National University
of computer and emerging sciences

Lab 15

Items	Description
Course Title	Object Oriented Programming
Lab Title	Templates in c++
Duration	3 Hours
Tools	Eclipse/ C++
Objective	To get familiar with the use of different concepts in classes in c++

Templates in C++

In C++, **templates** allow you to write generic and reusable code. Templates are used to define functions or classes that can operate on any data type without specifying the type explicitly. This helps to write code that works with different types of data while avoiding redundancy.

There are two types of templates in C++:

1. **Function Templates** – Allow you to write a generic function that works for any data type.
2. **Class Templates** – Allow you to define a class that works with any data type.

1. Function Templates

A function template defines a function blueprint that can work with any data type. The syntax for defining a function template is:


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

// Function Template
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    cout << add(5, 3) << endl;        // Works with int
    cout << add(2.5, 3.7) << endl;    // Works with double
    cout << add('A', 2) << endl;      // Works with char (outputs 'C' because 'A' + 2 gives 'C')

    return 0;
}
```

In this example:

- The function `add` can handle both integers and floating-point numbers because of the template.
- The type `T` is inferred by the compiler based on the arguments passed during the function call.

2. Class Templates

Class templates allow the creation of classes that can operate with any data type. The syntax for class templates is:

```
template <typename T>
class ClassName {
    T data;
public:
    ClassName(T val) : data(val) {}
    T getData() { return data; }
};
```

Here, T is a placeholder for the type that will be specified when the object is created.

Example 1: Generic Box Class

Here's a simple class template that works with any type of data:

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

// Class Template
template <typename T>
class Box {
    T value;
public:
    Box(T val) : value(val) {}
```

```
    T getValue() {
        return value;
    }
};

int main() {
    Box<int> intBox(42);            // Box with an int
    Box<double> doubleBox(3.14);    // Box with a double

    cout << "Integer Box: " << intBox.getValue() << endl;
    cout << "Double Box: " << doubleBox.getValue() << endl;

    return 0;
}
```

In this example, the `Box` class template works with any data type. When creating an object of `Box`, the type is specified (like `Box<int>` or `Box<double>`).

Example 2: Multiple Template Parameters in a Class

You can define class templates with multiple parameters. For instance, a class that stores two values of different types:

```
template <typename T, typename U>
class Pair {
    T first;
    U second;
public:
    Pair(T f, U s) : first(f), second(s) {}
    void display() {
        cout << "First: " << first << ", Second: " << second << endl;
    }
};
int main() {
    Pair<int, double> pair1(5, 3.14);
    Pair<string, int> pair2("Age", 30);
```

```
    pair1.display();

    pair2.display();

    return 0;
}
```

Conclusion

Templates in C++ are powerful tools for writing generic and reusable code. They enable the creation of functions and classes that work with any data type, thus improving code efficiency and maintainability. Through function templates, class templates, and specialization, you can handle a wide variety of data types while keeping the code compact and flexible.

Lab Task

Task: Implement a Template-Based Vector Class

Objective:
Create a class `Vector` that mimics the basic functionality of the `std::vector` in C++. The class should be template-based, allowing it to work with any data type (e.g., `int`, `double`, `char`, etc.). Implement functions for adding elements, accessing elements, and other key operations like resizing and operator overloading.

Requirements:

1. Class Definition:

- The `Vector` class should be defined as a template class. The template should allow the class to store any data type (`T`), where `T` will be deduced at runtime based on the type of data passed during object creation.

2. Member Variables:

- **`T* data`**: A dynamically allocated array that will hold the elements of the vector.
- **`int size`**: The current number of elements in the vector.
- **`int capacity`**: The current capacity of the vector (the total number of elements it can hold before resizing is necessary).

3. Constructor:

- The constructor should initialize the vector with a specified **capacity** (default should be 10). It should also initialize the `size` to 0 and allocate memory for the `data` array.
- **Vector(int cap = 10);**

4. Destructor:

- The destructor should properly release the dynamically allocated memory using `delete[]`.
- **~Vector();**

5. Member Functions:

- **`push_back(const T& value)`**: Adds a new element at the end of the vector. If the vector's capacity is full, it should resize automatically (doubling the capacity).

- `resize()`: A private function that should resize the vector when it reaches full capacity. This function will create a new array of double the capacity, copy existing elements, and update the `data` pointer.

- `T& operator[](int index)`: Allows element access using the `[]` operator. This should return the element at the specified index.

- `Vector<T> operator+(const Vector<T>& other)`: Overload the `+` operator to add corresponding elements of two vectors of the same size. The result should be returned as a new vector.

- `int getSize() const`: Returns the current number of elements in the vector.

Additional Requirements:

- The vector class should automatically resize when it runs out of capacity by doubling its size.
- Proper memory management must be implemented, including a deep copy constructor and proper cleanup in the destructor.
- Ensure the class is flexible and works with any data type (`int`, `double`, `string`, etc.).