Experiment –Week 14

Template Classes

Templates are a very powerful feature of C++. They allow you to write a single code segment fora set of related functions, called a function template, and for a set of related classes, called a classtemplate. The syntax we use for templates is:

```
template <class
Type>declaration;
```

Function templates: A function template behaves like a function except that the template can have arguments of many different types In other words; a function template represents a family of functions. The general format of a function template is as follows:

```
template<class T>

return_type functionname(argument T )

{

// body of function with Type T

};
```

Class templates: A class template provides a specification for generating classes based on parameters. Class templates are generally used to implement containers.

```
template <class
type>

class class_name

{

//(Body of the class)

};
```

Let's take an example, just declare an "integer" generic class. This class contains two member variables which are of Type T, and a member function "greater" to return greater number.

```
template<class
T>

class number
```

```
{

T no_1,
no_2;
public:

number (T n1, T n2)

{   no_1=n1;

    no_2=n2;

}

T greater();

};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type “int” with the values 115 and 36 we would write:

```
number<int> myobject1(1,2);
```

This same class could also be used to create an object to store any other type, such as:

```
number<double> myobject2(1.2,2.2);
```

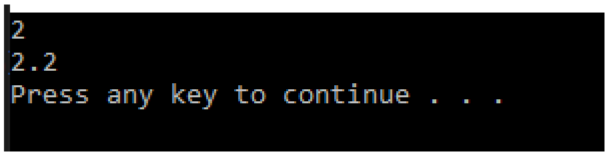
If a member function is defined outside the definition of the class template, it shall be precededwith the template <...>prefix:
Notice the syntax of the definition of a member function “greater” is:

```
template<class T>

T number<T>::greater()

{

        // body of greater definition

}
```

Exercise – 1 Write Output of the following code

```
template<class T>
class number
{
T no_1, no_2;
public:
number (T n1, T n2) { no_1=n1; no_2=n2; }
T greater();
};
template<class T>
T number<T>::greater()
{
if(no_1>no_2)
return no_1;
else
return no_2;
}
void main()
{
number<int> myobject1(1,2);
cout<<myobject1.greater()<<endl;
number<double> myobject2(1.2,2.2);
cout<<myobject2.greater()<<endl;
}
```

Output:



```
2
2.2
Press any key to continue . . .
```

Exercise 2: **(10 points)**

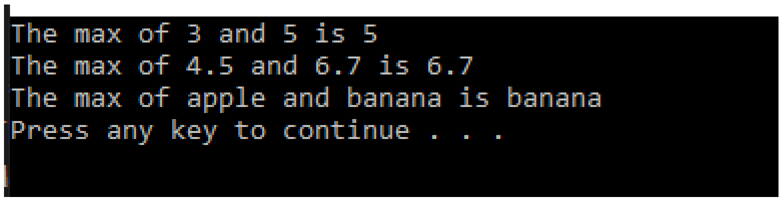
Study the myMAXfunction provided below. You are required to create a C++ template based myMAXfunction and test it on different built-in data types.

```
//Make a template out of this function. Don't forget the return type.
int myMax(int one, int two)

{

      int bigger;
      if(one < two)

            bigger = two;

      else

      bigger = one;
      returnbigger;

}

int main()

{

      int i_one = 3, i_two = 5;

      cout <<"The max of "<< i_one <<" and "<< i_two <<" is "

      << myMax(i_one, i_two) << endl;

//Test your template on float and string types
      return 0;

}
```

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

template <typename T>
```

```cpp
T myMax(T one, T two) {
   if (one < two) {
      return two;
   }
   else {
      return one;
   }
}

int main() {
   int i_one = 3, i_two = 5;
   cout << "The max of " << i_one << " and " << i_two << " is "
      << myMax(i_one, i_two) << endl;

   float f_one = 4.5, f_two = 6.7;
   cout << "The max of " << f_one << " and " << f_two << " is "
      << myMax(f_one, f_two) << endl;

   string s_one = "apple", s_two = "banana";
   cout << "The max of " << s_one << " and " << s_two << " is "
      << myMax(s_one, s_two) << endl;

   system("pause");
   return 0;
}
```

Output:



```
The max of 3 and 5 is 5
The max of 4.5 and 6.7 is 6.7
The max of apple and banana is banana
Press any key to continue . . .
```

Exercise – 3:

Consider the class of points in the *xy* plane. The location of each point is determined by the real numbers (x, y) specifying the cartesian coordinates. The class definition is:

```
#include<iostream>
using namespace std;
class point{
public:
point();
point(double value_x, double value_y);
double get_x() const;
double get_y() const;
void print() const;
void move(double dx, double dy);
private:
double x, y;
};
point::point(){
x = 0.0; y = 0.0;

}
point::point(double a, double b){
x = a; y = b;
}
void point::print() const{
cout<<x<<""<<y<< endl;
}
double point::get_x() const{
return x;
}
double point::get_y() const{
return y;
}
void point::move(double dx, double dy){
```

```
x = x+dx;
y = y+dy;
}
```

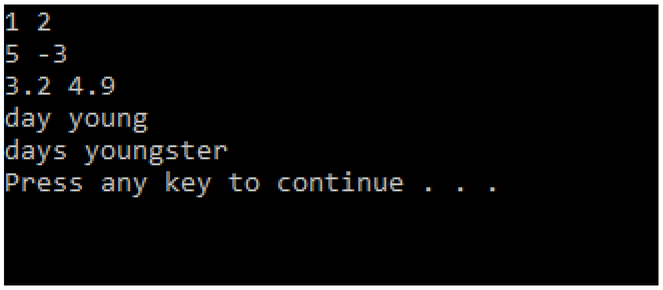
Generalize the class Point into a template and test your code using following main function.

```
int main()
{
point<int> A = point<int>(1, 2);
A.print();
A.move(4, -5);
A.print();
point<float>B(3.2, 4.9);
cout << B.get_x() <<""<< B.get_y() << endl ;
point<string> C("day", "young");
C.print();
C.move("s","ster");
C.print();
return 0;
}
```

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

template <typename T>
class point{
public:
   point();
   point(T value_x, T value_y);
   T get_x();
   T get_y();
   void print();
   void move(T dx, T dy);
private:
   T x, y;
};

template <typename T>
point<T>::point(){
   x = 0.0; y = 0.0;
}
```

```cpp
template <typename T>
point<T>::point(T a, T b){
   x = a; y = b;
}

template <typename T>
void point<T>::print(){
   cout << x << " " << y << endl;
}

template <typename T>
T point<T>::get_x(){
   return x;
}

template <typename T>
T point<T>::get_y(){
   return y;
}

template <typename T>
void point<T>::move(T dx, T dy){
   x = x + dx;
   y = y + dy;
}

int main()
{
   point<int> A(1, 2);
   A.print();
   A.move(4, -5);
   A.print();
   point<float> B(3.2, 4.9);
   cout << B.get_x() << " " << B.get_y() << endl;
   point<string> C("day", "young");
   C.print();
    C.move("s","ster");
    C.print();
    system("pause");
   return 0;
}
```

Output:



```
1 2
5 -3
3.2 4.9
day young
days youngster
Press any key to continue . . .
```