Course Code: CL-1004	**Course: Object Oriented Programming Lab**

Lab # 10

Outline:

- Introduction to template
- Template function
- Template class
- Lab Tasks

Templates in C++

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. **Templates** are powerful features of C++ which allows you to write **generic** programs.

A template is a **blueprint** or **formula** for creating a **generic class** or a **function**. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger **codebase** for the purpose of code **reusability** and **flexibility** of the programs.

What is the difference between function overloading and templates?

Both function overloading and templates are examples of polymorphism feature of OOP. Function overloading is used when multiple functions do similar operations, templates are used when multiple functions do identical operations. You can use overloading when you want to apply different operations depending on the type. Templates provide an advantage when you want to perform the same action on types that can be different

How templates work?

Templates are expanded at compile time. This is like macros. The difference is, the 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 the same function/class.

The concept of templates can be used in two different ways:

- **Function Templates**
- **Class Templates**

Function Templates

A function template works in a similar to a normal <u>function</u>, with one key difference.

A single function template can work with different data types at once but a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

How to declare a function template?
A function template starts with the keyword **template** followed by template parameter/s inside **<** **>** which is followed by function declaration.

```
template <class T>

T someFunction(T arg)

{

    ... .. ...

}
```

In the above code, T is a template argument also called place holder that accepts different data types (int, float), and **class** is a keyword.

You can also use keyword **typename** instead of class in the above example.

When, an argument of a data type is passed to someFunction(), compiler generates a new version of someFunction() for the given data type.

How to call a function template?

We can call the template function someFunction() a couple of ways. Firstly, we can call it by explicitly specifying the type like so

int myint = 5;

someFunction <int>(myint);

//Explicit type parametrizing.
double mydouble = 99.9;

someFunction <double>(mydouble);

However with template function the compiler can perform type deduction to determine the parametrizing types when we don't provide them, hence we can also call someFunction() like so

int myint = 5;

someFunction <>(myint);

//Implicit type parametrizing.
double mydouble = 99.9;

someFunction (mydouble);

The first call with empty angle brackets tells the compiler that we are calling a template function and the second call leaves it up to the compiler to infer. The problem is with the second call you cannot have any other functions by the same name as function templates cannot be overloaded

```cpp
1    // overloaded functions
2    #include <iostream>
3    using namespace std;
4
5    int Max (int a, int b)
6    {
7       Return  a < b ? b:a;
8    }
9
10   double Max (double a, double b)
11   {
12      return a < b ? b:a;
13   }
14
15   int main ()
16   {
17      cout << sum (10,20) << '\n';
18      cout << sum (1.0,1.5) << '\n';
19      return 0;
20   }
```

This code can be reduced by templates

```cpp
1    #include <iostream>
2    using namespace std;
3
4    template <typename T1>
5    T1  Max (T1  a, T1  b)  {
6       return a < b ? b:a;
7    }
8
9    int main () {
10     cout << "Max integer are: " << Max(22, 2) << endl;
11     cout << "Max float are: " << Max(3.9, 22.8) << endl;
12      return 0;
13   }
```
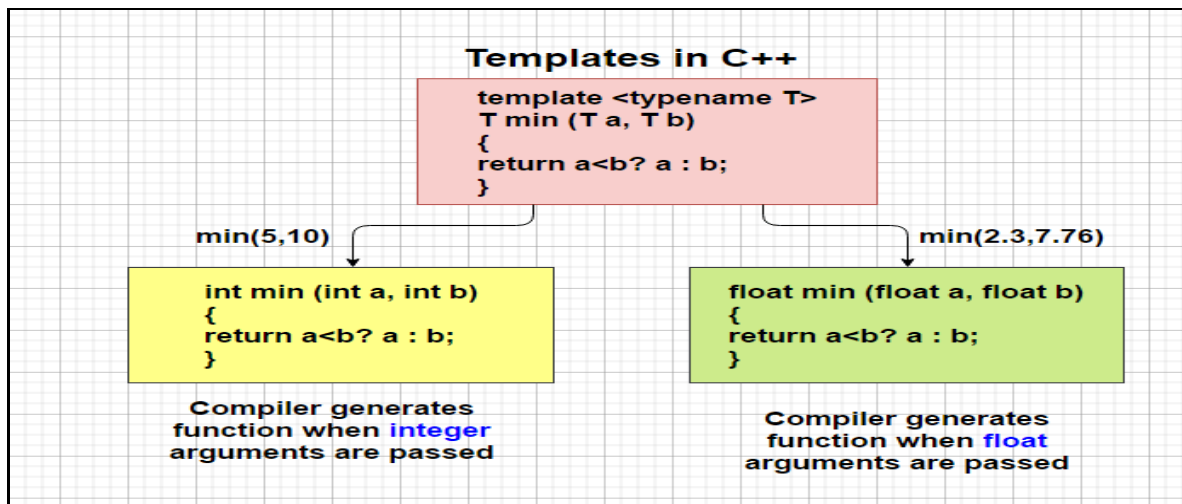
Function template with more than one type parameter

All you need to do is add the extra type to the template prefix, so it looks like this:

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

```
1    #include <iostream>
2    using namespace std;
3
4    template <typename T1,typename T2> // multiple args for combinition of data types
5    T  Max (T1  a, T2  b)  {
6        return a < b ? b:a;
7    }
8
9    int main () {
10     cout << "Max integer are: " << Max(22, 2.77) << endl;
11     cout << "Max float are: " << Max1(3.9, 22)) << endl;
12       return 0;
                                    T Max1 (T  a, T1  b)
13    }
```

Can you have unused type parameters?

No, you may not. If you declare a template parameter then you absolutely must use it inside of your function definition otherwise the compiler will complain. So, in the example above, you would have to use both T1 and T2, or you will get a compiler error.

Class Templates

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

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, you need to define the data type inside a < > when creation.

className<dataType> classObject;

For example:

className<int> classObject;

className<float> classObject;

className<string> classObject;

```cpp
Class Template Example Program in C++
1  // class templates
2  #include <iostream>
3  using namespace std;
4
5  template <class T>
6  class mypair {
7      T a, b;
8    public:
9      mypair (T first, T second)
10        {a=first; b=second;}
11      T getmax ();
12 };
13
14 template <class T>
15 T mypair<T>::getmax ()
16 {
17   T retval;
18   retval = a>b? a : b;
19   return retval;
20 }
21
22 int main () {
23   mypair <int> myobject (100, 75);
24   cout << myobject.getmax();
25   return 0;
26 }
```

Can there be more than one argument to templates?

Yes, like normal parameters, we can pass more than one data types as arguments to templates. The following example demonstrates the same.

```cpp
3    template <class t1 , class t2>
4    class sample{
5        t1 a;    t2 b;
6        public: void getdata(){
7                cout << "enter value for a and b" << endl;
8                cin >> a >> b;}
9                void display(){
10               cout<< "value of a: " << a << endl;
11               cout<< "value of b: " << b << endl;}  };
12   int main(){
13       sample  <int , int >s1;
14       sample <float , float > s2;
15       cout<< "two integer data" << endl;
16       s1.getdata();
17       s1.display();
18       cout<< "two float data" << endl;
19       s2.getdata();
20       s2.display();
21
22   }
```

Specializing templates

Normally when we write a template class or function we want to use it with many different types, however sometimes we want to code a function or class to make use of a particular type more efficiently. This is when we use a template specialization. To declare a template specialization we still use the template keyword and angle brackets <> but leave out the parameters like so.

template<>

So we can create a function called printFunction which prints out its type and value like so

template<typename T>

void printFunction(T arg) {

    cout << "printFunction arg is type " << typeid(arg).name() << " with value " << arg << endl;

}

Then we can specialize this for integer values like so.

template<>

void printFunction(int intarg) {

        cout << "printFunction specialization with int arg only called with type " << typeid(intarg).name() << " with value " << intarg << endl;

}

And we can do the same thing with classes : This is the syntax used in the class template specialization:

template <> class mycontainer <char> { ... };

First of all, notice that we precede the class template name with an empty template<> parameter list. This is to explicitly declare it as a template specialization.

But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:

```
template <class T> class mycontainer { ... };
template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

Exercise:

1. Create a C++ Program to add, subtract, multiply and divide two numbers using class template. Two numbers can be of the same datatype or combination of different data types.

2. Create a C++ Program to swap the data using template function, instead of calling a function by passing a value, use call by reference , Two numbers can be of same datatype or combination of different data types.

3. Create a C++ class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, declare a class template specialization for that type