Unit 7: Function Templates and Exception Handling [4hrs]

Template

- Template is a new concept which enable us to define generic class es and functions and thus provides support for generic programming.
- Generic programming is 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.
- Since the 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 classes or function.

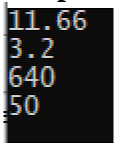
Format:

```
        template<class T>
        class  class_name
        {
                    ………………..//class member
                    ………………..// specifications with
                    ………………..// anonymous type T
                    ………………..// where or appropriate
        };
```

Example:

```
#include <iostream>
using namespace std;
template<class T1>
class Test
{
        T1 a;
        public:
        void add (T1 x, T1 y)
{
        a=x+y;
}
        void mul(T1 x, T1 y)
{
        a = x * y;
}
        void div(T1 x, T1 y)
{
        a = x/y;
}
        void sub(T1 x, T1 y)
{
        a = x -y;
}
        void show()
{
        cout<<a<<"\n";
}
};
int main()
{
        Test <float>  testf;
        Test <int> testi;
        testf.add(5.23,6.43);testf.show();
        testf.div(6.4,2.0);testf.show();
        testi.mul(20,32); testi.show();
        testi.sub(200,150); testi.show();
return 0;
}
```

Output



```
11.66
3.2
640
50
```

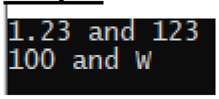
Class Template with multiple parameters:

```
template <class T1, class T2, ……..>
class  class_name
{
        ……………….
        ……………….
        ……………….          // Body of the class
        ……………….
};
```

Example

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

Output



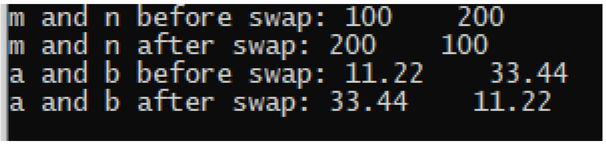
```
1.23 and 123
100 and W
```

Function Templates:

```
template<class T>
returntype functionname (arguments of type t)
{
        ………………………..
        ………………………..  // Body of function with type T
        ………………………..
        ………………………..  // Wherever appropriate
}
```

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

template <class T>
void swap1(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";
        swap1 (m,n);
        cout <<"m and n after swap: "<<m<<"    "<<n<<"\n";

        cout <<"a and b before swap: "<<a<<"    "<<b<<"\n";
        swap1(a,b);
        cout <<"a and b after swap: "<<a<<"    "<<b<<"\n";

}
```

```
int main()
{
        fun(100,200,11.22,33.44);
        return 0;
}
```



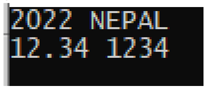
```
m and n before swap: 100     200
m and n after swap: 200     100
a and b before swap: 11.22     33.44
a and b after swap: 33.44     11.22
```

Function Templates with multiple parameters:

```
Template<class T1, class T2, ………>
returntype functionname (arguments of types T1, T2….)
{
        ……………………..
        ……………………..  // Body of function with type T
        ……………………..
}
```

Example

```
#include<iostream>
#include<string.h>
using namespace std;
template<class T1, class T2>
void display (T1 x, T2 y)
{
        cout<< x << " " <<y <<"\n";
}
int main ()
{
        display (2022, "NEPAL");
        display (12.34, 1234);
        return 0;
}
```



```
2022 NEPAL
12.34 1234
```

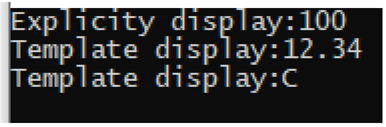
Overloading of template functions:

A template function may be overloaded either by template function 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 that could be created with an exact match.
3. Try normal overloading resolution to ordinary functions and call the one that matches.

Example:

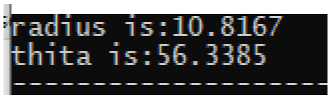
```
#include<iostream>
#include<string.h>
using namespace std;
template<class T>
        void display (T x)
        {
                    cout<<”Template display:”<<x<<”\n”;
        }
        void display (int x)
        {
                    cout<<”Explicity display:”<<x<<”\n”;
        }
int main()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



```
Explicity display:100
Template display:12.34
Template display:C
```

Type conversion using Template:

```
/*----------Rectangle to polar using Template and one class to another class type conversion using
the concept of Template--------------*/
#include<iostream>
#include<math.h>
using namespace std;
template<class T>
class rectangle
{          T  x;
                    T  y;
                    public:
                    rectangle(T a,T b)
                    {          x=a;
                               y=b;
                     }
                     T get_x()
                     {  return(x);
                     }
                     T get_y()
                     { return(y);
                     }
};
template <class T1>
class polar
{
                     T1 radius;
                     T1 thita;
                     public:
                     polar(){ }
                     polar(rectangle  <float>  r)
                     {   T1 tempx=r.get_x();
                         T1 tempy=r.get_y();
                         radius = sqrt(tempx*tempx + tempy*tempy);
                        thita = atan(tempy/tempx);
          }

            void  show()
            {          cout<<"radius is:"<<radius<<endl;
                    cout<<"thita is:"<<thita*(180/3.14);
            }
};
            int main()
            {
            rectangle  <float>   r(6.0,9.0);
            polar  <float>  p(r);
            p.show();

            return 0;
            }
```



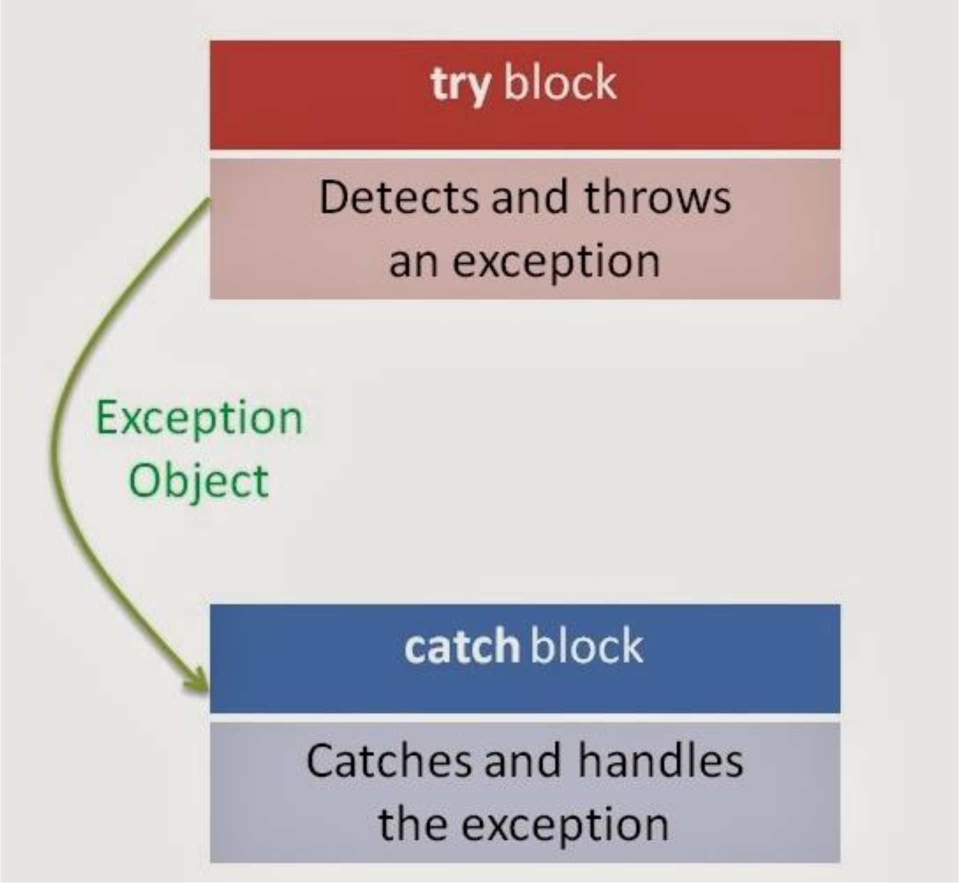
```
radius is:10.8167
thita is:56.3385
--------------------
```

Exception Handling

- The two most common types of bugs are logic errors and syntactic errors
- The logic errors occurs due to poor understanding of the problem and solution procedure
- The syntactic error occurs due to poor understanding of the language itself
- We often come across with some peculiar problems other than logic or syntax errors. They are known as exceptions.
- Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing
- Anomalies might include conditions such as division by zero, access to an array outside of its bounds or running out of memory or disk space.
- Exception handling is a new feature added to ANSI C++

Basics of Exception Handling

- The purpose of exception handling mechanism is to provide means to detect and report an "exceptional Circumstances" so that appropriate actions can be taken.
- This mechanism suggests 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)
- " try "
    - The keyword try is used to preface a block of statements which may generate exceptions.
- " throw "
    - When an exception is detected, it is thrown using a throw statement in the try block
- " catch "
    - " catch " catches the exceptions thrown by the throw statement in the try block.


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

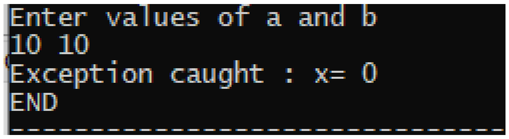
Note: The catch block that catches the exceptions must immediately follow the try block that throws the exception.

General form:

```
--------------
--------------
try {
        ---------
        ---------
        Throw exception; // block of statements which detects and throw an exception
        ----------
        ----------
}
catch(type argument)   // catches exception
{
        ----------
        -----------     //Block of statements that handles the exception
 }
---------
----------
```

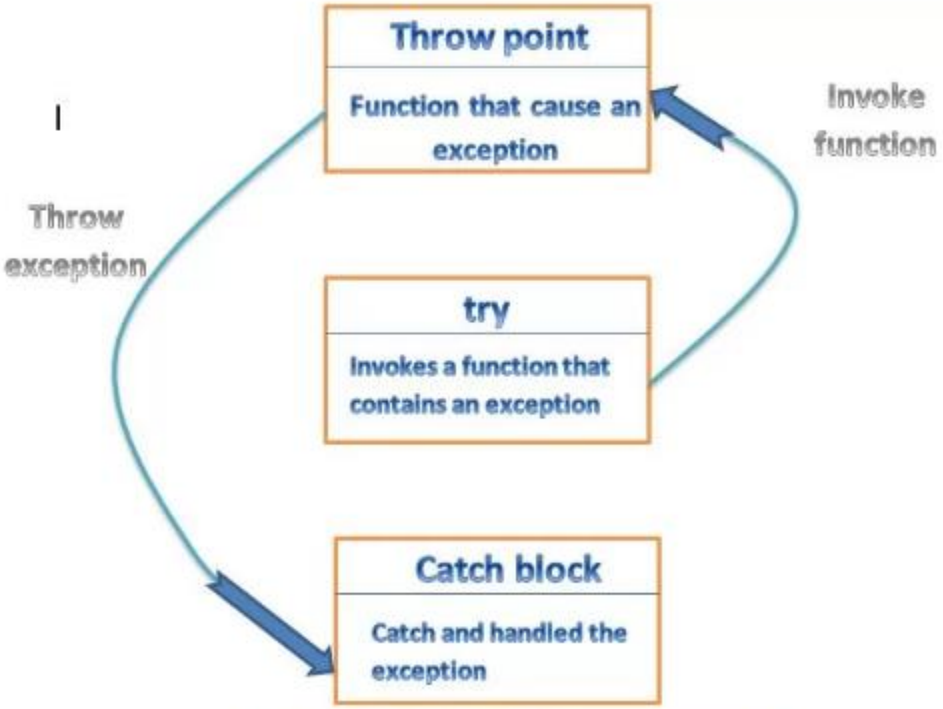
Example

```
#include<iostream>
using namespace std;
int main()
{

        int a,b;
        cout<<"Enter values of a and b\n";
        cin>>a>>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;
}
```



```
Enter values of a and b
10 10
Exception caught : x= 0
END
-------------------------------
```

<u>Throw Point outside " try " block</u>


Throw point
Function that cause an exception
Invoke function
Throw exception
try
Invokes a function that contains an exception
Catch block
Catch and handled the exception

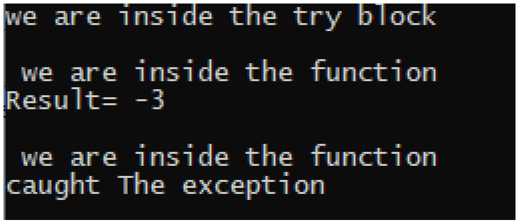
Function invoke by try block throwing exception

```
type function(arg list) // function with exception
{
…….
…….
throw (object); // throws exception
…….
…….
```

```
}
try
{

…..
….. invoke function here
…..
}
catch (type arg) // catches exception
{
…….
……. Handles exception here
…….
```

Note: The try block is immediately followed by the catch block, irrespective of the location of the throw point. In the below program show 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 << "\n we 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)
{
cout << "caught The exception \n";
}
}
```



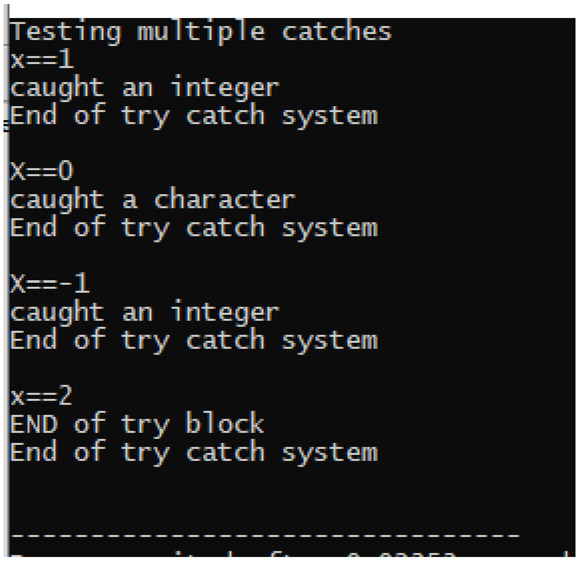
```
we are inside the try block

 we are inside the function
Result= -3

 we are inside the function
caught The exception
```

Multiple Catch Statement

- It is possible that a program segment has more than one condition to throw an exception
- In such cases, we can associate more than one catch statement with a ‘ try ‘.

Example:

```
#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; // 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);
}
```



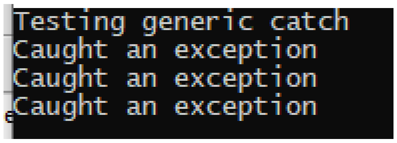
```
Testing multiple catches
x==1
caught an integer
End of try catch system

X==0
caught a character
End of try catch system

X==-1
caught an integer
End of try catch system

x==2
END of try block
End of try catch system


--------------------------------
```

Catch All Exceptions

- Catch Catches all exceptions, irrespective of their type.

Example

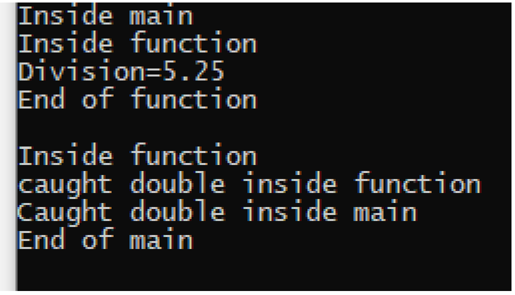
```
#include <iostream>
using namespace std;
void test(int x)
{
try
{
if (x==0) throw x;
if (x==-1) throw 'x';
if (x==1) throw 1.0;
}
catch (...)
{
cout << "Caught an exception \n";
}
}
int main()
{
cout << "Testing generic catch \n";
test(-1);
test(0);
test(1);
return 0;
}
```



```
Testing generic catch
Caught an exception
Caught an exception
Caught an exception
```

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 function \n";
throw ; // re-throwing 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 main \n";
return 0;
}
```



```
Inside main
Inside function
Division=5.25
End of function

Inside function
caught double inside function
Caught double inside main
End of main
```