Experiment No: 8 Date:

Aim: To study templates in c++

Theory:

1. Multiple Parameters in Classes
In C++, templates support multiple parameters in classes, allowing developers to create generic classes that can work with different data types. This flexibility enhances code reusability and adaptability.

2. Function Template
Function templates enable the creation of generic functions that can operate on various data types. They provide a concise way to write algorithms without duplicating code for each data type, promoting code efficiency.

3. Template Function
Template functions are functions defined with template parameters, allowing them to accept and operate on multiple data types. This eliminates the need for redundant function implementations, enhancing code maintainability.

4. Member Function Template
C++ supports member function templates, allowing template parameters within class methods. This feature is powerful for designing generic algorithms inside classes, providing flexibility in member function implementations.

5. Class Template
Class templates in C++ allow developers to create generic classes that work with different data types. This promotes code abstraction and reusability, as the template can adapt to various scenarios without rewriting the class.

6. Template Classes
Template classes are instances of class templates with specific data types defined. They offer a way to create customized classes for different scenarios while benefiting from the generic structure provided by the class template.

7. Multiple Parameters in Template Classes
Similar to functions, template classes can have multiple parameters, allowing developers to design highly flexible and reusable components. This is particularly useful when a class needs to operate on more than one data type simultaneously.

8. Overloading of Template Functions
C++ supports overloading template functions, enabling developers to define multiple template functions with the same name but different parameter types. This enhances code expressiveness and accommodates various use cases.

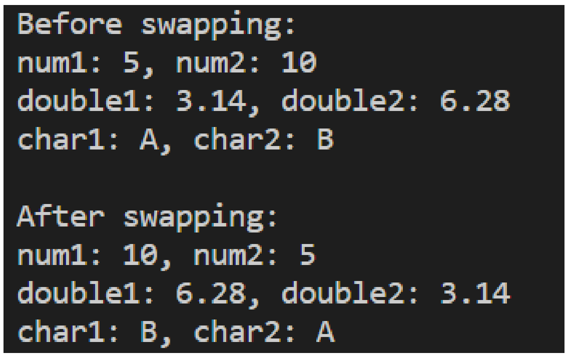
9. Non-Type Template Arguments
Non-type template arguments allow developers to pass values, such as integers or enumerations, as template parameters. This feature enables the creation of generic classes and functions that operate on both data types and constant values.

These subtopics cover the essential aspects of templates in C++, providing a comprehensive overview of how templates facilitate generic programming and enhance code flexibility.

[A] Write a C++ program to implement a function template to swap two elements

```
#include <iostream>
using namespace std;
template <typename T>
void swapElements(T& a, T& b) {
   T temp = a;
   a = b;
   b = temp;
}

int main() {
   int num1 = 5, num2 = 10;
   double double1 = 3.14, double2 = 6.28;
   char char1 = 'A', char2 = 'B';

   cout << "Before swapping:" << endl;
   cout << "num1: " << num1 << ", num2: " << num2
<< endl;
   cout << "double1: " << double1 << ", double2: " <<
double2 << endl;
   cout << "char1: " << char1 << ", char2: " << char2 <<
endl;

   swapElements(num1, num2);
   swapElements(double1, double2);
   swapElements(char1, char2);

   cout << "\nAfter swapping:" << endl;
   cout << "num1: " << num1 << ", num2: " << num2
<< endl;
   cout << "double1: " << double1 << ", double2: " <<
double2 << endl;
   cout << "char1: " << char1 << ", char2: " << char2 <<
endl;

   return 0;
}
```

Output:



```
Before swapping:
num1: 5, num2: 10
double1: 3.14, double2: 6.28
char1: A, char2: B

After swapping:
num1: 10, num2: 5
double1: 6.28, double2: 3.14
char1: B, char2: A
```

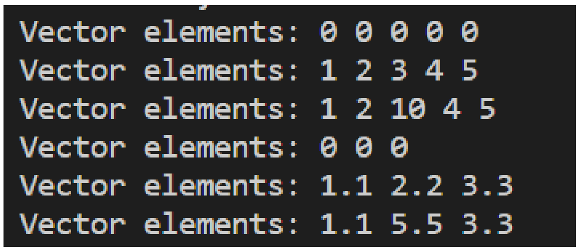
[B] Write a C++ program to create a class template to represent a generic vector. Include the member functions to perform the following tasks

1. Create the vector
2. To modify the value of a given element
3. To display the vector elements

```
#include <iostream>
using namespace std;
template<class T> class vector {
   T* v; // Type T vector
   int size;
public:
   vector(int m) {
      v = new T[size = m];
      for (int i = 0; i < size; i++)
         v[i] = 0;
   }

   vector(T* a, int m) {
      v = new T[size = m];
      for (int i = 0; i < size; i++)
         v[i] = a[i];
   }

   void modifyElement(int index, T value) {
      if (index >= 0 && index < size)
         v[index] = value;
      else
         cout << "Index out of bounds." << endl;
   }

   void display() {
      cout << "Vector elements: ";
      for (int i = 0; i < size; i++) {
         cout << v[i] << " ";
      }
      cout << endl;
   }

   ~vector() {
      delete[] v;
   }
};

int main() {
   vector<int> intVector(5);
   intVector.display();

   int intArray[] = {1, 2, 3, 4, 5};
   vector<int> intVectorFromArray(intArray, 5);
   intVectorFromArray.display();

   intVectorFromArray.modifyElement(2, 10);
   intVectorFromArray.display();
```

```
    vector<double> doubleVector(3);
    doubleVector.display();

    double doubleArray[] = {1.1, 2.2, 3.3};
    vector<double>
doubleVectorFromArray(doubleArray, 3);
    doubleVectorFromArray.display();


    doubleVectorFromArray.modifyElement(1, 5.5);
    doubleVectorFromArray.display();

    return 0;
}
```

Output:



```
Vector elements: 0 0 0 0 0
Vector elements: 1 2 3 4 5
Vector elements: 1 2 10 4 5
Vector elements: 0 0 0
Vector elements: 1.1 2.2 3.3
Vector elements: 1.1 5.5 3.3
```

Conclusion: All the concepts regarding templates were understood and successfully implemented in the above codes.

Nitesh Naik

(Subject Faculty)