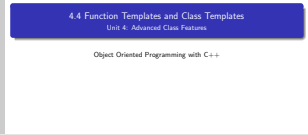
4.4 Function Templates and Class Templates

Unit 4: Advanced Class Features

Object Oriented Programming with C++



Lecture 36: Function Templates and Class Templates

- Course: Object Oriented Programming with C++
- Unit 4: Advanced Class Features
- Topic: 4.4 Function Templates and Class Templates

Lecture 36: Function Templates and Class Templates
- Course: Object Oriented Programming with C++
- Unit 4: Advanced Class Features
- Topic: 4.4 Function Templates and Class Templates

Welcome everyone to Lecture 36. Today we are diving into one of the most powerful features of C++: Templates. Templates allow us to write generic, reusable code, paving the way for the Standard Template Library (STL). By the end of this session, you'll know how to write functions and classes that can seamlessly operate on any data type.

The Problem: Code Duplication

- Suppose we need a function to find the maximum of two numbers.
- For integers: int max(int a, int b) { return (a ¿ b) ? a : b; }
- For floats: float max(float a, float b) { return (a ¿ b) ? a : b; }
- For chars: char max(char a, char b) { return (a ¿ b) ? a : b; }
- Issue: The logic is identical, but we have to write and maintain three separate functions.
- Overloading works, but it scales poorly for many types.

The Problem: Code Duplication
- Suppose we need a function to find the maximum of two numbers.
- For integers: int max(int a, int b) { return (a ¿ b) ? a : b; }
- For floats: float max(float a, float b) { return (a ¿ b) ? a : b; }
- For chars: char max(char a, char b) { return (a ¿ b) ? a : b; }
- Issue: The logic is identical, but we have to write and maintain three separate functions.
- Overloading works, but it scales poorly for many types.

Let's start with a motivating example. Imagine you want to write a simple function that returns the larger of two values. In a strongly typed language like C++, you must specify the data types. So, you write one for integers. But then you need it for floats, and then characters. The logic '(a ¿ b) ? a : b' doesn't change, but you are forced to duplicate the code. Function overloading solves the naming issue, but not the maintenance nightmare of duplicated logic. What if we could write the logic just once?

Introduction to Generic Programming

- Generic Programming: A paradigm where algorithms are written in terms of types that are specified later.
- Focuses on writing code that works with any data type without modification.
- In C++, generic programming is implemented using 'Templates'.
- Templates allow variables, functions, and classes to operate with generic types.

Introduction to Generic Programming
- Generic Programming: A paradigm where algorithms are written in terms of types that are specified later.
- Focuses on writing code that works with any data type without modification.
- In C++, generic programming is implemented using 'Templates'.
- Templates allow variables, functions, and classes to operate with generic types.

This is where generic programming comes in. Generic programming is a style of computer programming in which algorithms are written in terms of to-be-specified-later types that are then instantiated when needed for specific types provided as parameters. In C++, we achieve this using a feature called 'Templates'. Templates are essentially blueprints. You write a single blueprint, and the compiler generates the specific implementations for you based on the types you actually use.

What are Function Templates?

- A Function Template is a blueprint for creating a family of overloaded functions.
- It defines a generalized mathematical or algorithmic behavior.
- The compiler automatically generates a specific version of the function based on the arguments passed.
- Significantly reduces code size and improves maintainability.

What are Function Templates?
- A Function Template is a blueprint for creating a family of overloaded functions.
- It defines a generalized mathematical or algorithmic behavior.
- The compiler automatically generates a specific version of the function based on the arguments passed.
- Significantly reduces code size and improves maintainability.

A function template isn't an actual function; it's a recipe to create functions. When you define a function template, you leave the data types as variables. When you call the function later in your code, the C++ compiler looks at the arguments you passed, deduces their types, and automatically writes a custom version of the function just for those types. This eliminates code duplication and ensures that if you need to fix a bug in the logic, you only have to fix it in one place.

Function Template Syntax

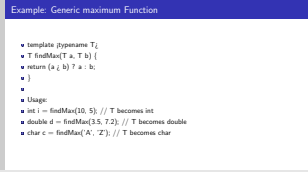
- Keyword 'template' followed by template parameters inside angle brackets ¡ ¿.
- Syntax:
- template ¡class T¿
- T functionName(T param1, T param2) {
- // function body
- }
- 'class' or 'typename' can be used interchangeably here.
- 'T' is a placeholder for the data type.

Function Template Syntax
- Keyword 'template' followed by template parameters inside angle brackets ¡ ¿.
- Syntax:
- template ¡class T¿
- T functionName(T param1, T param2) {
- // function body
- }
- 'class' or 'typename' can be used interchangeably here.
- 'T' is a placeholder for the data type.

Let's look at the syntax. We start with the keyword 'template', telling the compiler that what follows is a template. Then, in angle brackets, we define the template parameters. 'class T' or 'typename T' tells the compiler that 'T' is a placeholder for an actual data type that will be provided later. You can use any name instead of 'T', but 'T' is the universal standard for 'Type'. Inside the function body, you use 'T' wherever you would normally put a specific data type like int or double.

Example: Generic maximum Function

- template ¡typename T¿
- T findMax(T a, T b) {
- return (a ¿ b) ? a : b;
- }
-
- Usage:
- int i = findMax(10, 5); // T becomes int
- double d = findMax(3.5, 7.2); // T becomes double
- char c = findMax('A', 'Z'); // T becomes char



Here is our earlier problem solved using a template. Notice how clean it is. We write 'findMax' exactly once. When we call findMax(10, 5), the compiler sees two integers. It seamlessly replaces 'T' with 'int' and compiles an integer version of the function. When we pass 3.5 and 7.2, it generates a double version. The compiler does the heavy lifting, saving us from writing overloaded functions manually.

How it Works: Template Instantiation

- Templates do not consume memory until they are called.
- Instantiation: The process where the compiler generates a specific function from the template.
- Implicit Instantiation: Compiler deduces the type automatically (e.g., findMax(5, 10)).
- Explicit Instantiation: Programmer specifies the type (e.g., findMax¡int¿(5, 10)).
- Each instantiated version is a separate function in the final binary.

How it Works: Template Instantiation
- Templates do not consume memory until they are called.
- Instantiation: The process where the compiler generates a specific function from the template.
- Implicit Instantiation: Compiler deduces the type automatically (e.g., findMax(5, 10)).
- Explicit Instantiation: Programmer specifies the type (e.g., findMax¡int¿(5, 10)).
- Each instantiated version is a separate function in the final binary.

It's crucial to understand what happens under the hood. Writing a template doesn't create any executable code immediately. Code is only generated when you call the template function. This is called 'instantiation'. Usually, the compiler is smart enough to guess the type based on the arguments—this is Implicit Instantiation. Sometimes, you might want to be explicit, like findMax¡double¿(5, 5.5). Note that because the compiler generates a new function for every unique type used, excessive use of templates with many different types can increase your compiled binary size, a phenomenon known as 'code bloat'.

Function Templates with Multiple Parameters

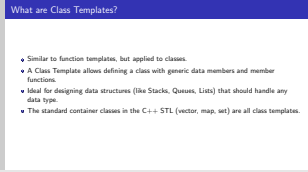
- Templates are not restricted to a single type parameter.
- You can define templates with multiple generic types.
- Syntax: template ¡class T1, class T2¿
- void display(T1 a, T2 b) {
- cout ¡¡ a ¡¡ " and " ¡¡ b ¡¡ endl;
- }
- Usage: display(10, "Hello"); // T1 is int, T2 is const char*

Function Templates with Multiple Parameters
- Templates are not restricted to a single type parameter.
- You can define templates with multiple generic types.
- Syntax: template ¡class T1, class T2¿
- void display(T1 a, T2 b) {
- cout ¡¡ a ¡¡ " and " ¡¡ b ¡¡ endl;
- }
- Usage: display(10, "Hello"); // T1 is int, T2 is const char*

What if a function needs to handle two different types at the same time? You just add more parameters to the template declaration, separated by commas. In this example, 'T1' and 'T2' are independent placeholders. When we call display with an integer and a string, T1 is deduced as an int, and T2 as a string. This provides massive flexibility in how we design utility functions.

What are Class Templates?

- Similar to function templates, but applied to classes.
- A Class Template allows defining a class with generic data members and member functions.
- Ideal for designing data structures (like Stacks, Queues, Lists) that should handle any data type.
- The standard container classes in the C++ STL (vector, map, set) are all class templates.



Now we move from functions to classes. A Class Template is a blueprint for creating classes. Think about a Stack data structure. The logic for pushing and popping items is exactly the same whether it's a stack of integers, a stack of strings, or a stack of custom Employee objects. Class templates allow us to write that Stack class once and use it for anything. In fact, this is exactly how the C++ Standard Template Library, or STL, is built.

Class Template Syntax

- Declared by prefixing the class definition with 'template ¡class T¿'.
- template ¡class T¿
- class ClassName {
- T memberData;
- public:
- ClassName(T arg) : memberData(arg) {}
- T getMemberData();
- };
- Member function definition outside the class requires the template prefix as well.

Class Template Syntax
- Declared by prefixing the class definition with 'template ¡class T¿'.
- template ¡class T¿
- class ClassName {
- T memberData;
- public:
- ClassName(T arg) : memberData(arg) {}
- T getMemberData();
- };
- Member function definition outside the class requires the template prefix as well.

The syntax for a class template is very similar to a function template. You prefix the class keyword with 'template ¡class T¿'. Inside the class, 'T' becomes a valid type. You can use it to declare variables, return types, and function arguments. One important syntax quirk in C++: if you define a member function outside the class declaration, you must re-declare the template prefix above the function definition, and attach '¡T¿' to the class name scope resolution.

Example: Generic Pair Class

- template ¡class T1, class T2¿
- class Pair {
- T1 first;
- T2 second;
- public:
- Pair(T1 a, T2 b) : first(a), second(b) {}
- void display() {
- cout ¡¡ "(" ¡¡ first ¡¡ ", " ¡¡ second ¡¡ ")" ¡¡ endl;
- }
- };

Example: Generic Pair Class
template ¡class T1, class T2¿
class Pair {
T1 first;
T2 second;
public:
Pair(T1 a, T2 b) : first(a), second(b) {}
void display() {
cout ¡¡ "(" ¡¡ first ¡¡ ", " ¡¡ second ¡¡ ")" ¡¡ endl;
}
};

Let's look at a concrete example. Here is a 'Pair' class. It holds two values, which might be of different types, so we use template ¡class T1, class T2¿. We declare 'first' of type T1, and 'second' of type T2. The constructor takes these two generic types, and the display method prints them. This single piece of code can replace dozens of customized pair classes.

Instantiating a Class Template

- Unlike function templates, class templates usually require EXPLICIT instantiation.
- You must specify the data type in angle brackets when creating an object.
- Usage for the Pair class:
- Pair¡int, double¿ obj1(10, 3.14);
- Pair¡string, int¿ obj2("Age", 25);
- obj1.display(); // Outputs: (10, 3.14)

Instantiating a Class Template
- Unlike function templates, class templates usually require EXPLICIT instantiation.
- You must specify the data type in angle brackets when creating an object.
- Usage for the Pair class:
- Pair¡int, double¿ obj1(10, 3.14);
- Pair¡string, int¿ obj2("Age", 25);
- obj1.display(); // Outputs: (10, 3.14)

When it comes to using the class template, there is a key difference from function templates. Historically, and in most common use cases, the compiler cannot deduce class template types just from constructor arguments. You must explicitly tell the compiler what types to use by providing them in angle brackets when you declare the object. 'Pair¡int, double¿' tells the compiler to generate a specific Pair class where T1 is int and T2 is double, and then create an object of that specific class.

Defining Member Functions Outside the Class

- When defining class template methods outside the class body:
- 1. Prefix with the template declaration.
- 2. Add the template argument list ¡T¿ to the class scope.
- Example:
- template ¡class T¿
- T MyClass¡T¿::myMethod() {
- // implementation
- }

Defining Member Functions Outside the Class
- When defining class template methods outside the class body:
- 1. Prefix with the template declaration.
- 2. Add the template argument list ¡T¿ to the class scope.
- Example:
- template ¡class T¿
- T MyClass¡T¿::myMethod() {
- // implementation
- }

This is a common stumbling block for students. If you define your methods inside the class body, it looks like a normal class. But if you separate your declaration and definition—which is best practice—the outside definition looks quite complex. You have to remind the compiler, 'Hey, this function belongs to a template class'. So you write template ¡class T¿, then the return type, then MyClass¡T¿:: to indicate the scope, and finally the function name.

Non-Type Template Parameters

- Templates can also take standard data types (like int) as parameters, not just types.
- These act as compile-time constants.
- Syntax: template ¡class T, int size¿
- class Array {
- T arr[size]; // size is a constant defined at compile time
- };
- Usage: Array¡int, 50¿ myArr; // creates an array of 50 ints

Non-Type Template Parameters
- Templates can also take standard data types (like int) as parameters, not just types.
- These act as compile-time constants.
- Syntax: template ¡class T, int size¿
- class Array {
- T arr[size]; // size is a constant defined at compile time
- };
- Usage: Array¡int, 50¿ myArr; // creates an array of 50 ints

Templates aren't limited to just substituting types. You can also pass actual values, known as non-type template parameters. A classic example is a static Array class. Here, 'int size' is a non-type parameter. When you create an object 'Array¡int, 50¿', the compiler essentially hardcodes the number 50 into that specific class generation. This is evaluated entirely at compile time, making it incredibly fast and useful for fixed-size memory allocation.

Default Template Arguments

- You can provide default types or values for template parameters, similar to default function arguments.
- template ¡class T = int, int size = 10¿
- class Array { ... };
- Usage:
- Array¡¿ arr1; // Uses ¡int, 10¿
- Array¡double¿ arr2; // Uses ¡double, 10¿
- Useful for defining standard behaviors while retaining flexibility.

Default Template Arguments
- You can provide default types or values for template parameters, similar to default function arguments.
- template ¡class T = int, int size = 10¿
- class Array { ... };
- Usage:
- Array¡¿ arr1; // Uses ¡int, 10¿
- Array¡double¿ arr2; // Uses ¡double, 10¿
- Useful for defining standard behaviors while retaining flexibility.

Just like standard functions can have default arguments, templates can have default arguments. In this example, if a user just types Array¡¿, the compiler will assume they want an array of integers of size 10. If they type Array¡double¿, it assumes double and 10. This is highly utilized in the C++ Standard Library to provide sensible defaults while allowing power users to customize memory allocators and comparators.

Summary & Best Practices

- Templates enable Generic Programming, reducing code duplication.
- Function Templates act as blueprints for generating overloaded functions.
- Class Templates act as blueprints for generating classes (crucial for data structures).
- Instantiation happens at compile-time; improper use can lead to 'code bloat'.
- Template definitions typically must reside entirely in header files (.h), not in source files (.cpp).

Summary & Best Practices
- Templates enable Generic Programming, reducing code duplication.
- Function Templates act as blueprints for generating overloaded functions.
- Class Templates act as blueprints for generating classes (crucial for data structures).
- Instantiation happens at compile-time; improper use can lead to 'code bloat'.
- Template definitions typically must reside entirely in header files (.h), not in source files (.cpp).

To summarize, templates are a cornerstone of modern C++. They allow you to write 'Don't Repeat Yourself' (DRY) code. Function templates handle algorithms, and class templates handle data structures. One vital practical note before we finish: unlike standard classes where you put declarations in .h and definitions in .cpp, template code usually must be entirely contained within the header file. Because the compiler needs the template blueprint at the exact moment of instantiation in another file, separating them causes linker errors.