test (-1);
cout <<”x== 2 \n”;
test (2);
return 0;
}
```

- The program when executed first invokes the function test() with x=1 and throws x an int exception.

- This matches the type of parameter m in catch 2 and therefore catch2 handler is executed.
- Immediately after the execution , the function throws ‘x’, a character type exception and therefore the first handler is executed.
- Finally the handler catch3 is executed when a double type exception is thrown.Every time only the handler which catches the exception is executed and all other handlers are bypassed.

Catch All Exceptions:

- In some cases when all possible type of exceptions can not be anticipated and may not be able to design independent catch handlers to catch them, in such situations a single catch statement is forced to catch all exceptions instead of certain type alone.
- This can be achieved by defining the catch statement using ellipses as follows

```
catch(. . .)
{
        //statement for processing all exceptions
}
```

- The below program illustrate the functioning of catch(…)

CATCHING ALL EXCEPTIONS

```
#include <iostream>
using namespace std;
void test(int x)
{
try
{
if (x== 0) throw x; //int
if ( x== -1) throw ‘x’; //char
if ( x== 1) throw 1.0; //float
}
catch(. . .) //catch all
{
cout<<”caught an exception \n”;
}
}
int main()
{
cout<<”testing generic catch\n”;
test(-1);
test(0);
test(1);
return 0;
}
```

- We can use the catch(. . .) as a default statement along with other catch handlers so that it can catch all those exceptions that are not handled explicitly.

Rethrowing an Exception:

- A handler may decide to rethrow an exception caught without processing them.
- In such situations we can simply invoke throw without any argument like throw;

- This cause 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.
- The following program shows how an exception is rethrown and caught.

RETHROWING AN EXCEPTION

```
#include <iostream>
using namespace std;
void divide(double x, double y)
{
cout<<”Inside Function \n”;
try
{ if (y== 0.0)
throw y; //throwing double
else
cout<<”division = “<< x/y<<”\n”;
}
catch(double) //Catch a double
{
cout<<”Caught double inside a function \n”;
throw; //rethrowing double
}
cout<<”end of function\n\n”;
}
int main()
{
cout <<”inside main \n”;
try
{ divide(10.5,2.0);
divide(20.0,0.0);
}
catch (double)
{ cout <<”caught double inside main \n”;
}
cout <<”End of mai\n “;
return 0;
}
```

- When an exception is rethrown, it will not be caught by the same catch statement or any other catch in that group.
- It will be caught by the an appropriate catch in the outer try/catch sequence for processing.

Specifying Exceptions:

- In some cases it may be possible to restrict a function to throw only certain specified exceptions.
- This is achieved by adding a throw list clause to function definition. The general form of using an exception specification is:

  Type function (arg-list) throw (type-list)

  ```
          {
          …………..
          ………… function body
  ```

```
                ……….
        }
```

- The type list specifies the type of exceptions that may be thrown.
- Throwing any other type of exception will cause abnormal program termination.
- To prevent a function from throwing any exception, it can be done by making the type list empty like throw(); //empty list in the function header line.
- The following program will show this

TESTING THROW RESTRICTIONS

```
#include <iostream>
using namespace std;
void test (int x) throw (int,double)
{
if (x== 0) throw 'x'; //char
else
if (x== 1) throw x; //int
else
if (x== -1) throw 1.0; //double
cout <<"End of function block \n ";
}
int main()
{
try
{
cout<<"testing throw restrictions\n";
cout<<"x== 0\n ";
test (0);
cout<<"x==1 \n";
test(1);
cout<<"x== -1 \n";
test(-1);
cout <<"x== 2 \n";
test(2);
}
catch( char c)
{
cout <<"caught a character \n";
}
catch(int m)
{
cout<<"caught an integer \n";
}
catch (double d)
{
cout<<"caught a double \n";
}
cout<<" end of try catch system \n \n";
return 0; }
```