

Object Oriented Programming C++

Chapter: Templates

Notes for Engineering Students

Topics covered: Concepts of Templates, Function and Class Templates, Advantages of Templates

1. Concepts of Templates

1.1 Introduction

A **template** is a feature of C++ that allows us to write a single piece of code that works with different data types. Instead of writing separate functions or classes for `int`, `float`, `double` and so on, the programmer writes one generic definition, and the compiler generates the type-specific versions when needed. This style of programming is called **generic programming**.

For example, a function that finds the larger of two numbers has the same logic for `int`, `float` and `double`. Only the data type changes. Templates remove this repetition by treating the data type itself as a parameter.

1.2 Key Points

- A template is a blueprint or formula for creating functions or classes.
- Templates are a way of achieving **generic programming** in C++.
- The data type is passed as a parameter, hence templates are also called **parameterized types**.
- Templates are expanded at **compile time**. This is known as **compile-time (static) polymorphism**.
- The Standard Template Library (STL) such as `vector`, `list`, `map` is built entirely on templates.
- There are two kinds of templates: **function templates** and **class templates**.

1.3 General Syntax

```
template <class T>
// or
template <typename T>
```

Here, `T` is called the **template type parameter** (a placeholder for an actual data type). The keywords `class` and `typename` mean the same thing in this context. Any valid identifier can be used in place of `T`.

1.4 How Templates Work (Instantiation)

A template by itself is not a real function or class. No code is generated for it until it is used. When the programmer calls a function template or creates an object of a class template with a particular

data type, the compiler creates an actual function or class by replacing the template parameter with that type. This process is called **instantiation**.

- The compiler examines the data type of the arguments.
- It substitutes the type in place of the template parameter.
- It generates a separate copy of the function or class for each different type used.
- The generated version is then compiled like normal code.

1.5 Types of Template Parameters

- **Type parameters:** represent a data type, for example `template <class T>`.
- **Non-type parameters:** represent a constant value such as an integer, for example `template <class T, int size>`.
- **Default template arguments:** a default type can be provided, for example `template <class T = int>`.

1.6 Important Rules

- The keyword `template` must be written before the function or class definition.
- Every template parameter used must appear in the template parameter list.
- Template definitions are normally written in **header files**, because the compiler must see the full definition to instantiate it.
- Templates cannot be separated into a `.h` declaration and a `.cpp` definition in the usual way without extra steps.
- Template code does not cause an error until it is instantiated with a type that does not support the operations used in it. For example, using `>` on a class that has not overloaded it.

2. Function Templates

2.1 Definition

A **function template** is a generic function definition that can operate on different data types. The compiler generates the correct version of the function depending on the type of arguments passed in the call.

2.2 Syntax

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

2.3 Example: Finding the Larger of Two Values

```
#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;    // T becomes int
    cout << maximum(3.5, 2.1) << endl;  // T becomes double
    cout << maximum('a', 'z') << endl;  // T becomes char
    return 0;
}

```

Output:

```

20
3.5
z

```

Explanation: The same function `maximum()` works for `int`, `double` and `char`. The compiler creates three different versions internally.

2.4 Function Template with Multiple Parameters

More than one generic type can be used by separating them with commas.

```

template <class T1, class T2>
void display(T1 a, T2 b)
{
    cout << a << " " << b << endl;
}

// Call:  display(10, 4.5);    // T1 = int, T2 = double

```

2.5 Example: Swapping Two Values

```

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

```

2.6 Overloading Function Templates

- A function template can be overloaded by another template with a different number of parameters.
- It can also be overloaded by an ordinary (non-template) function with the same name.
- When a call is made, the compiler first looks for an exact match among ordinary functions. If found, the ordinary function is called. Otherwise the template is used.

2.7 Template Specialization (Explicit)

Sometimes a different behaviour is needed for a particular data type. In that case an **explicit specialization** is written using `template <>`.

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

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

2.8 Important Points about Function Templates

- Both arguments must be of the same type if the same template parameter `T` is used for both. A call such as `maximum(5, 2.5)` gives an error because `T` cannot be both `int` and `double`.
- The type can be given explicitly: `maximum<double>(5, 2.5);`
- Each instantiation creates a separate function, so every instance has its own copy of static variables.
- Template functions are type-safe, unlike macros.

3. Class Templates

3.1 Definition

A **class template** is a generic class definition in which the data members, member function arguments and return types can use a generic data type. It is used to create a family of classes that differ only in the data types they handle. Examples are a stack of integers, a stack of floats and a stack of characters, all from one class definition.

3.2 Syntax

```
template <class T>
class ClassName
{
    // data members of type T
    // member functions using type T
};
```

3.3 Creating Objects of a Class Template

The data type must be written in angle brackets when the object is created.

```
ClassName<int> obj1;
ClassName<float> obj2;
```

A class created from a class template is called a **template class**. The class template is the blueprint, and the template class is the actual class generated from it.

3.4 Example: Simple Class Template

```
#include <iostream>
```

```

using namespace std;

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

```

3.5 Defining Member Functions Outside the Class

When a member function of a class template is defined outside the class, the template header must be repeated and the scope resolution operator must use the class name with the template parameter.

```

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;
}

```

3.6 Example: Stack Class Template

```
template <class T>
class Stack
{
    T arr[50];
    int top;
public:
    Stack() { top = -1; }
    void push(T x) { arr[++top] = x; }
    T pop()      { return arr[top--]; }
    bool isEmpty() { return top == -1; }
};

// Stack<int> s1;   Stack<char> s2;   Stack<double> s3;
```

3.7 Class Template with Multiple Parameters

```
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; }
};

// Pair<int, float> p(5, 2.5f);
```

3.8 Non-Type Template Parameters and Default Arguments

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

// Array<int, 10> x;   creates an array of 10 integers
```

- Non-type arguments must be constant expressions known at compile time.
- Default type arguments can be given, for example `template <class T = int>`.

3.9 Class Template Specialization

A class template can be specialized for a specific data type when that type needs different handling.

```
template <>
class Sample<char>
{
    // special version for char
};
```

3.10 Important Points about Class Templates

- Member functions of a class template are themselves function templates.

- Member functions are instantiated only when they are actually used.
- Class templates support inheritance. A template class can be derived from another template class or from an ordinary class.
- Friend functions and static members can also be used. Each instantiation has its own copy of static members.

3.11 Function Template vs Class Template

- A function template creates a family of **functions**. A class template creates a family of **classes**.
- In a function template, the type is usually deduced from the arguments. In a class template, the type must be written explicitly in angle brackets (before C++17).
- A function template is called like a normal function. A class template is used to create objects.
- Function templates can be overloaded. Class templates cannot be overloaded, but can be specialized.

4. Advantages of Templates

- **Code reusability:** One definition is written and reused for many data types, which avoids duplicate code.
- **Less code and less effort:** The programmer does not need to write separate functions or classes for each type, so program size is smaller.
- **Type safety:** The compiler checks types at compile time. This is safer than C-style macros and `void *` pointers, which do not perform type checking.
- **Easy maintenance:** A change or bug fix made in the template applies automatically to all types.
- **Better than macros:** Macros are simple text substitution without type checking. Templates are understood by the compiler and give better error detection.
- **Compile-time polymorphism:** The type is resolved at compile time, so there is no run-time overhead of virtual function calls.
- **Flexibility and generic programming:** The same algorithm works for built-in types and user-defined types.
- **Foundation of the STL:** Containers such as `vector`, `list`, `stack`, `queue` and algorithms such as `sort` and `find` are all written using templates.
- **Improved performance:** Since code is generated for the exact type, no type conversion or run-time type checking is needed.

Limitations (for understanding)

- Code bloat. Every new type creates another copy of the code, so executable size can increase.
- Error messages from templates are often long and difficult to read.

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

5. Multiple Choice Questions (15)

Q1. Templates in C++ are used to achieve:

- (a) Data hiding (b) Generic programming (c) Inheritance (d) Dynamic binding

Q2. Which keyword is used to define a template?

- (a) generic (b) type (c) template (d) class

Q3. In `template <class T>`, the word `class` can be replaced by:

- (a) struct (b) object (c) typename (d) define

Q4. Templates are resolved at:

- (a) Run time (b) Compile time (c) Link time only (d) Load time

Q5. The process of creating an actual function or class from a template is called:

- (a) Overloading (b) Instantiation (c) Overriding (d) Encapsulation

Q6. Which of the following is a correct function template declaration?

- (a) `template T add(T a, T b);` (b) `template <class T> T add(T a, T b);`
 (c) `template <T> T add(T a, T b);` (d) `T template add(T a, T b);`

Q7. What happens when `maximum(5, 2.5)` is called for `template <class T> T maximum(T a, T b)?`

- (a) Returns 5 (b) Returns 2.5 (c) Compilation error because of conflicting types (d) Runtime error

Q8. How do we create an object of the class template `Stack` for integers?

- (a) `Stack s;` (b) `Stack(int) s;` (c) `Stack<int> s;` (d) `int Stack s;`

Q9. A class generated from a class template is called a:

- (a) Template class (b) Abstract class (c) Friend class (d) Base template

Q10. Which of the following is NOT an advantage of templates?

- (a) Code reusability (b) Type safety (c) Reduced executable size in all cases (d) Easy maintenance

Q11. Which syntax is used for explicit specialization?

- (a) `template <class T>` (b) `template <>` (c) `specialize <>` (d) `template <T>`

Q12. Templates are usually defined in:

- (a) Header files (b) Only .cpp files (c) Library files only (d) Object files

Q13. A non-type template parameter must be:

- (a) A variable (b) A constant expression (c) A string input at run time (d) A pointer to a function only

Q14. The Standard Template Library (STL) is mainly built using:

- (a) Macros (b) Templates (c) Pointers only (d) Virtual functions

Q15. If an ordinary function and a function template both match a call exactly, which one is called?

- (a) The template (b) The ordinary function (c) Both (d) Compiler error

Answer Key

1. (b) Generic programming
2. (c) template
3. (c) typename
4. (b) Compile time
5. (b) Instantiation
6. (b) `template <class T> T add(T a, T b);`
7. (c) Compilation error because of conflicting types
8. (c) `Stack<int> s;`
9. (a) Template class
10. (c) Reduced executable size in all cases
11. (b) `template <>`
12. (a) Header files
13. (b) A constant expression
14. (b) Templates
15. (b) The ordinary function

6. Broad Questions (10)

1. What is a template in C++? Explain the concept of generic programming and how templates support it.
2. Explain the syntax of a function template. Write a program using a function template to find the largest of two numbers of any data type.
3. Explain the process of template instantiation. How does the compiler generate code from a template?
4. Write a C++ program to swap two values using a function template and show it working with `int` and `float` types.
5. What is a class template? Explain its syntax and show how objects of a class template are created.
6. Write a C++ program to implement a `Stack` class using a class template with `push()`, `pop()` and `isEmpty()` functions.
7. Explain how member functions of a class template are defined outside the class. Give a suitable example.
8. Explain class templates with multiple parameters and non-type parameters with examples.

9. Differentiate between function templates and class templates. Also differentiate between templates and macros.
10. List and explain the advantages and limitations of templates in C++.