Object Oriented Programming (CT-260)
Lab 10

Introduction to Template Function & Template Class

Objectives

The objective of this lab is to familiarize students with template functions and template classes. By the end of this lab, students will be able to understand the concepts of template and generic functions along with template classes using examples.

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

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.

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 except that 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 function, 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.

Defining 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.

Calling 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

```
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

```
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.

```
// overloaded functions
#include<iostream>
using namespace std;

int Max (int a, int b)
{
     return a < b ? b : a;
}

double Max (double a, double b)
{
     return a < b ? b : a;
}

int main()
{
     cout << sum (10, 20) << endl;
     cout << sum (1.0, 1.5) << endl;
     return 0;
}
```

Example 1: Template Functions

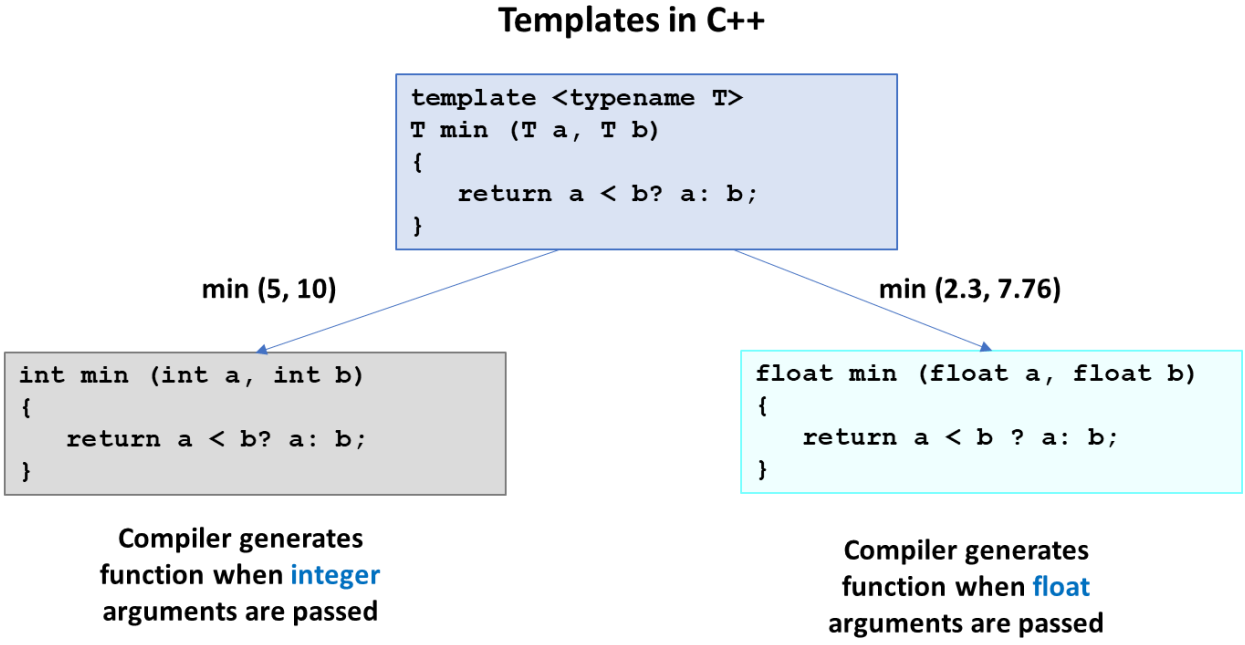
This code can be reduced by templates.

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

template <typename T1>
T1 Max (T1 a, T1 b){
     return a < b? b : a;
}

int main()
{
     cout << "Max integer are:" << Max(22, 2) << endl;
     cout << "Max float are:" << Max (3.9, 22.8) << endl;
     return 0;
}
```

The figure below explains the execution of the above-mentioned program and the concept of how templates work.



Function template with more than one type of parameters

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...
}
```

Example 2: Template Functions with multiple type parameters

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

template <typename T1, typename T2>
T Max (T1 a, T2 b){
```

```
        return a < b? b : a;
    }

    int main()
    {
        cout << "Max integer are:" << Max(22, 2.77) << endl;
        cout << "Max float are:" << Max (3.9, 22) << endl;
        return 0;
    }
```

Template Functions with Unused type parameters

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.

Declaring 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.

Creation of 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;
```

Example 3: Class Template

```
    // Class Templates
    #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? b : a;
     return rectal;
}

int main()
{
     mypair <int> myobject (100,75);
     cout << myobject.getmax();
     return 0;
}
```

Multiple Argument Class Templates

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

Example 4: Multiple Argument Template

```
#include<iostream>
#include<conio.h>
using namespace std;

template <class t1, t2>
class sample {
      t1 a; t2 b;
      public:
          void getdata(){
               cout<< "Enter value for a and b" << endl;
               cin >> a >> b;
          }
          void display(){
               cout << "Value of a:" << a << endl;
               cout << "Value of b:" << b << endl;
          }
};

int main()
{
     sample <int, int> s1;
     sample <float, float> s2;
     cout << "Two integer data" << endl;
     s1. getdata();
     s1. display();
     cout << "two float data" << endl;
     s2. getdata();
     s2. display();
     return 0;
}
```

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 as :

```
template <>
```

Hence, we can create a function called printFunction which prints out its type and value as shown in code below.

```
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 as shown in the below mentioned code

```
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. The syntax used in the class template specialization is mentioned below:

```
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.

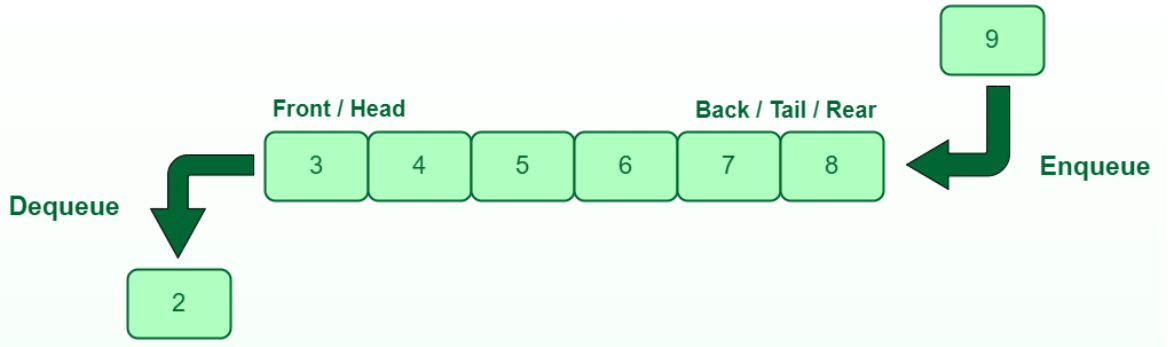
Exercise

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.

3. Create a C++ class called mycontainer that can store one element of any type and 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.

4. Create an abstract class template for 1D dynamic array. Inherit a dynamic Queue template class from the abstract class which must have following methods: isFull( ), isEmpty( ), size( ), Front( ), Rear( ), enqueue( ), dequeue( ), resize( ).



5. A print shop receives print job requests from various clients throughout the day. The print shop wants to efficiently manage the print jobs and ensure they are printed in the correct order.
   - Create an object of class queue that you have just created to represent the printer job queue.
   - Whenever a print job request arrives, enqueue (add) the job to the back of the queue.
   - The print shop has a printer that can handle one job at a time. If the printer is idle and there are jobs in the queue, dequeue (remove) the job from the front of the queue and start printing it.
   - Once a print job is completed, check if there are more jobs in the queue. If there are, dequeue the next job and start printing it. If the queue is empty, the printer remains idle.
   - Repeat steps 3 and 4 until all print jobs have been completed.

This scenario can be solved using a queue because it follows the First-In-First-Out (FIFO) principle. The print jobs are handled in the same order they arrived at the print shop. The queue ensures that jobs are printed in the correct sequence, without skipping or rearranging the order in which they were received.