Objectives:

To understand the concept and use of templates in C++ for enabling generic programming, to
explore function templates, their overloading, class templates, and how they can be extended
using inheritance. To gain familiarity with the Standard Template Library (STL), its components
such as containers, iterators, and algorithms, and how STL supports efficient, type-safe, and
reusable code.

Theory

Function Template

A function template allows writing a single function definition to work with different data types. Instead of writing separate functions for int, float, etc., we use templates to generalize the function.

Syntax:

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

Example:

```
#include <iostream> using namespace std;
template <typename T> T add(T a, T b) {
return a + b;
}
int main() {
cout << "Sum: " << add(3, 4) << endl;
cout << "Sum: " << add(3.5, 2.1) << endl;
return 0;
}
```

Overloading Function Template

Templates can be overloaded like regular functions. You can define both template and non-template versions or different template versions with varying parameters.

Syntax:

```
template <typename T> T maxVal(T a, T b);
int maxVal(int a, int b, int c);
```

Example:

```
#include <iostream> using namespace std;
template <typename T> T maxVal(T a, T b) {
return (a > b) ? a : b;
}
int maxVal(int a, int b, int c) {
return (a > b && a > c) ? a : (b > c ? b : c);
}
int main() {
cout << maxVal(3, 7) << endl;
cout << maxVal(3, 7, 5) << endl;
return 0;
}
```

Class Template

A class template allows creating a class to handle any data type. Useful for generic data structures like Stack, Queue, etc.

Syntax:

```
template <typename T> class Box {
```

```
T value;
public:
void set(T v);
T get();
};
```
Example:
```
#include <iostream> using namespace std;
template <typename T> class Box {
T value;
public:
void set(T v) { value = v; }
T get() { return value; }
};
int main() {
Box<int> b1;
b1.set(10);
cout << b1.get() << endl;
Box<string> b2;
b2.set("Hello");
cout << b2.get() << endl;
return 0;
}
```

Derived Class Template

Templates can be used in inheritance. A derived class template can inherit from a base class template, allowing reuse and extension of generic behaviors.

Syntax:
```
template <typename T> class Base {};
template <typename T> class Derived : public Base<T> {};
```
Example:
```
#include <iostream> using namespace std;
template <typename T> class Vehicle {
public:
void showType() {
cout << "Generic Vehicle" << endl;
}
};
template <typename T> class Car : public Vehicle<T> {
public:
void showType() {
cout << "Car of type: " << typeid(T).name() << endl;
}
};
int main() {
Car<int> c;
c.showType();
return 0;
}
```

Introduction to Standard Template Library

The STL is a powerful set of C++ template classes that provide generic containers, algorithms, and iterators.

The three main components of STL are:

Containers: Store collections of data (e.g., vector, list)

Iterators: Pointers to elements in containers

Algorithms: Operate on container elements (sort, find)1. Containers store and manage collections of objects. Example: vector, list, set, map.

Syntax:
```
#include <vector> vector<int> v;
v.push_back(10);
```
Example:

```
#include <iostream> #include <vector> using namespace std;
int main() {
vector<int> v = {1, 2, 3};
v.push_back(4);
for (int x : v)
cout << x << " ";
return 0;
}
```

2. Iterators are objects that act like pointers. They help traverse containers.

Syntax:

```
vector<int>::iterator it;
```

Example:

```
#include <iostream> #include <vector> using namespace std;
int main() {
vector<int> v = {10, 20, 30};
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it)
cout << *it << " ";
return 0;
}
```

3. Algorithms

STL Algorithms perform operations like searching, sorting, counting, transforming, etc., on container elements.

Syntax:

```
#include <algorithm> sort(container.begin(), container.end());
```

Example:

```
#include <iostream> #include <vector> #include <algorithm> using namespace std;
int main() {
vector<int> v = {4, 1, 3, 2};
sort(v.begin(), v.end());
for (int x : v)
cout << x << " ";
return 0;
}
```

Q1) Write a function template swapValues() that swaps two variables of any data type. Demonstrate its use with int, float, and char.

Code:

```
#include <iostream>
using namespace std;
template <typename T>
void swapValues(T& a, T& b) {
T temp = a;
a = b;
b = temp;
}
int main() {
int int1 = 5, int2 = 10;
float float1 = 5.5, float2 = 10.5;
char char1 = 'A', char2 = 'B';
cout << "Before swapping:" << endl;
cout << "int values: " << int1 << ", " << int2 << endl;
cout << "float values: " << float1 << ", " << float2 << endl;
cout << "char values: " << char1 << ", " << char2 << endl;
swapValues(int1, int2);
swapValues(float1, float2);
swapValues(char1, char2);
cout << "After swapping:" << endl;
cout << "int values: " << int1 << ", " << int2 << endl;
cout << "float values: " << float1 << ", " << float2 << endl;
cout << "char values: " << char1 << ", " << char2 << endl;
```

```
return 0;
}
```
Output:
```
Before swapping:
int values: 5, 10
float values: 5.5, 10.5
char values: A, B
After swapping:
int values: 10, 5
float values: 10.5, 5.5
char values: B, A
```

Q2) Write a program to overload a function template maxValue() to find the maximum of two values (for same type) and three values (for same type). Call it using int, double, and char.

Code:
```
#include <iostream>
using namespace std;
template <typename T>
T maxValue(T a, T b) {
return (a > b) ? a : b;
}
template <typename T>
T maxValue(T a, T b, T c) {
return maxValue(maxValue(a, b), c);
}
int main() {
int int1 = 5, int2 = 10, int3 = 7;
double double1 = 5.5, double2 = 10.5, double3 = 7.5;
char char1 = 'A', char2 = 'B', char3 = 'C';
cout << "Max int (two values): " << maxValue(int1, int2) << endl;
cout << "Max int (three values): " << maxValue(int1, int2, int3) << endl;
cout << "Max double (two values): " << maxValue(double1, double2) << endl;
cout << "Max double (three values): " << maxValue(double1, double2, double3) << endl;
cout << "Max char (two values): " << maxValue(char1, char2) << endl;
cout << "Max char (three values): " << maxValue(char1, char2, char3) << endl;
return 0;
}
```
Output:
```
Max int (two values): 10
Max int (three values): 10
Max double (two values): 10.5
Max double (three values): 10.5
Max char (two values): B
Max char (three values): C
```

Q3) Create a class template Calculator<T> that performs addition, subtraction, multiplication, and division of two data members of type T. Instantiate it with int and float.

Code:
```
#include <iostream>
using namespace std;
template <typename T>
class Calculator {
private:
T num1;
T num2;
public:
Calculator(T n1, T n2) : num1(n1), num2(n2) {}
T add() {
return num1 + num2;
}
T subtract() {
return num1 - num2;
```

```
}
T multiply() {
return num1 * num2;
}
T divide() {
if (num2 != 0) {
return num1 / num2;
} else {
cout << "Division by zero error!" << endl;
return 0;
}
}
};
int main() {
Calculator<int> intCalc(10, 5);
cout << "Int Addition: " << intCalc.add() << endl;
cout << "Int Subtraction: " << intCalc.subtract() << endl;
cout << "Int Multiplication: " << intCalc.multiply() << endl;
cout << "Int Division: " << intCalc.divide() << endl;
Calculator<float> floatCalc(10.5, 5.2);
cout << "Float Addition: " << floatCalc.add() << endl;
cout << "Float Subtraction: " << floatCalc.subtract() << endl;
cout << "Float Multiplication: " << floatCalc.multiply() << endl;
cout << "Float Division: " << floatCalc.divide() << endl;
return 0;
}
```

Output:

```
Int Addition: 15
Int Subtraction: 5
Int Multiplication: 50
Int Division: 2
Float Addition: 15.7
Float Subtraction: 5.3
Float Multiplication: 54.6
Float Division: 2.01923
```

Discussion:

Templates in C++ allow writing functions and classes that work with any data type, making code reusable. Function templates and class templates help reduce repetition. The STL provides ready-made containers, iterators, and algorithms that make working with data easier and safer.

Conclusion:

Templates and the STL help write simple, efficient, and reusable C++ code. They make programming faster and reduce errors by supporting generic programming and ready-to-use
tools.