

Object Oriented Programming C++

Chapter: Templates

Exam Oriented Questions and Answers (Set 2)

Contents: Part A: Short Answer Questions. Part B: Broad (Long) Answer Questions. Part C: Coding Questions with Programs and Output. All questions in this set are different from Set 1.

Part A: Short Answer Questions

Q1. What is a parameterized type?

Ans: A parameterized type is a type (class or function) in which the data type is passed as a parameter. Templates are called parameterized types because the actual data type is supplied when the template is used, for example `Stack<int>`.

Q2. What is meant by compile-time polymorphism in templates?

Ans: In templates, the compiler decides at compile time which version of the function or class to generate and use, depending on the data type. This is called compile-time (static) polymorphism. It has no run-time overhead, unlike virtual functions.

Q3. Can a template have more than one type parameter?

Ans: Yes. Multiple parameters are separated by commas, for example `template <class T1, class T2>`. Each parameter is replaced by its own data type during instantiation.

Q4. How can the data type be specified explicitly while calling a function template?

Ans: The type is written in angle brackets after the function name, for example `maximum<double>(5, 2.5)`. This is useful when the compiler cannot deduce the type or when the arguments have different types.

Q5. Can templates be used with user-defined types?

Ans: Yes. A template works with a class type provided that the class supports all operations used inside the template. For example, to use `a > b` in a template, the class must overload the `>` operator.

Q6. Differentiate between function overloading and function templates.

Ans: In function overloading the programmer writes a separate function body for each type. In a function template only one body is written and the compiler generates the versions. Overloading is suitable when the logic differs for each type, and templates are suitable when the logic is the same.

Q7. Differentiate between a function template and a template function.

Ans: A **function template** is the generic definition written by the programmer. A **template function** is the actual function generated by the compiler from the template for a specific data type.

Q8. What is the purpose of `T()` in a template?

Ans: `T()` creates a default-initialized value of type `T`. It gives 0 for numeric types, the null character for `char`, and calls the default constructor for class types. It is often used to return a default value, for example from `pop()` of an empty stack.

Q9. Is the template code checked before it is instantiated?

Ans: Only the basic syntax is checked when the template is defined. Errors related to the operations used on `T` are detected only when the template is instantiated with a particular type.

Q10. What is partial specialization?

Ans: Partial specialization gives a special definition of a class template for a subset of its parameters, for example `template <class T> class Data<T, int>`. It is allowed for class templates but not for function templates.

Q11. What happens to static members of a class template?

Ans: Each instantiation of a class template has its own separate copy of the static members. For example, `Counter<int>::count` and `Counter<double>::count` are different variables.

Q12. Can a class template be inherited?

Ans: Yes. A class template can be a base class of another class template, for example `Derived<T> : public Base<T>`. A normal class can also be derived from a particular instance such as `Base<int>`.

Q13. What is the scope of a template parameter name?

Ans: The name of a template parameter, such as `T`, is valid only inside the template definition where it is declared. It cannot be used outside of it.

Q14. Are `Stack<int>` and `Stack<float>` the same class?

Ans: No. They are two different and unrelated classes generated from the same class template. An object of one type cannot be assigned to the other directly.

Q15. Can a template parameter be used to pass a value such as the size of an array?

Ans: Yes, by using a non-type template parameter, for example `template <class T, int size> class Array`. The value given must be a compile-time constant.

Q16. Can a member function template be virtual?

Ans: No. A member function template cannot be declared virtual, because the compiler would not be able to build a fixed virtual table for all possible instantiations.

Part B: Broad (Long) Answer Questions

Q1. Explain the need for templates. Show how a program without templates differs from one with templates.

Ans: Many operations, such as finding the larger of two numbers, have the same logic for every data type. Without templates the programmer must write a separate function for each type, which wastes effort, increases code size and makes maintenance difficult.

Without templates (function overloading):

```
int maximum(int a, int b)      { return (a > b) ? a : b; }
float maximum(float a, float b) { return (a > b) ? a : b; }
char maximum(char a, char b)   { return (a > b) ? a : b; }
```

With a template (one definition):

```
template <class T>
T maximum(T a, T b)
{
    return (a > b) ? a : b;
}
```

- The template version is shorter and avoids repeated code.
- It automatically supports new types, including user-defined types, without any change.
- A bug fix in the template applies to all types at once.
- The compiler still checks types, so it is safe, unlike macros.

Q2. Differentiate between function overloading and function templates.

Point	Function Overloading	Function Template
Definition	Several functions with the same name but different parameter lists.	A single generic function definition.
Code written	A separate body for each type.	One body for all types.
Code size	More source code.	Less source code.
Logic	Can be different for each version.	Same logic for all types.
Who creates versions	The programmer.	The compiler (instantiation).
Best used when	Behaviour differs for different types.	Behaviour is the same for different types.

Q3. Explain how templates can be used with user-defined data types. Give an example.

Ans: A template can be used with a class type provided the class supports every operation used inside the template. If the template uses > and <<, the class must overload these operators. Otherwise the compiler reports an error at instantiation.

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

```

class Point
{
    int x, y;
public:
    Point(int a = 0, int b = 0) { x = a; y = b; }
    bool operator>(const Point &p) const
    {
        return (x * x + y * y) > (p.x * p.x + p.y * p.y);
    }
    friend ostream &operator<<(ostream &o, const Point &p)
    {
        o << "(" << p.x << "," << p.y << ")";
        return o;
    }
};

template <class T>
T maximum(T a, T b)
{
    return (a > b) ? a : b;
}

int main()
{
    Point p1(1, 2), p2(3, 4);
    cout << maximum(p1, p2) << endl;    // (3,4)
    cout << maximum(7, 3) << endl;      // 7
    return 0;
}

```

The same function template works for `int` and for the user-defined class `Point`, because `Point` overloads `>` and `<<`.

Q4. Explain inheritance with class templates, with an example.

Ans: A class template can act as a base class or a derived class. A derived class template is declared as `template <class T> class Derived : public Base<T>`. In a derived template, members of the base template are accessed using `this->` (or `Base<T>::`), because the compiler does not look into a dependent base class automatically.

```

template <class T>
class Base
{
protected:
    T value;
public:
    Base(T v) { value = v; }
};

template <class T>
class Derived : public Base<T>
{
public:
    Derived(T v) : Base<T>(v) { }
    void show() { cout << "Value = " << this->value << endl; }
};

int main()
{

```

```

    Derived<int> d1(50);
    Derived<double> d2(7.5);
    d1.show();    // Value = 50
    d2.show();    // Value = 7.5
    return 0;
}

```

- A normal (non-template) class can also be derived from a particular instance, for example `class A : public Base<int>.`
- A template class can be derived from an ordinary class as well.

Q5. Explain static members in a class template.

Ans: Each instantiation of a class template is a separate class, so each has its own copy of the static data members. The static member must be defined outside the class using the template header.

```

template <class T>
class Counter
{
public:
    static int count;
    Counter() { count++; }
};

template <class T>
int Counter<T>::count = 0;

int main()
{
    Counter<int> a, b, c;
    Counter<double> d;
    cout << Counter<int>::count << endl;    // 3
    cout << Counter<double>::count << endl; // 1
    return 0;
}

```

Three objects of `Counter<int>` share one counter, while `Counter<double>` has its own counter.

Q6. Explain partial specialization and full specialization of class templates.

Ans:

- **Full (explicit) specialization:** a special version for a complete set of types, written with `template <>`, for example `template <> class Data<int, char>.`
- **Partial specialization:** a special version for some of the parameters, while the others remain generic, for example `template <class T> class Data<T, int>.`
- Partial specialization is available only for class templates. Function templates can only be fully specialized (or overloaded).
- The compiler chooses the most specialized version that matches the types used.

```

template <class T1, class T2>
class Data { public: void show() { cout << "General" << endl; } };

```

```

template <class T>
class Data<T, int> { public: void show() { cout << "Second is int" << endl; } };

template <>
class Data<char, char> { public: void show() { cout << "Both char" << endl; } };

// Data<float, double> -> General
// Data<float, int>    -> Second is int
// Data<char, char>   -> Both char

```

Q7. Differentiate between compile-time polymorphism (templates) and run-time polymorphism (virtual functions).

Point	Templates	Virtual Functions
Binding	At compile time (static).	At run time (dynamic).
Speed	Faster, no overhead.	Slightly slower due to the virtual table.
Mechanism	Instantiation of generic code.	Base class pointer and overriding.
Requires	Types supporting the same operations.	Inheritance relationship between classes.
Code size	May increase (code bloat).	Does not increase per type.
Type known	Must be known at compile time.	Can be decided at run time.

Q8. Explain function templates with multiple parameters and the problem with the return type.

Ans: A function template can use more than one generic type. Each parameter is deduced independently.

```

template <class T, class U>
void display(T a, U b)
{
    cout << a << " - " << b << endl;
}

// display(10, "Hello");    T = int, U = const char*
// display(2.5, 'A');       T = double, U = char

```

If the return type depends on the argument types, it cannot be decided by the return statement alone. For example, in `template <class T, class U> ??? add(T a, U b)` the result may be `int`, `double` or another type. Solutions are: (1) give the return type explicitly as a third template parameter, `add<int, double, double>(...)`, or (2) use `auto` with a trailing return type (C++11) or `auto` alone (C++14).

Q9. Explain class templates with default template arguments.

Ans: A template parameter can be given a default type, which is used when no type is given while creating the object. The empty angle brackets `<>` are written to use the default.

```

template <class T = int>
class Box
{

```

```

    T v;
public:
    Box(T x) { v = x; }
    void show() { cout << "Box: " << v << endl; }
};

int main()
{
    Box<> b1(5);           // T is int (default)
    Box<double> b2(2.5);   // T is double
    b1.show();             // Box: 5
    b2.show();             // Box: 2.5
    return 0;
}

```

Q10. What are the common errors made while using templates? How are they avoided?

Ans:

- **Conflicting types:** `maximum(5, 2.5)` fails because `T` cannot be both `int` and `double`. Give the type explicitly or use two parameters.
- **Missing operator:** using a class with a template that applies `>` or `<<` when the class has not overloaded them. Overload the required operators.
- **Definition in a .cpp file:** the compiler cannot instantiate the template, giving linker errors. Keep the full definition in the header file.
- **Missing template header:** forgetting `template <class T>` before a member function defined outside the class. Repeat the header and write `Class<T>::`.
- **Not writing the type for a class template object:** `Stack s;` is an error before C++17. Write `Stack<int> s;`.
- **Accessing base members in a derived template** without `this->`. Use `this->member`.

Part C: Coding Questions with Programs and Output

C1. Write a function template to find the smallest of three values.

```

#include <iostream>
using namespace std;

template <class T>
T minimum(T a, T b, T c)
{
    T m = a;
    if (b < m) m = b;
    if (c < m) m = c;
    return m;
}

int main()
{
    cout << minimum(8, 3, 5) << endl;
    cout << minimum(2.5, 7.1, 1.2) << endl;
    cout << minimum('x', 'b', 'm') << endl;
    return 0;
}

```

Output:

```
3
1.2
b
```

C2. Write a function template to reverse the elements of an array.

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

template <class T>
void reverseArray(T a[], int n)
{
    for (int i = 0; i < n / 2; i++)
    {
        T temp = a[i];
        a[i] = a[n - 1 - i];
        a[n - 1 - i] = temp;
    }
}

template <class T>
void printArray(T a[], int n)
{
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    cout << endl;
}

int main()
{
    int a[] = {1, 2, 3, 4, 5};
    char c[] = {'a', 'b', 'c'};

    reverseArray(a, 5);
    reverseArray(c, 3);
    printArray(a, 5);
    printArray(c, 3);
    return 0;
}
```

Output:

```
5 4 3 2 1
c b a
```

C3. Write a function template for linear search. It should return the index of the key, or -1 if not found.

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

template <class T>
int linearSearch(T a[], int n, T key)
{
    for (int i = 0; i < n; i++)
        if (a[i] == key)
            return i;
    return -1;
}
```

```

}

int main()
{
    int a[] = {4, 8, 15, 16};
    double b[] = {1.1, 2.2, 3.3};

    cout << "Index: " << linearSearch(a, 4, 15) << endl;
    cout << "Index: " << linearSearch(b, 3, 9.9) << endl;
    return 0;
}

```

Output:

```

Index: 2
Index: -1

```

C4. Write a program to implement a Queue using a class template.

```

#include <iostream>
using namespace std;
#define SIZE 5

template <class T>
class Queue
{
    T arr[SIZE];
    int front, rear;
public:
    Queue() { front = 0; rear = -1; }

    void enqueue(T x)
    {
        if (rear == SIZE - 1)
            cout << "Queue Full" << endl;
        else
            arr[++rear] = x;
    }

    T dequeue()
    {
        if (front > rear)
        {
            cout << "Queue Empty" << endl;
            return T();
        }
        return arr[front++];
    }
};

int main()
{
    Queue<int> q;
    q.enqueue(5);
    q.enqueue(15);
    q.enqueue(25);
    cout << q.dequeue() << endl;
    cout << q.dequeue() << endl;

    Queue<double> d;

```

```

    d.enqueue(1.5);
    cout << d.dequeue() << endl;
    cout << d.dequeue() << endl;
    return 0;
}

```

Output:

```

5
15
1.5
Queue Empty
0

```

C5. Write a class template Number with a function to compare two objects.

```

#include <iostream>
using namespace std;

template <class T>
class Number
{
    T val;
public:
    Number(T v) { val = v; }
    bool isGreater(Number<T> n) { return val > n.val; }
};

int main()
{
    Number<int> a(10), b(7);
    Number<double> c(2.5), d(9.5);

    cout << (a.isGreater(b) ? "First is greater" : "Second is greater") <<
endl;
    cout << (c.isGreater(d) ? "First is greater" : "Second is greater") <<
endl;
    return 0;
}

```

Output:

```

First is greater
Second is greater

```

C6. Write a program that uses a function template with a user-defined class Point (overload > and <<).

```

#include <iostream>
using namespace std;

class Point
{
    int x, y;
public:
    Point(int a = 0, int b = 0) { x = a; y = b; }
    bool operator>(const Point &p) const
    {
        return (x * x + y * y) > (p.x * p.x + p.y * p.y);
    }
}

```

```

        friend ostream &operator<<(ostream &o, const Point &p)
        {
            o << "(" << p.x << "," << p.y << ")";
            return o;
        }
};

template <class T>
T maximum(T a, T b)
{
    return (a > b) ? a : b;
}

int main()
{
    Point p1(1, 2), p2(3, 4);
    cout << maximum(p1, p2) << endl;
    cout << maximum(7, 3) << endl;
    return 0;
}

```

Output:

```

(3,4)
7

```

C7. Write a program to show inheritance of class templates.

```

#include <iostream>
using namespace std;

template <class T>
class Base
{
protected:
    T value;
public:
    Base(T v) { value = v; }
};

template <class T>
class Derived : public Base<T>
{
public:
    Derived(T v) : Base<T>(v) { }
    void show() { cout << "Value = " << this->value << endl; }
};

int main()
{
    Derived<int> d1(50);
    Derived<double> d2(7.5);
    d1.show();
    d2.show();
    return 0;
}

```

Output:

```

Value = 50

```

```
Value = 7.5
```

C8. Write a program to show static members in a class template.

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

template <class T>
class Counter
{
public:
    static int count;
    Counter() { count++; }
};

template <class T>
int Counter<T>::count = 0;

int main()
{
    Counter<int> a, b, c;
    Counter<double> d;
    cout << Counter<int>::count << endl;
    cout << Counter<double>::count << endl;
    return 0;
}
```

Output:

```
3
1
```

C9. Write a function template with two different generic types and call it with different type combinations.

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

template <class T, class U>
void display(T a, U b)
{
    cout << a << " - " << b << endl;
}

int main()
{
    display(10, "Hello");
    display(2.5, 'A');
    display('x', 100);
    return 0;
}
```

Output:

```
10 - Hello
2.5 - A
x - 100
```

C10. Write a program that uses a class template with a default template argument.

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

template <class T = int>
class Box
{
    T v;
public:
    Box(T x) { v = x; }
    void show() { cout << "Box: " << v << endl; }
};

int main()
{
    Box<> b1(5);
    Box<double> b2(2.5);
    b1.show();
    b2.show();
    return 0;
}
```

Output:

```
Box: 5
Box: 2.5
```

C11. Predict the output of the following program.

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

template <class T>
class Box
{
    T v;
public:
    Box(T x) { v = x; }
    T get() { return v; }
};

int main()
{
    Box<int> a(7);
    Box<float> b(2.5f);
    Box<char> c('K');

    cout << a.get() << " " << b.get() << " " << c.get() << endl;
    cout << a.get() + b.get() << endl;
    return 0;
}
```

Output:

```
7 2.5 K
9.5
```

Explanation: Each object uses its own generated class. In `a.get() + b.get()`, the `int` value 7 is converted to `float` and added to 2.5, giving 9.5.

C12. Find the error in the following code and correct it.

```
class Student
{
public:
    int marks;
    Student(int m) { marks = m; }
};

template <class T>
T maximum(T a, T b)
{
    return (a > b) ? a : b;
}

int main()
{
    Student s1(80), s2(90);
    Student best = maximum(s1, s2);
    return 0;
}
```

Error: The template uses the `>` operator, but the class `Student` does not overload it, so instantiation of `maximum<Student>` fails.

Correction: overload the operator inside the class.

```
class Student
{
public:
    int marks;
    Student(int m) { marks = m; }
    bool operator>(const Student &s) const { return marks > s.marks; }
};
```