Object Oriented Programming in C++

Segment-7

Course Code: 2301

Prepared by Sumaiya Deen Muhammad

Lecturer, CSE, IIUC

Contents

Template, Exception Handling and Standard Template Library:

- Generic functions,
- Generic classes,
- Exception handling,
- throwing mechanism,
- catching mechanism,
- Rethrowing mechanism,
- Specifying exceptions Templates,
- Components of STL,
- Container,
- Algorithms.

Templates in C++ Programming

Templates in C++ programming allows function or class to work on more than one data type at once without writing different codes for different data types. Templates are often used in larger programs for the purpose of code reusability and flexibility of program. The concept of templates can be used in two different ways:

- Function Templates
- Class Templates •

Templates in C++ Programming

Templates allow programmer to create a common class or function that can be used for a variety of data types. The parameters used during its definition is of generic type and can be replaced later by actual parameters. This is known as the concept of **generic programming**. The main advantage of using a template is the reuse of same algorithm for various data types, hence saving time from writing similar codes.

- For example, consider a situation where we have to sort a list of students according to their roll number and their percentage. Since, roll number is of integer type and percentage is of float type, we need to write separate sorting algorithm for this problem. But using template, we can define a generic data type for sorting which can be replaced later by integer and float data type.

Function Template

- A generic function that represents several functions performing same task but on different data types is called ***function template***.
- For example, a function to add two integer and float numbers requires two functions. One function accept integer types and the other accept float types as parameters even though the functionality is the same. Using a function template, a single function can be used to perform both additions. It avoids unnecessary repetition of code for doing same task on various data types.

Function Templates

- A function templates work in similar manner as function but with one key difference.
- A single function template can work on different types at once but, different functions are needed to perform identical task on different data types.
- If we need to perform identical operations on two or more types of data then, we can use function overloading. But better approach would be to use function templates because you can perform this task by writing less code and code is easier to maintain.

Why use Function Templates

- Templates are instantiated at compile-time with the source code.
- Templates are used less code than overloaded C++ functions.
- Templates are type safe.
- Templates allow user-defined specialization.
- Templates allow non-type parameters.

How to Define Function Template

- A function template starts with keyword ***template*** followed by template parameter/s inside <> which is followed by function declaration.

```
template <class T>
 T some_function(T argument)
{
    .... ... ....
}
```

- T is a template argumer
- We can also use keyword ***typename*** instead of ***class***.
- When, an argument is passed to **some_function( )**, compiler generates new version of **some_function()** to work on argument of that type.
- The templated type keyword specified can be either "**class**" or " **typename**":

```
template<class T>
template<typename T>
```

- Both are valid and behave exactly the same.

Example of Function Template

```
template <typename T>
T Sum(T n1, T n2)
{// Template function
    T rs;
    rs = n1 + n2;
    return rs;
}
int main()
{
    int A=10,B=20,C;
    long I=11,J=22,K;
    C = Sum(A,B);
// Calling template function
    cout<<"\nThe sum of integer values : "<<C;
    K = Sum(I,J);
// Calling template function
    cout<<"\nThe sum of long values : "<<K;
    return 0;
}
```

Example of Function Template

- Output:

```
The sum of integer values : 30
The sum of long values : 33
Process returned 0 (0x0)   execution time : 0.052 s
Press any key to continue.
```

Function Overloading vs Function Template

- Function Templates Example to show you function template use less code than function overloading:

```
template <typename T>
T square(T x)
{
    T result;
    result = x * x;
    return result;
}
int main()
{
    int i, ii;
    double d, dd;
    i = 2;
    d = 2.2;
    ii = square(i);
    cout << "Square of Integer Number "<< ": " << ii << endl;
    dd = square(d);
    cout<< "Square of double number "<< " : " << dd << endl;
    return 0;
}
```

Output:

```
Square of Integer Number : 4
Square of double number  : 4.84

Process returned 0 (0x0)   execution time : 0.048 s
Press any key to continue.
```

Function Overloading vs Function Template

```
#include <iostream>
using namespace std;
int square (int x)
{
    return x * x;
}
double square (double x)
{
    return x * x;
}
 int main()
 {
     int i, ii;
     double d, dd;
     i = 2;
     d = 2.2;
     ii = square(i);
     cout << "Square of Integer Number "<< " : " << ii << endl;
     dd = square(d);
     cout<< "Square of double number "<< " : " << dd << endl;
     return 0;
}
```

```
Square of Integer Number  : 4
Square of double number  : 4.84

Process returned 0 (0x0)   execution time : 0.489 s
Press any key to continue.
```

Home Work

- Write a generic function that swaps values in two variables. Your function should have two parameters of the same type. Test the function with int, double and string values.
- Write a C++ program using function templates to add two numbers of int and float data types?

Class Template

- Like function templates, we can also create class templates for generic class operations.
- Sometimes, we need a class implementation that is same for all classes, only the data types used are different.
- Normally, we would need to create a different class for each data type OR create different member variables and functions within a single class.
- This will unnecessarily bloat our code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.
- However, class templates make it easy to reuse the same code for all data types.

How to declare a Class Template?

```
template <class T>
class className
{
    ... .. ...
public:
    T var;
    T someOperation(T arg);
    ... .. ...
};
```

- In the above declaration, ***T*** is the template argument which is a placeholder for the data type used. Inside the class body, a member variable ***var*** and a member function ***someOperation()*** are both of type ***T***.

How to create a class template object?

- To create a class template object, we need to define the data type inside a <> when creation.

```
className<dataType> classObject;
```

- For Example:

```
className<int> classObject;
className<float> classObject;
className<string> classObject;
```

Class Template Example

```cpp
template <class T>
class Calculator
{   private:
        T num1, num2;
    public:
        Calculator(T n1, T n2)
        {
            num1 = n1;
            num2 = n2;
        }
    void displayResult()
    {
        cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;
        cout << "Addition is: " << add() << endl;
        cout << "Subtraction is: " << subtract() << endl;
        cout << "Product is: " << multiply() << endl;
        cout << "Division is: " << divide() << endl;
    }
    T add() { return num1 + num2; }
    T subtract() { return num1 - num2; }
    T multiply() { return num1 * num2; }
    T divide() { return num1 / num2; }
};
```

```cpp
int main()
{
    Calculator<int> intCalc(2, 1);
    Calculator<float> floatCalc(2.4, 1.2);

    cout << "Int results:" << endl;
    intCalc.displayResult();

    cout << endl << "Float results:" << endl;
    floatCalc.displayResult();

    return 0;
}
```

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

Explanation of the Previous Program

In the above program, a class template Calculator is declared.

The class contains two private members of type **T**: **num1** & **num2**, and a constructor to initialize the members.

It also contains public member functions to calculate the addition, subtraction, multiplication and division of the numbers which return the value of data type defined by the user. Likewise, a function **displayResult()** to display the final output to the screen.

In the main() function, two different Calculator objects **intCalc** and **floatCalc** are created for data types: **int** and **float** respectively. The values are initialized using the constructor.

Notice we use **<int>** and **<float>** while creating the objects. These tell the compiler the data type used for the class creation.

This creates a class definition each for int and float, which are then used accordingly.

Then, **displayResult()** of both objects is called which performs the Calculator operations and displays the output.

Example 2

```
#include<iostream>
using namespace std;
template<class T1,class T2> class sample
{
   T1 a;
   T2 b;
   public:
       void getdata()
       {
           cout<<"Enter a and b: "<<endl;
           cin>>a>>b;
       }
       void display()
       {
           cout<<"Displaying values"<<endl;
           cout<<"a="<<a<<endl;
           cout<<"b="<<b<<endl;
       }
};
```

```
int main()
{
    sample<int,int> s1;
    sample<int,char> s2;
    sample<int,float> s3;
    cout <<"Two Integer data"<<endl;
    s1.getdata();
    s1.display();
    cout <<"Integer and Character data"<<endl;
    s2.getdata();
    s2.display();
    cout <<"Integer and Float data"<<endl;
    s3.getdata();
    s3.display();
    return 0;
}
```

Example 2

- Output:

```
Two Integer data
Enter a and b:
2 8
Displaying values
a=2
b=8
Integer and Character data
Enter a and b:
3 k
Displaying values
a=3
b=k
Integer and Float data
Enter a and b:
4 8.99
Displaying values
a=4
b=8.99

Process returned 0 (0x0)   execution time : 16.049 s
Press any key to continue.
```

Explanation of the Previous Program

- In this program, a template class sample is created. It has two data a and b of generic types and two methods: **getdata()** to give input and **display()** to display data. Three object *s1*, *s2* and *s3* of this class is created. *s1* operates on both integer data, *s2* operates on one integer and another character data and *s3* operates on one integer and another float data. Since, *sample* is a template class, it supports various data types.

What is an Exception?

When we write a simple c++ program, the first thing that we see when a program doesn't work correctly are bugs. Most common bugs are **logical errors** and **syntactic errors.** We can overcome these bugs by debugging but sometimes we come across some problems other than logical or syntax errors. These are known as Exceptions.

Types of Exceptions

Exceptions are unusual conditions which occur at **runtime** like low disk space, division by zero, access to array outside its bounds. To overcome these anomalies Exception handling concept plays a major role in C++.

Exceptions are of two types. They are **synchronous** exceptions and **asynchronous** exceptions.

Errors like out of range index or overflow come under synchronous exceptions and errors which are beyond the program like keyboard interrupts are called as asynchronous exceptions

Exception Handle Mechanism

Exception handling performs the following tasks:

1. Finding the problem (Hit the exception).
2. Inform an error is occurred (Throw an exception).
3. Receive error information (Catch the exception).
4. Correct it (Handle the exception).

try-throw-catch

The keywords which are mainly used for exception handing mechanism are try, throw and catch. Usually exceptions are thrown by functions which are invoked from try blocks and the point at which the "throw" is executed is called throw point. Once an exception is thrown to catch block, control cannot return to throw point.

throw − A program throws an exception when a problem shows up. This is done using a **throw** keyword.

catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.

try − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

General Format

```
try
{
…….  //invoke function
}
catch(type arg)//catches exception
{
      ……. //handles exception
}
```

Exception Handling Example

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x = 50;
   int y = 0;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

Catching All exception

- Follow class lecture and see Robert Schildt book.

Rethrowing mechanism

Follow class lecture and see Robert Schildt book.

Components of STL

- Follow Class Lecture