

# Object Oriented Programming C++

## Chapter: Templates

### *Exam Oriented Questions and Answers*

**Contents:** Part A: Short Answer Questions. Part B: Broad (Long) Answer Questions. Part C: Coding Questions with Programs and Output.

## Part A: Short Answer Questions

---

**Q1.** What is a template in C++?

**Ans:** A template is a feature of C++ that allows a single function or class definition to work with different data types. The data type is passed as a parameter, and the compiler generates the required version when it is used. It is the basis of generic programming.

**Q2.** What is generic programming?

**Ans:** Generic programming is a style of programming in which algorithms and classes are written independent of any particular data type. The actual type is supplied later. In C++, it is achieved using templates.

**Q3.** What are the types of templates in C++?

**Ans:** There are two types: (1) **Function templates**, which create a family of functions, and (2) **Class templates**, which create a family of classes.

**Q4.** Write the syntax of a function template.

**Ans:** `template <class T>` (or `template <typename T>`) is written first, followed by the function definition that uses `T` as a data type, for example `T maximum(T a, T b) { ... }`.

**Q5.** Is there any difference between `class` and `typename` in `template <class T>`?

**Ans:** No. In a template parameter list, the keywords `class` and `typename` have the same meaning. Both declare `T` as a type parameter.

**Q6.** What is template instantiation?

**Ans:** Instantiation is the process by which the compiler generates an actual function or class from a template by replacing the template parameter with a real data type. It happens at compile time when the template is used.

**Q7.** Differentiate between a class template and a template class.

**Ans:** A **class template** is the generic blueprint written by the programmer, for example `template <class T> class Stack`. A **template class** is the actual class generated from it for a specific type, for example `Stack<int>`.

**Q8.** Why are templates better than macros?

**Ans:** Macros only perform text substitution and do no type checking, so they can give wrong results and hard-to-find errors. Templates are processed by the compiler, are type-safe, follow scope rules and give proper error messages.

**Q9.** What is a non-type template parameter?

**Ans:** It is a template parameter that represents a constant value instead of a data type, for example `template <class T, int size>`. The value must be a constant expression known at compile time.

**Q10.** What is template specialization?

**Ans:** Template specialization provides a special definition of a template for a particular data type when the general version is not suitable. It is written using `template <>`.

**Q11.** Can function templates be overloaded?

**Ans:** Yes. A function template can be overloaded by another template with a different parameter list, or by an ordinary function of the same name. If an ordinary function matches exactly, it is preferred over the template.

**Q12.** What happens if `maximum(5, 2.5)` is called for `template <class T> T maximum(T a, T b)`? How can it be corrected?

**Ans:** A compilation error occurs because `T` is deduced as `int` from the first argument and `double` from the second. It can be corrected by giving the type explicitly, `maximum<double>(5, 2.5)`, or by using two template parameters.

**Q13.** Why are templates normally defined in header files?

**Ans:** The compiler must see the complete template definition at the point where it is instantiated. If the definition is hidden in a separate `.cpp` file, the compiler cannot generate the code and a linker error occurs.

**Q14.** What is code bloat?

**Ans:** Code bloat is the increase in executable size caused by templates. Since a separate copy of the code is generated for every data type used, many instantiations make the program larger.

**Q15.** What is a default template argument?

**Ans:** It is a default data type given to a template parameter, for example `template <class T = int>`. If no type is specified when the class is used, the default type is taken.

**Q16.** How is the STL related to templates?

**Ans:** The Standard Template Library is completely based on templates. Containers such as `vector`, `list`, `stack`, `map` and algorithms such as `sort` and `find` are written as templates so that they work with any data type.

## Part B: Broad (Long) Answer Questions

---

**Q1.** What is a function template? Explain with syntax and an example.

**Ans:** A function template is a generic function definition that can work with different data types. The compiler automatically creates the proper version of the function depending on the type of the arguments passed in the call.

**Syntax:**

```
template <class T>
return_type function_name(T arg1, T arg2)
{
    // body
}
```

**Example:**

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

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

int main()
{
    cout << maximum(10, 20) << endl;
    cout << maximum(3.5, 2.1) << endl;
    cout << maximum('a', 'z') << endl;
    return 0;
}
```

**Output:**

```
20
3.5
z
```

**Features:**

- One function definition works for many data types, so code is reused.
- The compiler deduces `T` from the arguments and creates a separate function for each type used.
- Both arguments must have the same type when the same `T` is used for them.
- Function templates are type-safe and are resolved at compile time.

**Q2.** What is a class template? Explain its syntax and how objects are created, with an example.

**Ans:** A class template is a generic class definition in which data members, function parameters and return types can use a generic data type. It is used to create a family of classes for different data types from a single definition.

## Syntax:

```
template <class T>
class ClassName
{
    // members using type T
};

ClassName<int> obj1;    // template class for int
ClassName<float> obj2; // template class for float
```

## Example:

```
template <class T>
class Calculator
{
    T a, b;
public:
    Calculator(T x, T y) { a = x; b = y; }
    T add()      { return a + b; }
    T multiply() { return a * b; }
};

int main()
{
    Calculator<int> c1(4, 5);
    Calculator<float> c2(2.5f, 1.5f);
    cout << c1.add() << " " << c1.multiply() << endl;
    cout << c2.add() << " " << c2.multiply() << endl;
    return 0;
}
```

## Output:

```
9 20
4 3.75
```

Here the single class `Calculator` is used for `int` and `float`. The type must be written in angle brackets while creating the object.

**Q3.** Explain template instantiation. How does the compiler generate code from a template?

**Ans:** A template is only a blueprint. The compiler does not generate any code for it until it is used with a particular data type. The process of generating a real function or class from a template is called instantiation.

- When a function template is called (or a class template object is created), the compiler checks the data types involved.
- It substitutes the actual type in place of the template parameter (for example `T` becomes `int`).
- It generates a new function or class for that type and compiles it like normal code.
- If the same type is used again, the already generated version is reused. A new type causes a new version to be generated.
- For class templates, member functions are instantiated only when they are actually called.

- If the type does not support an operation used in the template (for example `>` on a class without that operator), a compile-time error is produced.

For example, calling `maximum(3, 4)` and `maximum(2.5, 1.5)` makes the compiler produce two functions, `int maximum(int, int)` and `double maximum(double, double)`.

**Q4.** Differentiate between function templates and class templates.

| Point          | Function Template                                       | Class Template                                                    |
|----------------|---------------------------------------------------------|-------------------------------------------------------------------|
| Purpose        | Creates a family of functions.                          | Creates a family of classes.                                      |
| Usage          | Called like a normal function.                          | Used to create objects.                                           |
| Type deduction | Type is usually deduced from the arguments.             | Type must be written explicitly in angle brackets (before C++17). |
| Example        | <pre>template &lt;class T&gt; T maximum(T a, T b)</pre> | <pre>template &lt;class T&gt; class Stack</pre>                   |
| Overloading    | Can be overloaded.                                      | Cannot be overloaded, but can be specialized.                     |
| Instantiation  | When the function is called.                            | When an object of the class is created.                           |

**Q5.** Differentiate between templates and macros.

| Point           | Templates                     | Macros                                                                         |
|-----------------|-------------------------------|--------------------------------------------------------------------------------|
| Processing      | Handled by the compiler.      | Handled by the preprocessor (text substitution).                               |
| Type checking   | Type-safe; types are checked. | No type checking.                                                              |
| Debugging       | Proper error messages.        | Errors are difficult to trace.                                                 |
| Scope           | Follow C++ scope rules.       | Ignore scope rules.                                                            |
| Side effects    | Arguments are evaluated once. | Arguments may be evaluated more than once, for example <code>SQR(x++)</code> . |
| Use for classes | Can define generic classes.   | Cannot define classes.                                                         |

**Q6.** Explain the advantages and limitations of templates.

**Advantages:**

- **Code reusability:** one definition works for many data types.
- **Reduced code size and effort:** no need to write the same logic again and again.
- **Type safety:** errors are caught at compile time, unlike macros and `void *`.
- **Easy maintenance:** a change in the template applies to all types.
- **Compile-time polymorphism:** no run-time overhead.
- **Basis of the STL:** containers and algorithms are written using templates.

**Limitations:**

- Code bloat, because every new type generates a new copy of the code.
- Error messages are long and difficult to understand.

- Definitions must be placed in header files, which can increase compilation time.
- The type used must support all operations used inside the template.

**Q7.** What is template specialization? Explain with examples for a function template and a class template.

**Ans:** Template specialization allows us to give a different implementation of a template for a specific data type, when the general version does not behave correctly or efficiently for that type. It is written using `template <>`.

#### Function template specialization:

```
template <class T>
void show(T x) { cout << "Generic: " << x << endl; }

template <>
void show<char>(char x) { cout << "Character: " << x << endl; }

// show(10);   prints  Generic: 10
// show('A');  prints  Character: A
```

#### Class template specialization:

```
template <class T>
class Printer
{
public:
    void print(T x) { cout << "Value: " << x << endl; }
};

template <>
class Printer<char>
{
public:
    void print(char x) { cout << "Character: " << x << endl; }
};
```

**Q8.** Explain overloading of function templates.

**Ans:** A function template can be overloaded in two ways:

- By another template with a different number or type of parameters, for example `template <class T> void show(T a)` and `template <class T> void show(T a, T b)`.
- By an ordinary (non-template) function with the same name, for example `void show(int a)`.

The compiler follows this order while resolving a call: (1) look for an exact match among ordinary functions, (2) if not found, try to create a matching function from a template, (3) if still not found, try ordinary functions with type conversion. If more than one match is possible, an ambiguity error occurs.

**Q9.** Explain how member functions of a class template are defined outside the class, with an example.

**Ans:** When a member function is defined outside a class template, the template header must be repeated before the function, and the class name must be written with the template parameter before the scope resolution operator, as `ClassName<T>::function()`.

```
template <class T>
class Sample
{
    T value;
public:
    void setValue(T v);
    T getValue();
};

template <class T>
void Sample<T>::setValue(T v)
{
    value = v;
}

template <class T>
T Sample<T>::getValue()
{
    return value;
}
```

**Q10.** Explain class templates with multiple parameters and non-type parameters.

**Ans:** A class template can take more than one parameter, separated by commas. Parameters can be type parameters or non-type parameters (constant values).

```
template <class T1, class T2>
class Pair
{
    T1 first;
    T2 second;
public:
    Pair(T1 a, T2 b) { first = a; second = b; }
    void show() { cout << first << " " << second << endl; }
};

template <class T, int size>
class Array
{
    T a[size];
    // ...
};

// Pair<int, float> p(5, 2.5f);
// Array<int, 10> x;    // array of 10 integers
```

- In `Pair<int, float>`, `T1` becomes `int` and `T2` becomes `float`.
- In `Array<int, 10>`, `size` is a non-type parameter and its value must be a compile-time constant.
- Default arguments are allowed, for example `template <class T = int>`.

## Part C: Coding Questions with Programs and Output

---

**C1.** Write a C++ program to swap two values using a function template. Show it for int, float and char.

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

template <class T>
void swapValues(T &a, T &b)
{
    T temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x = 10, y = 20;
    float p = 1.5f, q = 2.5f;
    char c1 = 'A', c2 = 'B';

    swapValues(x, y);
    swapValues(p, q);
    swapValues(c1, c2);

    cout << x << " " << y << endl;
    cout << p << " " << q << endl;
    cout << c1 << " " << c2 << endl;
    return 0;
}
```

**Output:**

```
20 10
2.5 1.5
B A
```

**C2.** Write a C++ program using a function template to find the largest element of an array.

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

template <class T>
T largest(T arr[], int n)
{
    T max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

int main()
{
    int a[] = {5, 12, 8, 3};
    double b[] = {2.5, 9.1, 4.7};

    cout << "Largest int: " << largest(a, 4) << endl;
}
```

```
    cout << "Largest double: " << largest(b, 3) << endl;
    return 0;
}
```

### Output:

```
Largest int: 12
Largest double: 9.1
```

### C3. Write a C++ program to sort an array of any type using a function template (bubble sort).

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

template <class T>
void bubbleSort(T a[], int n)
{
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (a[j] > a[j + 1])
            {
                T temp = a[j];
                a[j] = a[j + 1];
                a[j + 1] = 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[] = {5, 2, 9, 1};
    float b[] = {3.3f, 1.1f, 2.2f};

    bubbleSort(a, 4);
    bubbleSort(b, 3);
    printArray(a, 4);
    printArray(b, 3);
    return 0;
}
```

### Output:

```
1 2 5 9
1.1 2.2 3.3
```

### C4. Write a C++ program to implement a Stack using a class template.

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

template <class T>
class Stack
```

```

{
    T arr[SIZE];
    int top;
public:
    Stack() { top = -1; }

    void push(T x)
    {
        if (top == SIZE - 1)
            cout << "Stack Overflow" << endl;
        else
            arr[++top] = x;
    }

    T pop()
    {
        if (top == -1)
        {
            cout << "Stack Underflow" << endl;
            return T();
        }
        return arr[top--];
    }

    bool isEmpty() { return top == -1; }
};

int main()
{
    Stack<int> s;
    s.push(10);
    s.push(20);
    s.push(30);
    cout << "Popped: " << s.pop() << endl;
    cout << "Popped: " << s.pop() << endl;

    Stack<char> c;
    c.push('A');
    c.push('B');
    cout << "Popped: " << c.pop() << endl;
    return 0;
}

```

### Output:

```

Popped: 30
Popped: 20
Popped: B

```

### C5. Write a C++ program to demonstrate a class template with two generic data types.

```

#include <iostream>
using namespace std;

template <class T1, class T2>
class Pair
{
    T1 first;
    T2 second;
public:

```

```

    Pair(T1 a, T2 b) { first = a; second = b; }
    void show() { cout << first << " " << second << endl; }
};

int main()
{
    Pair<int, float> p1(5, 2.5f);
    Pair<char, int> p2('A', 65);
    p1.show();
    p2.show();
    return 0;
}

```

### Output:

```

5 2.5
A 65

```

### C6. Write a program to define the member functions of a class template outside the class.

```

#include <iostream>
using namespace std;

template <class T>
class Sample
{
    T value;
public:
    void setValue(T v);
    T getValue();
};

template <class T>
void Sample<T>::setValue(T v)
{
    value = v;
}

template <class T>
T Sample<T>::getValue()
{
    return value;
}

int main()
{
    Sample<int> s1;
    s1.setValue(100);
    cout << s1.getValue() << endl;

    Sample<float> s2;
    s2.setValue(3.14f);
    cout << s2.getValue() << endl;
    return 0;
}

```

### Output:

```

100

```

**C7.** Write a program to show a class template with a non-type parameter (array size) and find the sum of its elements.

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

template <class T, int size>
class MyArray
{
    T a[size];
public:
    void set(int i, T v) { a[i] = v; }
    T sum()
    {
        T s = 0;
        for (int i = 0; i < size; i++)
            s += a[i];
        return s;
    }
};

int main()
{
    MyArray<int, 3> x;
    x.set(0, 10);
    x.set(1, 20);
    x.set(2, 30);
    cout << x.sum() << endl;

    MyArray<float, 2> y;
    y.set(0, 1.5f);
    y.set(1, 2.5f);
    cout << y.sum() << endl;
    return 0;
}
```

**Output:**

```
60
4
```

**C8.** Write a program to show overloading of a function template with an ordinary function, and state the output.

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

template <class T>
void show(T x)
{
    cout << "Template: " << x << endl;
}

void show(int x)
{
    cout << "Ordinary: " << x << endl;
}
```

```
int main()
{
    show(5);
    show(2.5);
    show('A');
    return 0;
}
```

### Output:

```
Ordinary: 5
Template: 2.5
Template: A
```

**Explanation:** For `show(5)` the ordinary function matches exactly, so it is preferred. For `2.5` (double) and `'A'` (char) the template gives an exact match, whereas the ordinary function would need a conversion, so the template is used.

**C9.** Write a program to show explicit specialization of a class template for `char`.

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

template <class T>
class Printer
{
public:
    void print(T x) { cout << "Value: " << x << endl; }
};

template <>
class Printer<char>
{
public:
    void print(char x) { cout << "Character: " << x << endl; }
};

int main()
{
    Printer<int> p1;
    p1.print(10);

    Printer<char> p2;
    p2.print('Z');
    return 0;
}
```

### Output:

```
Value: 10
Character: Z
```

**C10.** Predict the output of the following program.

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

template <class T>
T add(T a, T b)
```

```

{
    return a + b;
}

int main()
{
    cout << add(3, 4) << endl;
    cout << add(2.5, 1.5) << endl;
    cout << add<double>(1, 2) << endl;
    return 0;
}

```

### Output:

```

7
4
3

```

**Explanation:** `add(3, 4)` uses `int`, giving 7. `add(2.5, 1.5)` uses `double`, giving 4.0, which `cout` prints as 4. In `add<double>(1, 2)` the type is given explicitly, so the integers are converted to `double` and the result 3.0 is printed as 3.

**C11.** Find the error in the following code and correct it.

```

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

int main()
{
    cout << maximum(10, 5.5);
    return 0;
}

```

**Error:** `T` is deduced as `int` from the first argument and `double` from the second, so the compiler cannot decide the type of `T` (conflicting types).

**Correction (option 1):** give the type explicitly.

```

cout << maximum<double>(10, 5.5);

```

**Correction (option 2):** use two template parameters.

```

template <class T1, class T2>
auto maximum(T1 a, T2 b) -> decltype(a > b ? a : b)
{
    return (a > b) ? a : b;
}

```