4.4 Function Templates and Class Templates

Unit 4: Advanced Class Features

Object Oriented Programming with C++

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

Function Templates and Class Templates

- Course: Object Oriented Programming with C++ (DI05011021)
- Unit 4: Advanced Class Features
- Lecture 32

Function Templates and Class Templates
- Course: Object Oriented Programming with C++ (DI05011021)
- Unit 4: Advanced Class Features
- Lecture 32

Welcome back to Object Oriented Programming with C++. Today, in Lecture 32, we are diving into one of the most powerful features of C++: Templates. This falls under Unit 4, Advanced Class Features. We will cover both function and class templates, which form the foundation of generic programming in C++.

The Need for Generic Programming

- Problem: Writing multiple functions that do the exact same thing but for different data types.
- Example: A 'swap' function for int, double, char, etc.
- Solution without templates: Function Overloading. This leads to code duplication and maintenance headaches.
- Generic Programming: Writing code that works with any data type.

The Need for Generic Programming
- Problem: Writing multiple functions that do the exact same thing but for different data types.
- Example: A 'swap' function for int, double, char, etc.
- Solution without templates: Function Overloading. This leads to code duplication and maintenance headaches.
- Generic Programming: Writing code that works with any data type.

Let's start by understanding why we need templates. Imagine you need to write a function to swap two variables. You might write one for integers. But then you need to swap doubles, so you overload the function. Then characters. Suddenly, you have three almost identical functions. This violates the DRY (Don't Repeat Yourself) principle. Generic programming solves this by allowing us to write code that is independent of specific data types.

What are Templates?

- A template is a blueprint or formula for creating a generic class or a function.
- It enables you to define a function or a class with placeholders for data types.
- The compiler generates the specific code at compile-time based on the types used.
- Two main types:
- 1. Function Templates
- 2. Class Templates

What are Templates?
- A template is a blueprint or formula for creating a generic class or a function.
- It enables you to define a function or a class with placeholders for data types.
- The compiler generates the specific code at compile-time based on the types used.
- Two main types:
- 1. Function Templates
- 2. Class Templates

A template is exactly what it sounds like - a blueprint. Just as a class is a blueprint for an object, a template is a blueprint for a class or a function. We use placeholders instead of actual data types like int or float. When we actually call the function or create an object of the class, we tell the compiler what type to use, and it generates the correct code for us right then and there. We'll look at both function and class templates today.

Function Templates: Syntax

- Defined using the 'template' keyword followed by template parameters enclosed in angle brackets '¡ ¿'.
- Syntax:
- template ¡class T¿ OR template ¡typename T¿
- return_type function_name(parameters) {
- // function body
- }
- T is a placeholder type name.

Function Templates: Syntax
- Defined using the 'template' keyword followed by template parameters enclosed in angle brackets '¡ ¿'.
- Syntax:
- template ¡class T¿ OR template ¡typename T¿
- return_type function_name(parameters) {
- // function body
- }
- T is a placeholder type name.

Here is the basic syntax for a function template. We start with the keyword 'template', followed by angle brackets. Inside, we declare our type parameters using the keyword 'class' or 'typename'—they mean exactly the same thing in this context. 'T' is the conventional name for a generic type, but you can use any valid identifier. Then, we write our function definition normally, but we use 'T' wherever we would normally use a specific type.

Function Template Example: Swap

- template ¡typename T¿
- void mySwap(T& a, T& b) {
- T temp = a;
- a = b;
- b = temp;
- }

Function Template Example: Swap
- template ¡typename T¿
- void mySwap(T& a, T& b) {
- T temp = a;
- a = b;
- b = temp;
- }

Let's look at the classic example: swapping two values. Here, 'T' is our placeholder type. We take two parameters, 'a' and 'b', both references to type 'T'. Inside the function, we declare a temporary variable of type 'T' to perform the swap. This single template can now swap integers, floats, characters, or even user-defined objects, provided the assignment operator is valid for them.

Using a Function Template

- int x = 10, y = 20;
- mySwap(x, y); // Compiler deduces T as int
- double c = 1.5, d = 2.5;
- mySwap(c, d); // Compiler deduces T as double
- // Explicit instantiation:
- mySwap¡int¿(x, y);

Using a Function Template
- int x = 10, y = 20;
- mySwap(x, y); // Compiler deduces T as int
- double c = 1.5, d = 2.5;
- mySwap(c, d); // Compiler deduces T as double
- // Explicit instantiation:
- mySwap¡int¿(x, y);

How do we use our mySwap template? It's remarkably simple. You just call it like a normal function. When we pass integers 'x' and 'y', the compiler looks at the arguments, deduces that 'T' must be 'int', and secretly generates an int-specific version of the function. This is called template argument deduction. You can also explicitly tell the compiler the type using angle brackets, like mySwap¡int¿(x, y), which is sometimes necessary if deduction fails.

Multiple Template Parameters

- Templates can have more than one type parameter.
- Syntax:
- template ¡class T1, class T2¿
- void printMultiple(T1 a, T2 b) {
- cout ¡¡ a ¡¡ " and " ¡¡ b ¡¡ endl;
- }

Multiple Template Parameters
- Templates can have more than one type parameter.
- Syntax:
- template ¡class T1, class T2¿
- void printMultiple(T1 a, T2 b) {
- cout ¡¡ a ¡¡ " and " ¡¡ b ¡¡ endl;
- }

Templates aren't limited to just one generic type. If your function needs to handle multiple distinct unknown types, you simply list them in the template parameter list separated by commas. In this example, printMultiple takes two parameters, 'a' of type 'T1' and 'b' of type 'T2'. They can be the same type when you call it, or entirely different types, like an int and a string.

Class Templates

- Similar to function templates, but applied to entire classes.
- Useful for creating generic data structures (e.g., stacks, queues, linked lists, arrays).
- Syntax:
- template ¡class T¿
- class ClassName {
- // Class members using T
- };

Class Templates
- Similar to function templates, but applied to entire classes.
- Useful for creating generic data structures (e.g., stacks, queues, linked lists, arrays).
- Syntax:
- template ¡class T¿
- class ClassName {
- // Class members using T
- };

Now let's move on to Class Templates. The motivation is the same. Imagine writing a Stack class. Do you write one Stack for ints, another for doubles, and another for strings? No, you write one generic Stack template. The syntax is very similar to function templates. You put the template declaration right before the class definition.

Class Template Example: A Generic Pair

- template ¡typename T1, typename T2¿
- class Pair {
- private:
- T1 first;
- T2 second;
- public:
- Pair(T1 a, T2 b) : first(a), second(b) {}
- T1 getFirst() { return first; }
- T2 getSecond() { return second; }
- };

Class Template Example: A Generic Pair
- template ¡typename T1, typename T2¿
- class Pair {
- private:
- T1 first;
- T2 second;
- public:
- Pair(T1 a, T2 b) : first(a), second(b) {}
- T1 getFirst() { return first; }
- T2 getSecond() { return second; }
- };

Here is a simple but useful example of a class template: a 'Pair' class that holds two values, potentially of different types. We use two template parameters, T1 and T2. The private members 'first' and 'second' use these types. The constructor and getter methods also use these placeholder types. This defines a structure that can hold any combination of two types.

Instantiating Class Templates

- Unlike function templates, class templates (usually) require explicit type arguments when creating an object.
- Syntax: ClassName¡Type¿ objectName;
- Example:
- Pair¡int, double¿ p1(10, 3.14);
- Pair¡string, int¿ p2("Age", 25);

Instantiating Class Templates
- Unlike function templates, class templates (usually) require explicit type arguments when creating an object.
- Syntax: ClassName¡Type¿ objectName;
- Example:
- Pair¡int, double¿ p1(10, 3.14);
- Pair¡string, int¿ p2("Age", 25);

When we want to create an object of a class template, we usually have to explicitly specify the types in angle brackets. The compiler cannot always guess them like it can with function arguments (though C++17 introduced Class Template Argument Deduction or CTAD, which helps). But traditionally, and explicitly, we write Pair¡int, double¿ to tell the compiler to generate a Pair class specifically for an int and a double.

Class Templates: Member Functions Defined Outside

- If you define member functions outside the class declaration, you must restate the template declaration.
- Syntax:
- template ¡class T¿
- ReturnType ClassName¡T¿::functionName() { ... }

Class Templates: Member Functions Defined Outside
- If you define member functions outside the class declaration, you must restate the template declaration.
- Syntax:
- template ¡class T¿
- ReturnType ClassName¡T¿::functionName() { ... }

A very common pitfall with class templates is defining member functions outside the class body. Every time you define a member function outside, you have to remind the compiler that it's part of a template class. You must precede the function definition with the 'template ¡class T¿' line, and you must include '¡T¿' after the class name in the scope resolution operator. It looks verbose, but it's required.

External Definition Example

- template ¡typename T¿
- class Box {
- T item;
- public:
- void setItem(T i);
- };
-
- template ¡typename T¿
- void Box¡T¿::setItem(T i) {
- item = i;
- }

External Definition Example
template ¡typename T¿
class Box {
T item;
public:
void setItem(T i);
};
template ¡typename T¿
void Box¡T¿::setItem(T i) {
item = i;
}

Here's that syntax in action. We declare the Box class template and its setItem method. Then, outside the class, we define setItem. Notice the 'template ¡typename T¿' on line 7, and the 'Box¡T¿::' on line 8. If you forget either of these, the compiler will generate an error because it won't associate the definition with the generic template class.

Non-Type Template Parameters

- Templates can also take parameters that are values, not types.
- Must be known at compile time (constants).
- Example: template ¡class T, int size¿
- Useful for specifying fixed sizes for arrays within a generic class.

Non-Type Template Parameters
- Templates can also take parameters that are values, not types.
- Must be known at compile time (constants).
- Example: template ¡class T, int size¿
- Useful for specifying fixed sizes for arrays within a generic class.

Besides types, templates can also accept non-type parameters. These are regular values, like integers, but they must be compile-time constants. A classic use case is a generic Array class where you pass both the type of the elements and the size of the array as template parameters. The compiler then generates a specific class for an Array of ints of size 10, and a different class for an Array of ints of size 20.

Non-Type Template Parameter Example

- template ¡typename T, int N¿
- class Array {
- private:
- T arr[N];
- public:
- int getSize() { return N; }
- };
- // Usage:
- Array¡int, 10¿ myIntArray;
- Array¡double, 5¿ myDoubleArray;

Non-Type Template Parameter Example
template ¡typename T, int N¿
class Array {
private:
T arr[N];
public:
int getSize() { return N; }
};
// Usage:
Array¡int, 10¿ myIntArray;
Array¡double, 5¿ myDoubleArray;

Here we see the generic Array class. It takes a type 'T' and an integer 'N'. Inside the class, we declare a raw C-style array 'T arr[N]'. Because 'N' is a template parameter, its value is known at compile time, which is exactly what C++ requires for declaring arrays. We can then instantiate Arrays of specific types and specific, fixed sizes.

Template Specialization

- Sometimes a generic template doesn't work for a specific data type.
- Template specialization allows defining a different implementation for a specific type.
- Example: A generic max() function, but you want specialized behavior when comparing char arrays (C-strings).

Template Specialization
- Sometimes a generic template doesn't work for a specific data type.
- Template specialization allows defining a different implementation for a specific type.
- Example: A generic max() function, but you want specialized behavior when comparing char arrays (C-strings).

What happens when your generic template logic works for 99% of types, but fails or is inefficient for one specific type? This is where Template Specialization comes in. You can tell the compiler: 'Use this generic blueprint for everything, EXCEPT for this one specific type; for that type, use this specialized blueprint instead.'

Function Template Specialization

- // Generic template
- template ¡typename T¿
- T myMax(T a, T b) { return (a ¿ b) ? a : b; }
-
- // Specialization for char*
- template ¡¿
- const char *myMax¡const char¿*(const char *a*, *const char* b) {
- return (strcmp(a, b) ¿ 0) ? a : b;
- }

Function Template Specialization
// Generic template
template ¡typename T¿
T myMax(T a, T b) { return (a ¿ b) ? a : b; }
// Specialization for char*
template ¡¿
const char *myMax¡const char¿*(const char *a*, *const char* b) {
return (strcmp(a, b) ¿ 0) ? a : b;
}

Here is an example. We have a generic myMax function that uses the greater-than operator. But if we pass C-strings (const char*), the greater-than operator will just compare their memory addresses, not the strings themselves! To fix this, we create a specialization for const char* that uses strcmp. Notice the syntax 'template ¡¿' which indicates this is a full specialization.

Standard Template Library (STL)

- Templates are the core foundation of the C++ Standard Template Library.
- STL provides heavily tested, optimized, and generic algorithms and data structures.
- Examples:
- - std::vector¡T¿
- - std::list¡T¿
- - std::sort(first, last)

Standard Template Library (STL)
- Templates are the core foundation of the C++ Standard Template Library.
- STL provides heavily tested, optimized, and generic algorithms and data structures.
- Examples:
- - std::vector¡T¿
- - std::list¡T¿
- - std::sort(first, last)

To conclude our introduction to templates, it is crucial to mention the Standard Template Library, or STL. Everything we just learned—function templates and class templates—is exactly how the STL is built. When you use std::vector¡int¿, you are instantiating a class template. When you use std::sort, you are calling a function template. Understanding templates is key to mastering C++ because it allows you to utilize the full power of the STL.

Summary

- Templates enable generic programming, promoting code reuse.
- Function templates operate on generic types, deducing them from arguments.
- Class templates create generic structures, usually requiring explicit type specification.
- Non-type parameters and specialization offer advanced control.
- Templates are compile-time constructs.

Summary
- Templates enable generic programming, promoting code reuse.
- Function templates operate on generic types, deducing them from arguments.
- Class templates create generic structures, usually requiring explicit type specification.
- Non-type parameters and specialization offer advanced control.
- Templates are compile-time constructs.

Let's summarize. Templates are a mechanism for generic programming, allowing us to write reusable, type-safe code. We covered function templates which can automatically deduce types, and class templates which define generic structures. We touched on non-type parameters and specialization for fine-tuning behavior. Remember that templates are resolved at compile time, meaning there is no runtime performance overhead compared to writing separate overloaded functions manually.