

Generic Programming Concepts: Function and Class Templates

Ali Haider

syedalihaider.ciit@gmail.com

Department of Computer Science IUB

Overview

- Generic Programming
- Templates
- Function Template
- Class Template

Generic Programming

- Generics is the idea to allow type (Integer, String, ... etc and user-defined types) to be a parameter to methods, classes and interfaces.
- For example, classes like an array, map, etc, which can be used using generics very efficiently.
- The method of Generic Programming is implemented to increase the efficiency of the code.
- Generic Programming enables the programmer to write a general algorithm which will work with all data types.
- It eliminates the need to create different algorithms if the data type is an integer, string or a character.

Cont...

- The advantages of Generic Programming are
 - Code Reusability
 - Reduce Function Overloading
 - Once written it can be used for multiple times and cases.
- Generics can be implemented in C++ using Templates.
- 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.

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 template concept can be used in two different ways: with functions and with classes.
- The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.
- There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

Working of Templates

- Templates are expanded at compiler time.
- Compiler does type checking before template expansion.
- The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.

The diagram illustrates the working of templates by showing how a single template definition is expanded into multiple concrete functions based on the types used in the code. Red arrows point from the template definition and its uses in the main function to the generated code blocks.

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Function Template

- The general form of a template function definition is:
- **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.

Example: Template for Sum Function

- `template <class T> T SUM(T first, T second) {`
- `return first+second;`
- `}`
- `int main() {`
- `cout<<SUM(10,13)<<endl;`
- `cout<<SUM(10.4444,13.555556)<<endl;`
- `cout<<SUM(10.8,13.9)<<endl;`
- `cout<<SUM(-10,13)<<endl;`
- `return 0;`
- `}`

Example: Template for Sum Function with Multiple Arguments

- `template <class T> T SUM(T first, T second) {`
- `return first+second;`
- `}`
- `template <class T> T SUM(T first, T second, T third) {`
- `return first+second+third;`
- `}`
- `int main() {`
- `cout<<SUM(10,13,34)<<endl;`
- `cout<<SUM(10.4444,13.555556)<<endl;`
- `cout<<SUM(10.8,13.9,23.55)<<endl;`
- `cout<<SUM(-10,13)<<endl;`
- `return 0;`
- `}`

Class Template

- Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is:
- **template <class type> class class-name {**
- **// class body**
- **}**
- **type** is the placeholder type name, which will be specified when a class is instantiated.
- You can define more than one generic data type by using a comma-separated list.
- Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

Example 1

```
#include<iostream>
using namespace std;
template <class T>
class mypair {
    private:
        T values [2];
    public:
        mypair (T first, T second)
        {
            values[0]=first; values[1]=second;
        }
        void showpair(){
            cout<<"First Element of Pair is "<<values[0]<<endl;
            cout<<"Second Element of Pair is "<<values[1]<<endl;
        }
};

int main(){
    mypair<int> myobject (115, 36);
    mypair<double> myfloats (3.0, 2.18);
    myobject.showpair() ;
    myfloats.showpair() ;
    return 0;
}
```

Example 2

```
#include <iostream>
using namespace std;
template <class T> class mypair {
    T a, b;
public:
    mypair (T first, T second)
    {a=first; b=second;}
    T getmax ();
};
template <class T> T mypair<T>::getmax () {
    T retval;
    retval = a>b? a : b;
    return retval;
}
int main () {
    mypair <int> myobject (100, 75);
    cout << myobject.getmax();
    return 0;
}
```

Cont...

- Notice the syntax of the definition of member function `getmax()`:
- **`template <class T> T mypair<T>::getmax ()`**
- Confused by so many T's? There are three T's in this declaration:
- The first one is the template parameter.
- The second T refers to the type returned by the function.
- And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Example 3

```
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; }
};

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;
}
```