AMBITION GURU

# Object-Oriented programming in C++ (CSIT 202)

II semester, BScCSIT

Compiled by Ankit Bhattarai
Ambition Guru College

# Syllabus

AMBITION GURU

| Unit | Contents | Hours | Remarks |
|---|---|---|---|
| 1. | Introduction to C++ and OOP | 3 | |
| 2. | Classes and objects | 7 | |
| 3. | Operator Overloading & Type Conversion | 7 | |
| 4. | Inheritance | 7 | |
| 5. | Polymorphism and Virtual Functions | 7 | |
| **6.** | **Templates and Exception Handling** | **7** | |
| 7. | I/O Stream | 8 | |

Practical Works

Credit hours : 3

AMBITION GURU

# Unit 6
# Templates and Exception Handling
(7 hrs.)

Function templates and class templates, use of templates, Exception handling basics: Try, throw, catch blocks, Multiple catch blocks and nested try, purpose of exceptional handling

# Templates

- If the same operation is to be performed with different data types, then we need *to write the same code(functions or classes) for the different data types.*
- For example, if we need to write functions for swap that support different data types then we need to make functions for all different types. Similarly, if we need to make a vector class that support different data types, then we need to make classes for each data type.
- But with template mechanism, we can make a single function and a single class and do the job by avoiding unnecessary repetition of source code.

# Templates

- It is the important feature of C++ which provides **great flexibility** to the language. It supports generic programming, allowing development of reusable software component with functions and classes, supporting different data types in a single framework.
- In other words,

A **template** in C++ is a feature that allows us to write **generic programs** – i.e., functions and classes that work with **any data type** without rewriting the same code for each type.

# Why Templates ?

AMBITION GURU

1. Code Reusability: We don’t need to write separate functions/classes for int, float, double, etc. One template works for all types.
2. Type Safety: Unlike macros (#define), templates check types at compile time → reduces errors.
3. Maintainability: Easy to maintain since only one function/class is written for multiple data types.
4. Flexibility: Can be used for user-defined types (objects), not just built-in types.
5. Efficiency: Compiler generates optimized code for the required data type (no runtime overhead).

## Types of templates

1) Function template.

2) Class template.

## Function template

- Template declared for the function is called function template.
- In function template a single function which is written once can be used for different data types.

### Syntax

```
template<class template_type>
return_type function_name(template_type argument)
{
        //Body of the function;
}
```

template is a keyword. The `template_type` written after the keyword class within the angular bracket specify the generic data type.

# Example program of function template

```
//Example program of function template
#include<iostream>
using namespace std;
template<class T>
T return_max(T a, T b)
{
T result;
if(a>b)
    result=a;
else
    result= b;
return result;
}
```

```
int main( )
{
        cout<<"Between two integer numbers maximum number="<<return_max(2,3)<<endl;
        cout<<"Between two double numbers maximum number="<<return_max(5.5,7.5);
        return 0;
}
```

**Output:**

Between two integer numbers maximum number=3
Between two double numbers maximum number=7.5

Example program to find the sum and average of the elements of the different types of array using function template.

```
#include<iostream>
using namespace std;
template<class T>
void sum_avg( T a[5])
{
     T sum=0;
     float avg;
     for(int i=0;i<5;i++)
     {
          sum=sum+a[i];
     }
     cout<<"Sum="<<sum;
     avg=float(sum)/5;
     cout<<" Average="<<avg<<endl;
}
```

```
int main( )
{
    int a1[5],i;
    double b1[5];
    cout<<"Enter the elements for integer type array:”;
    for(i=0;i<5;i++)
    {
        cin>>a1[i];
    }
    cout<<"Sum and Average of integer type array is:"<<endl;
    sum_avg(a1);
```

```
    cout<<"Enter the elements for double type array:"<<endl;
    for(i=0;i<5;i++)
    {
        cin>>b1[i];
    }
    cout<<"Sum and Average of double type array is:"<<endl;
    sum_avg(b1);
     return 0;
}
```

**Output:**

```
Enter the elements for integer type array:5
2
3
1
4
Sum and Average of integer type array is:
Sum=15 Average=1.5
Enter the elements for double type array:5.6
3.4
2.1
3.2
4.6
Sum and Average of double type array is:
Sum=18.9 Average=1.89
```

# Function template with multiple template parameters

## Syntax:

```
template<class template_type1,class  template_type2,…>

 return_type function_name(template_type1 argument,template_type2 argument…)

{

        //Body of the function.

}
```

**Example:**

```
#include<iostream>
using namespace std;
template<class T1,class T2>
void show_data(T1 a,T2 b)
{
    cout<<"("<<a<<" ,"<<b<<")"<<endl;
}
```

```
int main( )
{
     int inum=5;
     float fnum=5.5;
     char a[ ]="TEMPLATE";
     show_data(inum, fnum);
     show_data(inum, a);
     show_data(a, fnum);
     return 0;
}
```

**Output:**

(5 ,5.5)
(5 ,TEMPLATE)
(TEMPLATE ,5.5)

# Class template

# Class template:

AMBITION GURU

- Similar to the function template we can declare class template that operates on any type of data.
- A class that  operates on any type of data is called a class template.

## Syntax:

```
template<class template_type>
class class_name
{
private:
//data members
public:
//member function(s);
}
```

- Once a class is declared as template then the object of the template class can be created as:

```
class_name<data_type>object_name;
```

**OR**

```
class_name<data_type1,data_type2,……,data_typen>object_name;
```

(For the class using multiple template parameters)

## Declaring member function of the template class inside the class and defining outside of theclass definition.

AMBITION GURU

**Syntax:**

```
template<class
template_type>
class class_name
{
//……..;
public:
return_type function_name(template_type argument(s));
};
```

```
template<class template_type>
return_type class_name<template_type>::function_name(template_type argument(s))
{
      //Body of the function;
}
```

An example program to find the largest element of the array of the template class as well as defining the member function of the template class outside of the class definition.

```
#include<iostream>
using namespace std;
template<class T>
class Array{
    T a[5];
    int i;
    public:
    Array(T a1[5]){
        for(i=0;i<5;i++)
        {
            a[i]=a1[i];
        }
    }
```

```
void display( )
{
     int i;
     for(i=0;i<5;i++)
     {
          cout<<a[i]<<"\t";
     }
T largest( );
};
```

```
template<class T>
T Array<T>::largest( )
{
     int i;
     T L=a[0];
     for(i=1;i<5;i++)
     {
     if(a[i]>L)
          L=a[i];
     }
return L;
}
```

```
int main( )
{
   int temp1[ ]={1,2,3,4,5};
   double temp2[ ]={1.5,2.5,3.5,4.5,5.5};
   Array<int>obj1(temp1);
   cout<<"The elements of int type array are:"<<endl;
   obj1.display( );
   cout<<endl;
   cout<<"The largest element of the int type array is:"<<obj1.largest()<<endl;
   Array<double>obj2(temp2);
   cout<<"The elements of double type array are:"<<endl;
   obj2.display( );
   cout<<endl;
   cout<<"The largest element of the double type array is:"<<obj2.largest()<<endl;
   return 0;
}
```

**Output:**

```
The elements of int type array are:
1       2       3       4       5
The largest element of the int type array is:5
The elements of double type array are:
1.5     2.5     3.5     4.5     5.5
The largest element of the double type array is:5.5
```

# Default argument with class template

```
#include<iostream>
using namespace std;
template<class T=double>
class Array
{
     T a[5];
     public:
     void input( )
     {
          cout<<"Enter the elements to the array:"<<endl;
          for(int i=0;i<5;i++)
               cin>>a[i];
     }
     void display( );
};
```

```
template<class T>
void Array<T>::display( )
{
    int i;
    cout<<"The elements of the array are:"<<endl;
    for(i=0;i<5;i++)
    cout<<a[i]<<"\t";
}
```

```
int main( )
{
    Array<int>a;
    cout<<"Integer type array:"<<endl;
    a.input( );
    a.display( );
    cout<<endl;
    cout<<"Double type array:"<<endl;
    Array< >b;
    b.input( );
    b.display ( );
    return 0;
}
```

Output:

Integer type array:
Enter the elements to the array:
1
2
3
4
5
The elements of the array are:
1 2 3 4 5

Double type array:
Enter the elements to the array:
4.5
6.5
7.3
2.3
1.2
The elements of the array are:
4.5 6.5 7.3 2.3 1.2

# Derived class template

AMBITION GURU

- A base class can be a template class and if the derived class is non- template class, then *while creating the derived class we need to specify the template argument of the base class with the data type.*
- In this type we don't add any template extra to derived class.

```
//Example program:
#include<iostream>
using namespace std;
template<class T>
class base{
    protected:
    T a;
    public:
    base(T c){
        a=c;
    }
```

```
    void display( )
    {
        cout<<"The value of a="<<a<<endl;
    }
};

class derived: public base<int>
{
    protected:
    int d;
    public:
    derived(int x,int y):base<int>(x)
    {
        d=y;
    }
```

```
    void display( )
    {
        base<int>::display( );
        cout<<"The value of d="<<d<<endl;
    }
};

int main()
{
    derived d(20,30);
    d.display( );
    return 0;
}
```

**Output:**

The value of a=20

The value of d=30

- There may be situation where the derived class and base class is template class and while defining the derived class, the template argument to base class may be specified.

```
//Example program:
#include<iostream>
using namespace std;
template<class T>
class base
{
    protected:
    T a;
    public:
    base(T c)
    {
        a=c;
    }
```

```
    void display( )
    {
        cout<<"The value of a="<<a<<endl;
    }
};

template<class T>
class derived: public base<int>
{
    protected:
    T d;
    public:
    derived(int x,T y):base<int>(x)
    {
        d=y;
    }
```

```
    void display( )
    {
        base<int>::display( );
        cout<<"The value of d="<<d<<endl;
    }
};

int main( )
{
    derived <double> d(20,30.25);
    d.display( );
    return 0;
}
```

**Output:**

The value of a=20

The value of d=30.25

- We can add extra template parameter in the derived class along with the base class template parameter.

```
#include<iostream>
using namespace std;
template<class T>
class base
{
    protected:
    T a;
    public:
    base( T c)
    {
        a=c;
    }
```

```
    void display( )
    {
        cout<<"The value of a="<<a<<endl;
    }
};

template<class T,class T1>
class derived: public base<T>
{
    protected:
    T1 d;
    public:
    derived(T x,T1 y):base<T>(x)
    {
        d=y;
    }
```

```
        void display( )
        {
                base<T>::display( );
                cout<<"The value of d="<<d<<endl;
        }
};

int main( )
{
        derived<double,char>d(10.25,'A');
        d.display( );
        return 0;
}
```

**Output:**

The value of a=10.25

The value of d=A

- A derived class can be a template class from the base class which is a non-template class. While creating derived class, only template parameter is added in the program.

```
#include<iostream>
using namespace std;
class base
{
    protected:
    int a;
    public:
    base(int c)
    {
        a=c;
    }
```

```
    void display( )
    {
        cout<<"The value of a="<<a<<endl;
    }
};

template<class T>
class derived: public base
{
    T d;
    public:
    derived(int x,T y):base(x)
    {
        d=y;
    }
```

```
    void display( )
    {
        base::display( );
        cout<<"The value of d="<<d<<endl;
    }
};

int main( )
{
    derived<char>d(10, 'A');
    d.display( );
    return 0;
}
```

**Output:**

The value of a=10

The value of d=A

# Exceptional Handling

# Exception Handling:

AMBITION GURU

- The most common types of bugs or errors are ***logic errors*** and ***syntax errors***.
- The logic errors occur due to the poor understanding of the problem and the solution procedure while the syntax errors occur due to the poor understanding of the language.
- Exceptions are the errors other than logic and syntax errors. They are the runtime anomalies or unusual condition that a program may encounter(face) while executing such as division by zero, access to an array outside of its bound, running out of memory or disk space etc.
- The error handling mechanism in C++ is called *exception handling*. The purpose of exception handling mechanism is to provide means(way) to detect an exceptional circumstances so that the appropriate action can be taken.

The exception handling mechanism should perform 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 action(*Handle* the exception).

## Exception handling constructs(try, throw, catch):

- C++ exception handling mechanism is built upon three keywords ***try***, ***throw*** and *catch.*
- **try keyword** is used to introduce a block of statements which may generate exception. This block of statements is known as try block.
- When the exception is detected, it is thrown by using a **throw statement** in the try block.
- A **catch block**(catch handler or exception handler) defined by the keyword catch catches the exception thrown by the throw statement in the try block and handles it appropriately.

Note: The catch block should always follow the try block i.e. try block and catch block should appear simultaneously in the program.

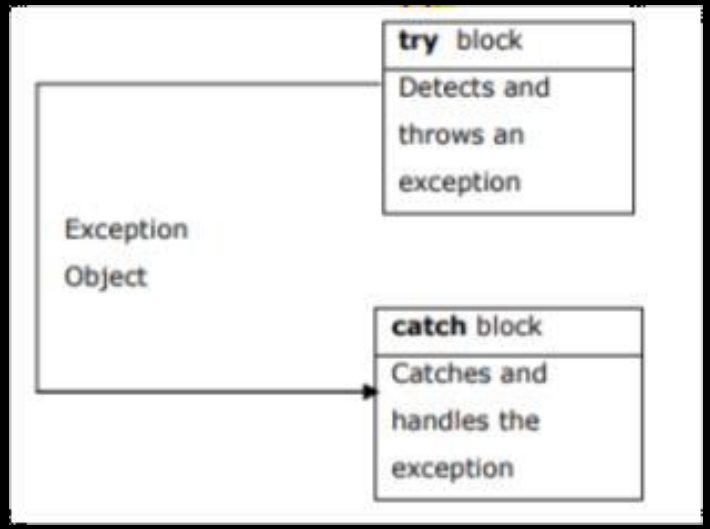


**Figure: Try block and catch block**

# An example program showing the exception handling

***//An example program showing the exception handling:***

AMBITION GURU

```
#include<iostream>
using namespace std;
void divide(int x, int y, int z)
{
    if((x-y)!=0)
    {
        int R=z/(x-y);
        cout<<"Result="<<R<<endl;
    }
    else
    {
        throw(x-y);
    }
}
```

```
int main( )
{
    int a,b,c;
    cout<<"Enter the three numbers:"<<endl;
    cin>>a>>b>>c;
    try
    {
        divide(a,b,c);
    }
    catch(int)
    {
        cout<<"Exception caught:"<<endl;
    }
    return 0;
}
```

**Output:**

Enter the three numbers:
5
10
15
Result=-3
Enter the three numbers:
5
5
10
Exception caught:

# Advantages of exception handling over conventional error handling

AMBITION GURU

- Traditionally, when the error is not handled locally ,the function could
  - terminate the program.
  - return a value that indicates error.
  - return some value and set the program in illegal state.
- Exception handling separates the error handling code from other code making the program more readable.
- Through the exception handling mechanism ,the program work even in the condition of error.
- The exception handling mechanism passes information about the error from the location where error occurs to the location where the error is handled.
- It solves the problem of communication of errors.

# Multiple Exception Handling

## Multiple Exception Handling:

- It is possible that program segment has more than one conditions to throw an exception. In such cases we can associate more than one catch *block* (catch handler or exception handler) with a *try* (much like the condition in a switch statement.)

## General form(Syntax):

```
try
{
        //try block;
}
catch(data_type1 argument)
{
        //catch block 1;
}
.
.
.
catch(data_type n argument)
{
        //catch block n;
}
```

- It is possible that arguments of several catch handler matches the type of an exception thrown. In such case, the first catch handler that matches the exception type is executed.

```
//Example program that shows handling multiple exceptions in C++
#include<iostream>
using namespace std;
void test( int b)
{
          cout<<"Inside function:"<<endl;
          try
          {
             if(b==0)
                    throw b;
```

```
     if(b==1)
           throw 1.0;
   if(b==2)
           throw 'R';
}
catch(int i)
{
           cout<<"Integer exception:"<<endl;
}
catch(double d)
{
           cout<<"Double exception:"<<endl;
}
```

```
        catch(char c)
        {
                cout<<"Character exception:"<<endl;
        }
        cout<<"End of the function:"<<endl;
}

int main( )
{
        test(0);
        test(1);
        test(2);
        test(3);
        cout<<"End of main function:";
        return 0;
}
```

**Output:**

```
Inside function:
Integer exception:
End of the function:
Inside function:
Double exception:
End of the function:
Inside function:
Character exception:
End of the function:
Inside function:
End of the function:
End of main function:
```

## Re-throwing an exception:

AMBITION GURU

- We can make the catch handler to rethrow the exception caught without processing it.
- In such situations we simply invoke ‘throw’ without any argument. This causes the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch handler listed after that enclosing try block.

```
//Example program:
#include<iostream>
using namespace std;
void divide(double x, double y)
{
        try
        {
           if(y==0.0)
                throw y;
```

```
        else
                cout<<"Division="<<x/y<<endl;
        }
        catch(double)
        {
                cout<<"Caught double inside the function:"<<endl;
                throw; //re-throwing an exception
        }
        cout<<"End of function:"<<endl;
}
```

```
int main()
{
        try
        {
                divide(10.5,2.0);
                divide(20.0,0.0);
        }
        catch(double)
        {
                cout<<"Caught double inside main function:"<<endl;
        }
        cout<<"End of main function:"<<endl;
        return 0;
}
```

**Output:**

Division=5.25
End of function:
Caught double inside the function:
Caught double inside main function:
End of main function:

# Catching all exceptions

## Catching all exceptions:

- In some situations, we may not be able to anticipate all possible types of exceptions and therefore we may not be able to design independent catch handlers to catch the exceptions.
- In such circumstances we can force a catch handler to catch all exceptions instead of a certain type alone.

***//Example program:***

```
#include<iostream>
using namespace std;
void test(int x)
{
        try
        {
        if(x==0)
                throw x;
        if(x==1)
                throw 1.0;
        if (x==2)
                throw 'A';
}
```

```
    catch(...) //catch all(parenthesis with three dots)
       {
          cout<<"Caught an exception:"<<endl;
       }
}

int main( )
{
        cout<<"Testing generic catch:"<<endl;
        test(0);
        test(1);
        test(2);
        return 0;
}
```

**Output:**

Testing generic catch:
Caught an exception:
Caught an exception:
Caught an exception:

## Exception with argument:

- The exception with argument helps the programmer to know what had value actually caused the exception by defining the object in the exception handler.
- That means adding data member to the exception class which can be retrieved by exception handler to know the cause.
- throw statement throws exception class object with some initial value which is used by exception handler.

```
//Example program:
#include<iostream>
using namespace std;
class time
{
     int hour, minute, second;
     public:
     class out_of_range //exception class
    {
      char *message;
      public:
      out_of_range(char *name )
      {
        message=name;
       }
```

```
        void display( )
        {
            cout<<message;
        }
    };


void input( )
{
   cout<<"Enter the values for hour, minute and second:"<<endl;
   cin>>hour>>minute>>second;
   if(hour>12 && minute>60 && second>60)
    {
      throw out_of_range("Exception! invalid values for hour, minute and second:");
    }
   if(hour>12 && minute>60)
    {
      throw out_of_range("Exception due to invalid values for hour and minute:");
    }
```

```
if(hour>12 && second>60)
  throw out_of_range("Exception due to invalid values for hour and second:");
if(minute>60 && second>60)
  throw out_of_range("Exception due to invalid values for minute and second:" );
if(hour>12)
  throw out_of_range("Exception due to invalid value for hour:");
if(minute>60)
  throw out_of_range("Exception due to invalid value for minute:");
if ( second>60)
  throw out_of_range("Exception due to invalid value for second:");
}
void display( )
{
   cout<<hour<<"Hour"<<minute<<"Minute and"<<second<<"Second"<<endl;
}
};
```

```
int main( )
{
  time t;
  try
 {
   t.input( );
   t.display( );
  }
 catch(time::out_of_range range)
 {
     range.display( );
  }
return 0;
}
```

**Run1:**

Enter the values for hour, minute and second:
1
65
75
Exception due to invalid values for minute and second:

**Run2:**

Enter the values for hour, minute and second:
1
45
55
1Hour45Minute and55Second

# Questions

AMBITION GURU

1. Write a C++ program to implement function template with multiple arguments.
2. Write a function template that returns the greater number between two integers.
3. What is exception handling? What are the tools used for exception handling ? Explain with programming example.
4. How can you define catch statement that can catch any type of exception? Illustrate the use of multiple catch statement with example.
5. When class templates are useful? How can you define a class that can implement stack with integer as well as sack of strings? Illustrate with example.
6. What is exception? Why exception handling is better to use? Explain exception handling with try….. catch by using suitable example.

# THANK YOU
# Any Queries ?