

# UNIT 7

# FUNCTION TEMPLATES AND EXCEPTION HANDLING

LH – 4HRS

PRESENTED BY:

**ER. SHARAT MAHARJAN**

OOP IN C++

# CONTENTS (LH – 4HRS)

7.1 Function templates,

7.2 Function templates with multiple arguments,

7.3 Class templates,

7.4 Templates and inheritance,

7.5 Exceptional Handling (try, throw and catch),

7.6 Multiple exceptions, exceptions with arguments,

7.7 Use of exceptional handling

# 7.1 Function templates

- A template is a simple and yet very powerful tool in C++. The simple idea is to **pass data type as a parameter so that we don't need to write the same code for different data types**. For example, a software company may need `sort()` for different data types. Rather than writing and maintaining the multiple codes, we can write one `sort()` and pass data type as a parameter.
- C++ adds two new keywords to support templates:
  - ***'template'*** and ***'typename'***. The second keyword can always be replaced by keyword ***'class'***.

- A template function may be defined as an unbounded function in which all the possible parameters to the function are not known in advance and a copy of the function has to be created when necessary.
- Template functions are defined by using the keyword, **template**.
- Templates are blueprints of a function that can be applied to different data types.

### Syntax of Template:

```
template <class type1, type2...>  
void function-name(type1 parameter1,type2 parameter2){  
    //function body}
```

### Example:

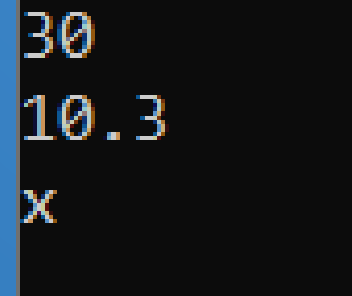
```
Template <class X>
```

```
X min(X a, X b){return (a>b)?a:b;}
```

## LAB 1: Finding maximum of two values by using template function

```
#include<iostream>
using namespace std;
template <class T> //function with one parameter
T myMax(T x, T y){
    return (x>y)?x:y;
}
int main(){
    cout<<myMax(30,20)<<endl;
    cout<<myMax(2.5,10.3)<<endl;
    cout<<myMax('c','x')<<endl;
    return 0;
}
```

### OUTPUT:

A screenshot of a terminal window with a black background and yellow text. It displays the output of the C++ program: 30, 10.3, and x, each on a new line.

```
30
10.3
x
```

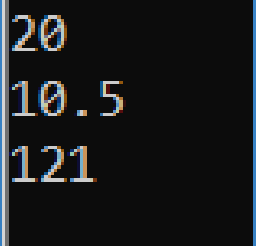
# 7.2 Function templates with multiple arguments

```
// 2 different data_type parameters:  
template<class T1, class T2>  
void someFunc(T1 var1, T2 var2 ){  
    // some code in here...}
```

## LAB 2: Finding maximum of two values of different types by using template function.

```
#include<iostream>  
using namespace std;  
template <class T,class U>  
T myMax(T x, U y){  
    return (x>y)?x:y;}  
int main(){  
    cout<<myMax(10.5,20)<<endl;  
    cout<<myMax<double>(10.5,5)<<endl;  
    cout<<myMax(100,'y')<<endl;  
    return 0;}
```

### OUTPUT:



```
20  
10.5  
121
```

## 7.3 Class templates

- Like function templates, class templates are useful when a class defines something that is **independent of the data type**.
- Following is an example of class template:

### LAB 3: Class template to find larger of two numbers

```
#include<iostream>
using namespace std;
template <class T>
class myPair{
    private:
        T a,b;
    public:
        myPair(T x,T y){
            a=x;
            b=y;
        }
        T getMax(){
            return (a>b)?a:b;
        }
};

int main(){
    myPair<int> object1(10,20);
    cout<<"Larger="<<object1.getMax()<<endl;
    myPair<double> object2(20.5,10.5);
    cout<<"Larger="<<object2.getMax()<<endl;
    return 0;
}
```

#### OUTPUT:

```
Larger=20
Larger=20.5
```



## 7.4 Templates and inheritance

- It is possible to inherit from a template class.
- All the usual rules for inheritance and polymorphism apply.
- If we want the new, derived class to be generic it should also be a template class; and pass its template parameter along to the base class.
- Following is an example of inheritance in template class:

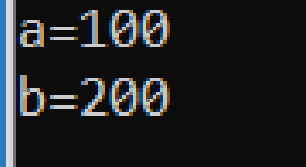
#### LAB 4: program to demonstrate inheritance in template classes

```
#include<iostream>
using namespace std;
template <class T>
class Base{
    private:
        T a;
    public:
        void setData(T p){
            a=p;}
        void getData(){
            cout<<"a="<<a<<endl;}};

template<class T>
class Derived: public Base<T>{
    private:
        int b;
    public:
        void setData(T m, T n){
            Base<T>::setData(m);
            b=n;}
        void getData(){
            Base<T>::getData();
            cout<<"b="<<b<<endl;}};

int main(){
    Derived<int> x;
    x.setData(100,200);
    x.getData();
    return 0;}
```

#### OUTPUT:



```
a=100
b=200
```

# 7.5 Exceptional Handling (try, throw and catch) + 7.7 Use of Exceptional Handling

- One of the advantages of C++ over C is Exception Handling. Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution.
- C++ provides following specialized keywords for this purpose.
  1. ***try***: represents a block of code that can throw an exception.
  2. ***catch***: represents a block of code that is executed when a particular exception is thrown.
  3. ***throw***: Used to throw an exception.

Following are main advantages of exception handling over traditional error handling.

**1) Separation of Error Handling code from Normal Code:** In traditional error handling codes, there are always **if else** conditions to handle errors. These conditions and the code to handle errors get **mixed up with the normal flow**. This makes the **code less readable and maintainable**. With try catch blocks, **the code for error handling becomes separate from the normal flow**.

**2) Functions/Methods can handle any exceptions they choose:** In C++, a function can specify the exceptions that it throws using the throw keyword.

**3) Grouping of Error Types:** In C++, both basic types and objects can be thrown as exception.

## C++ Exceptions:

- When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.
- When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (throw an error).

## C++ try and catch:

Exception handling in C++ consists of three keywords: **try**, **throw** and **catch**:

- The **try statement** allows us to define a block of code to be tested for errors while it is being executed.
- The **throw keyword** throws an exception when a problem is detected, which lets us create a custom error.
- The **catch statement** allows us to define a block of code to be executed, if an error occurs in the try block.

The **try and catch keywords** come in pairs:

- We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.
- In the catch block, we catch the error and do something about it. The catch statement takes a parameter: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.
- If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped.

## LAB 5: try block throwing exception

```
#include<iostream>
using namespace std;
int main(){
    float a,b;
    cout<<"Enter two numbers:"<<endl;
    cin>>a>>b;
    try{
        if(b==0)
            throw b;
        else
            cout<<"Result="<<a/b<<endl;
    }
    catch(float x){
        cout<<"Divide by zero exception."<<endl;
    }
    cout<<"END"<<endl;
    return 0;
}
```

### OUTPUT1:

```
Enter two numbers:
10
2
Result=5
END
```

### OUTPUT2:

```
Enter two numbers:
10
0
Divide by zero exception.
END
```

## 7.6 Multiple exceptions, exceptions with arguments

- 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 as shown below:

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

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

## LAB 6: program using multiple catch blocks for a single try block

```
#include<iostream>
#include<conio.h>
using namespace std;
void test(int x) {
    try {
        if (x > 0)
            throw x;
        else
            throw 'x';
    } catch (int x) {
        cout << "Catch a integer and that integer is:" << x<<endl;
    } catch (char x) {
        cout << "Catch a character and that character is:" << x<<endl;
    }
}
int main() {
    cout << "Testing multiple catches:"<<endl;
    test(10);
    test(0);
    getch();
    return 0;
}
```

### OUTPUT:

```
Testing multiple catches:
Catch a integer and that integer is:10
Catch a character and that character is:x
_
```



## Catching all exceptions:

- We can also define a **catch** block that captures all the exceptions independently of the type used in the call to **throw**.
- For that we have to write three points instead of the parameter type and name accepted by **catch**:

```
try{  
    //code here  
}  
catch(...){  
    cout<<"Exception occurred";  
}
```

## LAB 7: Catching all the exceptions

```

#include<iostream>
#include<conio.h>
using namespace std;
int main() {
    double x,y,result;
    char opr;
    cout<<"To proceed enter two numbers:"<<endl;
    try{
        cout<<"First number:"<<endl;
        cin>>x;
        cout<<"Operator:"<<endl;
        cin>>opr;
        cout<<"Second number:"<<endl;
        cin>>y;
        if(opr!='+'&&opr!='-'&&opr!='*'&&opr!='/')          //make sure the user typed a valid operator
            throw opr;
        if(y==0)      //find out if the denominator is 0
            throw y;
        switch(opr){ //perform an operation based on the user's choice
            case '+':
                result=x+y;
                break;
            case '-':
                result=x-y;
                break;
            case '*':
                result=x*y;
                break;
            case '/':
                result=x/y;
                break;
        }
        cout<<"Result="<<result<<endl;          //display the result of the operation
    }
    catch(...){          //catch all exception
        cout<<"Exception occurred!"<<endl;
    }
    getch();
    return 0;}

```

### OUTPUT 1:

```

To proceed enter two numbers:
First number:
10
Operator:
+
Second number:
20
Result=30

```

### OUTPUT 2:

```

To proceed enter two numbers:
First number:
10
Operator:
&
Second number:
20
Exception occurred!

```

### OUTPUT 3:

```

To proceed enter two numbers:
First number:
10
Operator:
/
Second number:
0
Exception occurred!

```

## Standard Exceptions in C++

- There are some standard exceptions in C++ under <exception> which we can use in our programs. They are arranged in a parent-child class hierarchy which is depicted below:
- **std::exception** - Parent class of all the standard C++ exceptions.
- **logic\_error** - Exception happens in the internal logical of a program.
  - **domain\_error** - Exception due to use of invalid domain.
  - **invalid\_argument** - Exception due to invalid argument.
  - **out\_of\_range** - Exception due to out of range i.e. size requirement exceeds allocation.
  - **length\_error** - Exception due to length error.
- **runtime\_error** - Exception happens during runtime.
  - **range\_error** - Exception due to range errors in internal computations.
  - **overflow\_error** - Exception due to arithmetic overflow errors.
  - **underflow\_error** - Exception due to arithmetic underflow errors
- **bad\_alloc** - Exception happens when memory allocation with new() fails.
- **bad\_cast** - Exception happens when dynamic cast fails.
- **bad\_exception** - Exception is specially designed to be listed in the dynamic-exception-specifier.
- **bad\_typeid** - Exception thrown by typeid.

THANK YOU FOR YOUR ATTENTION