Templates in C++
(Part 1)

Contents

- Function Templates
- Class Templates
- Template Specialization

Review Function Overloading

```
#include<iostream>

using namespace std;


int add(int x, int y)
{
    return x+x;
}

int add(int x, int y, int z)
{
    return x+y+z;
}


int main()
{

    cout << "Addition of two integers: ";
    cout << add(10,20) << endl;;

    cout << "\nAddition of three integers: ";
    cout << add(10,20,30) << endl;

    return 0;
}
```

C++ enables several functions of the same name to be defined, as long as they have different signatures (examining the number, types and order of the arguments in the call)

```
int square(int x)
{
    return x*x;
}

int square(double x)
{
    return x*x;
}
```

Review Function Template

- Overloaded functions are normally used to perform similar operations that involve different program logic on different data types.
- If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently by using function templates.
- Write a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate function template specializations to handle each type of call appropriately.
- Defining a single function template essentially defines a whole family of overloaded functions.

Review Function Template

```
#include<iostream>

using namespace std;

template <typename T>
T square(T sideValue)
{
    return sideValue*sideValue;
}

int main()
{

    cout << "\nInteger Square:"<< square(10) << endl;

    cout << "\nInteger Square:"<< square(5.5) << endl;

    return 0;
}
```

Function Overloading vs Function Template

Functoin Overloading

```
string ToString()
{
  return this;
}

string ToString(string)
{
  return this + string;
}
```

Function Template

```
template<Class T>
void print(T var)
{
 cout << var;
}
```

Function Template Overloading

```
//Funtion Template Overloading
// BY- S.M. Riazul Islam

#include<iostream>

using namespace std;

template <typename T>
void printValue(T val)
{
    T value;

    value=val;
    cout << "The value is:" << value<< endl;
}


template <typename T>
void printValue(T val1, T val2)
{
    T sum;

    sum=val1+val2;
    cout << "The sum is:" << sum<< endl;
}
```

```
int main()
{

    cout << "See Function Template" << endl;
    cout << "\nFloat value" << endl;
    printValue(40.7);

    cout << "\nInteger value" << endl;
    printValue(30);



    cout << "\n\nSee Function Template Overloading" << endl;
    cout << "\nFloat Sum" << endl;
    printValue(30.6, 34.9);

    cout << "\nInteger Sum" << endl;
    printValue(30, 34);

    return 0;
}
```

Function Template w. Multiple Parameters

```
//Function Template with multiple
// Parameters

#include<iostream>

using namespace std;

template <typename T, typename U>
T printSum(T val1, U val2)
{
    return val1+val2;
}

int main()
{

    cout << "\nThe sum is:" << printSum(40.5, 30);

    cout << endl;

    return 0;
}
```

Let's Do It Now

- Please write a program to overload a function template with multiple parameters

```
template <list of typename>
returnType functionName(list of parameters)
{

}

template <list of typename>
returnType functionName(list of parameters)
{

}

int main()
{
    //test your program
}
```

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

// function template printArray definition
template< typename T >
void printArray( const T * const array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
} // end function template printArray

int main()
{
   const int aCount = 5; // size of array a
   const int bCount = 7; // size of array b
   const int cCount = 6; // size of array c

   int a[ aCount ] = { 1, 2, 3, 4, 5 };
   double b[ bCount ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ cCount ] = "HELLO"; // 6th position for null

   cout << "Array a contains:" << endl;

   // call integer function-template specialization
   printArray( a, aCount );

   cout << "Array b contains:" << endl;

   // call double function-template specialization
   printArray( b, bCount );

   cout << "Array c contains:" << endl;

   // call character function-template specialization
   printArray( c, cCount );
} // end main
```

Function Template Specialization

```cpp
void printArray( const int * const array, int count )
{
   for ( int i = 0; i < count; i++ )
      cout << array[ i ] << " ";

   cout << endl;
} // end function printArray
```

```cpp
void printArray( const int * const, int );
void printArray( const double * const, int );
void printArray( const char * const, int );
```

Overloading Not required

Class Templates

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

template <class T>
class Calculator
{
private:
    T num1, num2;

public:
    Calculator(T n1, T n2)
    {
        num1 = n1;
        num2 = n2;
    }

    void displayResult()
    {
        cout << "Addition: " << add() << endl;
        cout << "Subtraction: " << subtract() << endl;
        cout << "Product: " << multiply() << endl;
        cout << "Division: " << divide() << endl;
    }

    T add() { return num1 + num2; }
    T subtract() { return num1 - num2; }
    T multiply() { return num1 * num2; }
    T divide();

};
```

```
template <class T>
T Calculator<T>::divide()
{
    return num1 / num2;
}
```

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

    cout << "Integer Calculations" << endl;
    intCalc.displayResult();

    cout << "\n\nFloat Calculations" << endl;
    floatCalc.displayResult();

    return 0;
}
```

Template Specialization

```cpp
#include <iostream>

using namespace std;

template <class T>
class TestSpecial
{
public:
    TestSpecial(T testValue)
    {
        cout << testValue << " is not a character" << endl;
    }
};


template <>
class TestSpecial<char>
{
public:
    TestSpecial(char testValue)
    {
        cout << testValue << " is a character!" << endl;
    }
};
```

```cpp
int main()
{
    TestSpecial <int> obj1(89);
    TestSpecial <double> obj2(56.7);
    TestSpecial <char> obj3('z');
    TestSpecial <int> obj4(89);
}
```

Non-Type Parameters and Default Types for Class Templates

```
template< typename T >
class Stack
{
```

- It's also possible to use non-type template parameters, which can have default arguments and are treated as *consts*

```
template< typename T, int elements > // nontype parameter elements
```

```
Stack< double, 100 > mostRecentSalesFigures;
```

```
template< typename T = string > // defaults to type string
```

```
Stack<> jobDescriptions;
```

Self-Review

- Distinguish between the terms “function template” and “function-template specialization.”
- Explain which is more like a stencil—a class template or a class-template specialization?
- What’s the relationship between function templates and overloading?
- Why might you choose to use a function template instead of a macro?
- What performance problem can result from using function templates and class templates?

C++ Templates: Exercise (Do at Home)

- Write a template function that swaps the value of two data types, test with float and integer.
  - Change the types to pass by reference
  - Change the types to pass by address
- Write a template function for calculating factorials.