

# Lab Manual - Templates

## Objective

- Implement template functions
- Implement template classes
- Implement templates with multiple types

## Function Templates

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one variable type or class without repeating the code for each type. This is achieved through *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as parameter. These function templates can use these parameters as if they were regular types. The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
```

While defining a function template the body of the function definition is preceded by a statement `template <class identifier>`. The `identifier` can then be used as the data type of the parameters, the return type of the function, the data type of local variables and/or the data types of parameters.

### Exercise 1:

- a. Write two function templates `GetMax` and `GetMin` that take two arguments and return the maximum and minimum of the two respectively.
- b. Then paste the following code in your source file and run the program. The program should run peacefully.

```
int main ()
{
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=GetMax<int>(i,j);
    n=GetMin<long>(l,m);
    cout << k << endl;
    cout << n << endl;
    return 0;
}
```

- c. Now remove the `<int>` and `<long>` from the code above and execute again. Does the program still work?

- d. Now replace the **main** function above with the `main` given below. You will need to change the code (declaration and definition) for `GetMin` and `GetMax` so that the following code works without an error.

```
int main ()
{
    char i='Z';
    int j=6, k;
    long l=10, m=5, n;
    k=GetMax<int,long>(i,m);
    n=GetMin<int,char>(j,l);
    cout << k << endl;
    cout << n << endl;
    return 0;
}
```

- e. Now remove the `<int,long>` and `<int,char>` from this new `main` and re-run the program, is there any trouble with this version?

## Class Templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. In this exercise we shall implement one such class template called a `Pair` that is used to store two variables of the same type.

In order to define a class template we use the following syntax:

```
template <class identifier> class_declaration;
```

### Exercise 2:

The specifications for the `Pair` class are given below:

- The class **Pair** has a private data member `values` which is an array of size two and its type is `T(identifier/class/template)`.
- A constructor that takes two parameters.
- A public member function called `GetMax` that returns the greater of the two stored variables. (This function has to be defined inline i.e. inside the class body).
- A public member function called `GetMin` that returns the smaller of the two stored variables. (This function has to be defined outside the class body). This is done by using the following syntax.

```
template <class identifier>
identifier Pair< identifier >::GetMax(){. . .}
```

- Now replace (which means you have to comment the previous code) the `main` method with the following, the program should run like a river.

```
int main ()
{
    Pair <double> myobject (1.012, 1.01234);
    cout << myobject.getmax();
    return 0;
}
```

## Template Specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template. For example, let's suppose that we have a very simple class called `Container` that can store one element of any type and that it has just one member function called `increase` that increases its value and also returns the increased value. But we find that when it stores an element of type `char` it would be more convenient to have a completely different implementation of the `Container` class with a function member `uppercase` that changes the case of the stored character to the upper case and returns the uppercase character, so we decide to declare a class template specialized for that type.

The general class template looks like:

```
template <class T>
class Container
{
    T data;
    ...
};
```

and the special class template (for `char` type data) is declared separately as:

```
template<>
class Container<char>
{
    char data;
    ...
};
```

### Exercise 3:

Now do the following,

- a. Complete the declaration and implementation of these templates with member function `increase` in the first template and `uppercase` in the second with the required functionality as stated above. (Note that you are not allowed to use the `toupper()` function) also you are not allowed to create inline functions.
- b. Add the following `main` to your program and watch it run.

```
int main ()
{
    Container<int> myint (7);
    Container<char> mychar ('j');
    cout << myint.increase() << endl;
    cout << mychar.uppercase() << endl;
    return 0;
}
```

### Non-Type Parameters for Templates

Besides the template arguments that are preceded by the `class` keyword, which represent types, templates can also have regular typed parameters, similar to those found in functions.

To try this out consider the following class template:

```
template <class T, int N>
class Sequence {
    T memblock [N];
public:
    void setmember (int x, T value);
    T getmember (int x);
};
```

`Sequence` is a class that stores a Sequence of elements, but here `N` is an integer. The member function `setmember` sets the member at position `x` in the `memblock` with value and `getmember` returns the value at index `x`.

### Exercise 4:

You are required to do the following:

- a. Implement the `Sequence` class. (do not use inline definitions)

b. Copy the following **main** and add it to your program, again watch it run.

```
int main ()
{
    Sequence <int,5> myints;
    Sequence <double,5> myfloats;
    myints.setmember (0,100);
    myfloats.setmember (3,3.1416);
    cout << myints.getmember(0) << '\n';
    cout << myfloats.getmember(3) << '\n';
    return 0;
}
```

### Exercise 5:

As a last exercise you have to change (augment) your code so that the following **main** runs without errors and gives the expected output. Here `Pair` refers to the class you created earlier, where each element in the `Pair` has the same type.

```
int main ()
{
    Pair <double> y (2.23,3.45);
    Sequence <Pair<double>,5> myPairs;
    myPairs.setmember (0,y);
    cout << myPairs.getmember(0) << '\n';
}
```

Notice that for this code to work you have to overload `<<` operator for the `Pair` class template. Also the pair class template will need a default constructor. The friend function in this case is declared in a class template using the following syntax:

```
template <class T>

class Pair{

...

```

```
template <class identifier> friend ostream & operator << (ostream& out,const
Pair<identifier>& p);
```

```
};
```