Lecture 5

Dr. Umair Rehman

Agenda

- Templates enable generic functions and classes.
- Specialization customizes templates for specific types.
- Partial Specialization tweaks class templates for patterns.
- Mixins add modular behavior via templated inheritance.
- CRTP gives compile-time polymorphism using self-referencing templates.

What Are Templates in C++?

- Templates allow you to write generic code
  - Code that works with any type.
- This avoids code duplication and increases reusability.
- There are two main kinds of basic templates:
  - Function Templates
  - Class Templates

Function Templates

```cpp
// 1. This is a function template declaration.
// It tells the compiler: "I'm going to write a function that works with any type T."
template <typename T>

// 2. Define a function 'max' that takes two values of type T.
// It returns the greater of the two using the ternary operator.
T max(T a, T b) {
    // 3. If a > b, return a. Otherwise, return b.
    return (a > b) ? a : b;
}

```

Function Templates

```
int main() {
    // 1. max(3, 7) – both are integers (int), so T = int
    std::cout << max(3, 7);             // Output: 7

    // 2. max(4.5, 2.1) – both are doubles, so T = double
    std::cout << max(4.5, 2.1);         // Output: 4.5

    // 3. max('a', 'z') – both are characters, so T = char
    // It returns the character with the higher ASCII value
    std::cout << max('a', 'z');         // Output: z

    return 0; // Good practice to include
}

```

Function Templates

- template <typename T> tells the compiler this is a template function.
- T is a placeholder type.
- The actual type is determined when you call the function.

Class Templates

```cpp
// 1. Declare a class template that works with any type T
template <typename T>
class Box {
private:
    // 4. Declare a private variable of type T to store the value
    T value;

public:
    // 7. Setter function: sets the internal value to the input parameter
    void set(T val) { value = val; }

    // 8. Getter function: returns the stored value
    // `const` means this method doesn't modify the object
    T get() const { return value; }
};

```

Class Templates

```cpp
int main() {
    // 2. Create a Box that stores an int (T = int)
    Box<int> intBox;

    // 3. Set the value inside intBox to 10
    intBox.set(10);

    // 4. Get the value from intBox and print it → prints 10
    std::cout << intBox.get();

    // 6. Create a Box that stores a std::string (T = std::string)
    Box<std::string> strBox;

    // 7. Set the value inside strBox to "hello"
    // Note: string literals like "hello" are of type const char*,
    // but they implicitly convert to std::string
    strBox.set("hello");

    // 8. Get the value from strBox and print it → prints hello
    std::cout << strBox.get();

    return 0; // Optional, but good practice
}

```

Template Instantiation

- When you use a template with a specific type, like Box<int>, the compiler generates a new class specifically for that type.

```
class Box_int {
private:
    int value;

public:
    void set(int val) { value = val; }
    int get() const { return value; }
};

```

Template Parameters

- Writing code where types (or even values) are passed as parameters to functions or classes — just like variables.

- Instead of hardcoding a specific type (int, float, etc.), you let the compiler figure it out based on usage.

Template Parameters

- You can have multiple parameters:

```
template <typename T, typename U>
T add(T a, U b) {
    return a + b;
}

```

Template Parameters: Generic Functions Work With Multiple Types

- You can write one function (like add) that works with any data type.

- Instead of making separate int add(), double add(), string add() — you make one.

Template Parameters: You Can Mix Types

- It introduces multiple template parameters (T, U) — shows that the two inputs don't have to be the same type.

- Example: add(3, 4.5) → T = int, U = double

Template Parameters: You Must Be Careful With Return Types

- This version returns type T only.

- It teaches you that type conversion matters
  - Returning a double + int as an int might lose data.
  - auto result = add(3, 4.5); // result = 7.5, type is double
  - add(5, 'A'); // 'A' is ASCII 65 → 5 + 65 = 70

Policy Based Design

- Policy-Based Design is a C++ template pattern where custom behavior is injected via template parameters
  - Not via inheritance or runtime polymorphism.
- It uses compile-time composition, so there's zero runtime overhead.

Policy Based Design

```
#include <iostream>
#include <string>

//////////////////////
// Policy Definitions
//////////////////////

// A simple logging policy that prints to console
class ConsoleLogger {
public:
    static void log(const std::string& message) {
        std::cout << "[Console] " << message << std::endl;
    }
};
```

Policy Based Design

```cpp
// A dummy logger that logs nothing (useful for disabling logs)
class NullLogger {
public:
    static void log(const std::string&) {
        // Do nothing
    }
};

// A custom file logger (for illustration – not writing to a real file here)
class FileLogger {
public:
    static void log(const std::string& message) {
        std::cout << "[File] (simulated) " << message << std::endl;
        // In real code: write to file
    }
};
```

Policy Based Design

```cpp
//////////////////////////////////
// Main class using a Policy
//////////////////////////////////

// This is a generic class that takes a logging policy as a template
template <typename LogPolicy>
class Printer {
public:
    void print(const std::string& msg) {
        LogPolicy::log("Printing: " + msg);  // Delegate logging to policy
        std::cout << msg << std::endl;       // Actual printing
    }
};
```

Policy Based Design

```
int main() {
    // Printer that logs to console
    Printer<ConsoleLogger> consolePrinter;
    consolePrinter.print("Hello from ConsoleLogger!");

    // Printer that does not log at all
    Printer<NullLogger> silentPrinter;
    silentPrinter.print("This one has no logging");

    // Printer that uses a simulated file logger
    Printer<FileLogger> filePrinter;
    filePrinter.print("This message is logged as if to a file");

    return 0;
}
```

Policy Based Design

Feature	Why it matters
Compile-time switch	No virtual function overhead
Reusable policies	Write once, reuse across many classes
No inheritance	Doesn’t require shared base classes
Flexible	

What is CRTP?

- Curiously Recurring Template Pattern.
  It’s when a base class is templated on the derived class.

- In short:
  - The child passes itself as a template argument to the parent.

- Static polymorphism (like virtual functions, but without runtime cost)
  - Code reuse
  - Compile-time interface enforcement

Logging with CRTP

```cpp
#include <iostream>
#include <string>

//////////////////////////////////////////
// CRTP Base Class: Expects Derived class
//////////////////////////////////////////

// Base class takes the derived class as a template parameter
template <typename Derived>
class LoggerBase {
public:
    // This method is implemented in the base but calls Derived's method
    void log() {
        // static_cast lets us access derived class methods
        std::cout << "[LOG] " << static_cast<Derived*>(this)->getData() << std::endl;
    }
};

```

Logging with CRTP

```

///////////////////////////////////////////
// Derived Class A
///////////////////////////////////////////

class TemperatureSensor : public LoggerBase<TemperatureSensor> {
public:
    std::string getData() const {
        return "Temperature: 22.5°C";
    }
};

///////////////////////////////////////////
// Derived Class B
///////////////////////////////////////////

class PressureSensor : public LoggerBase<PressureSensor> {
public:
    std::string getData() const {
        return "Pressure: 101.3 kPa";
    }
};
```

Logging with CRTP

```cpp
//////////////////////////////////////////
// Main
//////////////////////////////////////////

int main() {
    TemperatureSensor temp;
    temp.log();  // [LOG] Temperature: 22.5°C

    PressureSensor pressure;
    pressure.log();  // [LOG] Pressure: 101.3 kPa

    return 0;
}

```

What is CRTP?

Feature	CRTP	Virtual Function
Performance	Fast (no vtable)	Slower (needs vtable lookup)
Runtime polymorphism	Not supported	Supported
Compile-time checking	Enforced via templates	= Not enforced until runtime
Flexibility	Template magic (mixins, etc.)	Inheritance hierarchy

What is MIXINS?

- A mixin is a class that adds reusable behavior to another class without being its parent in a traditional inheritance hierarchy.

- Think of mixins as:
    - “Plug-in” building blocks you can stack together to give a class extra abilities.

MIXINS

```cpp
#include <iostream>
#include <string>

//////////////////////////////
// Mixin for logging behavior
//////////////////////////////

template <typename Derived>
class LoggerMixin {
public:
    void log(const std::string& msg) {
        std::cout << "[LOG] " << msg << std::endl;
    }
};

//////////////////////////////
// A regular class mixing in Logger
//////////////////////////////

class MyComponent : public LoggerMixin<MyComponent> {
public:
    void run() {
        log("Running MyComponent task...");
        std::cout << "Doing actual work..." << std::endl;
    }
};

```

MIXINS

```cpp
/////////////////////////////
// Main
/////////////////////////////

int main() {
    MyComponent comp;
    comp.run();
    // Output:
    // [LOG] Running MyComponent task...
    // Doing actual work...
    return 0;
}

```

Add More Mixins

```
template <typename Derived>
class TimestampMixin {
public:
    void printTime() {
        std::cout << "[TIME] 2025-09-17 16:00" << std::endl;
    }
};

class AdvancedComponent
    : public LoggerMixin<AdvancedComponent>,
      public TimestampMixin<AdvancedComponent> {
public:
    void run() {
        printTime();
        log("Advanced task running...");
    }
};

```

Add More Mixins

```
1. [TIME] Wed Sep 18 17:06:34 2025
2. [LOG] Advanced task running...
3.
```

Mixins vs Inheritance vs CRTP

Feature	Mixins	Classic Inheritance	CRTP
Multiple behaviors	Easily stackable	Often hard to scale	Usually used inside
Performance	Compile-time (fast)	Virtual table overhead	No runtime overhead
Flexibility	Choose features modularly	Rigid hierarchy	Great for enforcing APIs

Template Specialization

- Template Specialization lets you provide a custom implementation of a template for a specific type.

- Imagine you have a generic class or function that works for most types,but for some types, you want custom behavior.

- That's where specialization comes in.

Example 1: Function Template Specialization

```cpp
#include <iostream>
#include <string>

// Primary template: generic for all types
template <typename T>
void printType(const T& value) {
    std::cout << "Generic type: " << value << std::endl;
}

// Full specialization for std::string
template <>
void printType<std::string>(const std::string& value) {
    std::cout << "Specialized string type: " << value << std::endl;
}

int main() {
    printType(42);                  // Uses generic
    printType(3.14);                // Uses generic
    printType(std::string("Hi"));   // Uses specialized version
}

```

Example 2: Class Template Specialization (Generic Class)

```
template <typename T>
class Storage {
public:
    void print(const T& data) {
        std::cout << "Generic Storage: " << data << std::endl;
    }
};

```

Example 2: Class Template Specialization (Full specialization in Bool)

```cpp
// -----------------------------------------------------
// Template Specialization for bool (T = bool)
// This overrides the generic template when T is bool
// -----------------------------------------------------
template <>                          // Specialization — no <T> needed because type is fixed
class Storage<bool> {               // Specialized version of class Storage for type bool
public:                             // Public access specifier
    void print(bool data) {         // Function takes a bool by value (small, cheap to copy)
        std::cout << "Specialized Storage: "      // Print specialized label
                  << (data ? "true" : "false")    // Use ternary operator to print "true" or "false"
                  << std::endl;                   // End line
    }
}; // End of specialized class

```

Example 2: Class Template Specialization (Usage)

```cpp
int main() {
    Storage<int> s1;
    s1.print(123);  // Generic

    Storage<bool> s2;
    s2.print(true); // Specialized
}
```

Example 3: Partial Specialization Customize Behavior When The Type Is A Pointer:

```
template <typename T>
class Wrapper {
public:
    void print() {
        std::cout << "Generic Wrapper" << std::endl;
    }
};

```

Example 3: Partial Specialization
Customize Behavior When The Type Is A Pointer:

```
// This is a partial specialization: it handles only pointer types (e.g., int*, double*)
template <typename T>
class Wrapper<T*> {
public:
    // Specialized print function for pointer types
    void print() {
        std::cout << "Pointer Wrapper" << std::endl;
    }
};

```

Example 3: Partial Specialization
Customize Behavior When The Type Is A Pointer:

```
int main() {
    Wrapper<int> a;
    a.print(); // Generic Wrapper

    Wrapper<int*> b;
    b.print(); // Pointer Wrapper
}

```

When to Use Template Specialization

- You should use it when...You need a generic default behavior
- But want different logic for some types (e.g. bool, char*)
- You want compile-time dispatch based on type
- You're implementing type traits or meta-programming logic

Conclusion

- Templates enable generic functions and classes.
- Specialization customizes templates for specific types.
- Partial Specialization tweaks class templates for patterns.
- Mixins add modular behavior via templated inheritance.
- CRTP gives compile-time polymorphism using self-referencing templates.