Lecture 05

Functions. Templates and generic programming in C++.

Advanced Programming - SISSA, UniTS, 2024-2025

Pasquale Claudio Africa

28 Oct 2024

Outline

1. Advanced topics on functions

- Function pointers
- Functors (function objects)
- Lambda functions
- Function wrappers (`std::function`)

2. Generic programming and templates

- Generic programming
- Function templates
- Class templates
- Notes on code organization
- Advanced template techniques and concepts

Advanced topics on functions

Functions in C++

In C++, functions are categorized as free functions or member functions (methods) of a class. Subcategories exist for normal functions and template functions, including automatic return functions.

For a free, non-template function, the typical declaration syntax is as follows:

```
return_type function_name(...);

auto function_name(...) -> return_type;

auto function_name(...);
```

The first two forms are essentially equivalent, so choose the one that suits your preference. In the third scenario, the compiler deduces the return type, which will be explained in more detail later.

Function names and function identifiers

A function is **identified** by:

- Its **name**, `function_name` in the previous examples. More precisely, its **fully qualified name**, which includes the namespace, for instance `std::transform`.
- The number and type of its **parameters**.
- The presence of the `const` qualifier (for methods).
- The type of the enclosing class (for methods).

Two functions with different identifiers are eventually treated as **different functions**, which is the key for **function overloading**.

⚠️ **The return type is NOT part of the function identifier!**

Why use free functions

A function represents a mapping from input data (via its arguments) to an output provided through the returned value or, in some cases, through an argument if the corresponding parameter is a non-const reference.

As a result, a free function is typically **stateless**, meaning that two different calls to a function with the same input will yield the same output. The only exception to this rule is when a local variable is declared `static`.

Hence, you typically implement a function when your requirement is indeed a straightforward input/output mapping, akin to the standard mathematical definition of a function, $f : U \to V$.

What type does a function return

In rare instances, a function returns a reference to an object passed by reference. This is usually done when concatenation is desired, with the streaming operator being a typical example.

```
std::ostream &operator<<(std::ostream &os, const MyClass &obj)
{
  // ...
  return os; // Return the stream.
}
```

This enables concatenation like so:

```
std::cout << x << " concatenated with " << y;
```

Default parameters

In the **declaration** of a function, you can provide default values for the rightmost parameters.

```
std::vector<double> cross_prod(const std::vector<double> &a,
                               const std::vector<double> &b,
                               const unsigned int ndim = 2);
```

For instance:

```
a = cross_prod(c, d); // This sets ndim to 2.
```

A recall of function overloading

```
int fun(int i);
double fun(const double &z);
// double fun(double y); // Error: ambiguous!

auto x = fun(1);   // Calls fun(int).
auto y = fun(1.0); // Calls fun(const double &).
```

The function that gives the best match of the argument types is chosen. Beware of possible ambiguities and implicit conversions!

Callable objects

A callable object refers to an object that can be called as if it were a function, i.e., using the function call operator `operator()`. Callable objects include:

- **Functions** (free functions or member functions).
- **Function pointers**.
- **Member function pointers**: These allow you to call member functions of a class.
- **Functors (function objects)**: Instances of classes that overload `operator()`.
- **Lambda functions**: Introduced in C++11, they are useful for short, local functions.
- **Function wrappers**: `std::function` generalizes the concept of a function pointer.

Function pointers

Pointers to functions

```
double integrand(double x);

// Define a new type "Pointer to a function
// taking a double as an input and returning a double".
using f_ptr = double (*)(double); // Or: typedef double (*f_ptr)(double);
double integrate(double a, double b, const f_ptr fun);

double I = integrate(0, 3.1415, integrand); // Passing function as a pointer.

f_ptr my_sin = std::sin; // Assigning a function pointer.

I = integrate(0, 3.1415, my_sin);
```

The name of the function is interpreted as a pointer to that function. However, you may precede it by `&`: `f_ptr f = &integrand`.

We will see in a while a safer and more general alternative to function pointers, the function wrapper `std::function` of the STL.

Dynamic function selection

You can use function pointers to select and call functions at runtime based on user input or other conditions.

```
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }

int main() {
    // ...
    int (*operation)(int, int); // 'operation' is a function pointer.

    if (user_input == "add") {
        operation = add;
    } else {
        operation = subtract;
    }

    const int result = operation(10, 5);  // Calls either add or subtract based on user input.

    return 0;
}
```

Member function pointers

```
std::vector<Shape*> shapes;
shapes.push_back(new Circle(3.0));
shapes.push_back(new Rectangle(2.0, 4.0));
shapes.push_back(new Circle(2.5));

// Define a member function pointer for the area function.
double (Shape::*area_fun)() const = &Shape::area;

for (const auto shape : shapes) {
    const double area = (shape->*area_fun)();
    std::cout << "Area: " << area << std::endl;
}

// Cleanup allocated objects.
for (auto shape : shapes) {
    delete shape;
}
```

Functors (function objects)

Functors (function objects)

A **function object** or **functor** is a class object which overloads the **call operator** (`operator()`). It has semantics very similar to that of a function:

```
class Cube {
public:
  double operator()(const double &x) const { return x * x * x; }
};

Cube cube{}; // A function object.
auto y = cube(3.4); // Calls Cube::operator()(3.4).
auto z = Cube{}(8.0); // I create the functor on the fly.
```

If the call operator returns a `bool`, the function object is a **predicate**. If a call to the call operator does not change the data members of the object, you should declare `operator()` as `const` (as with any other method).

Why functors?

A characteristic of a functor is that it may have a state, so it can interact with other objects and store additional information to be used to calculate the result.

```
class Calculator {
public:
    int result = 0;

    class Add {
    public:
        Calculator& calc;
        Add(Calculator& c) : calc(c) {}
        void operator()(int x, int y) { calc.result = x + y; }
    };
};

Calculator calc;
Calculator::Add add(calc);
add(5, 3); // Result is stored in calc.result;
```

Predefined functors in the Standard Template Library

Under the header `<functional>`, you find a lot of predefined (templated) functors and predicates.

```
std::vector<int> in = {1, 2, 3, 4, 5};

std::vector<int> out;

std::transform(in.begin(), in.end(),    // Source.
               std::back_inserter(out), // Destination.
               std::negate<int>());

const double prod = std::accumulate(in.begin(), in.end(), 1.0, std::multiplies<int>());
```

Now `out = {-1, -2, -3, -4, -5}.`.

`std::negate<type>` is a **unary functor** provided by the standard library.

`std::back_inserter<type>` inserts the transformed elements at the end (back) of vector `out`.

Some predefined functors in the STL

Functor	Description
`plus<T>`, `minus<T>`	Addition/Subtraction (Binary)
`multiplies<T>`, `divides<T>`	Multiplication/Division (Binary)
`modulus<T>`	Modulus (Unary)
`negate<T>`	Negative (Unary)
`equal_to<T>`, `not_equal_to<T>`	(Non-)Equality Comparison (Binary)
`greater`, `less`, `greater_equal`, `less_equal`	Comparison (Binary)
`logical_and<T>`, `logical_or<T>`, `logical_not<T>`	Logical AND/OR/NOT (Binary)

For a full list, have a look at this web page.

Lambda functions

Lambda expressions

We have a very powerful syntax to create short (and inlined) functions quickly: the lambda expressions (also called lambda functions or simply lambdas). They are similar to Matlab anonymous functions, like `f = @(x) x^2` or Python's `f = lambda x: x**2`.

```
auto f = [] (double x) { return 3 * x; }; // f is a lambda function.

auto y = f(9.0); // y is equal to 27.0.
```

Note that I did not need to specify the return type in this case, the compiler deduces it as `decltype(3 * x)`, which returns `double`.

Capture specification

The capture specification allows you to use variables in the enclosing scope inside the lambda, either by value (a local copy is made) or by reference.

- `[]`: Captures nothing.
- `[&]`: Captures all variables by reference.
- `[=]`: Captures all variables by making a copy.
- `[y]`: Captures only `y` by making a copy.
- `[&y]`: Captures only `y` by reference.
- `[=, &x]`: Captures any referenced variable by making a copy, but capture variable `x` by reference.
- `[this]`: Captures the `this` pointer of the enclosing class object.
- `[*this]`: Captures a copy of the enclosing class object.

An example of use of `[this]`

With `[this]`, we get the `this` pointer to the calling object:

```
class MyClass {
public:
    void compute(double a) {
        auto prod = [this, &a]() { x *= a; };
        std::for_each(v.begin(), v.end(), prod);
    }
private:
    double x = 1.0;
    std::vector<double> v;
};

MyClass c;
double res = c.compute();
```

Here, `compute()` uses the lambda `prod` that **changes** the member `x`. To be more explicit, you can write `this->x *= a;`.

Function wrappers

Function wrappers

And now the **catch-all function wrapper**. The class `std::function<>` declared in `<functional>` provides polymorphic wrappers that generalize the notion of a function pointer. It allows you to use any **callable object** as **first-class objects**.

```
int add(int a, int b) {
    return a + b;
}

std::function<int(int, int)> func = add;

const int result = func(2, 3);
```

Function wrappers are **very useful** when you want to have a common interface to callable objects.

Function wrappers introduce a little overhead, since the callable object is stored internally as a pointer, but they are extremely flexible, and often the overhead is negligible.

Function wrappers and polymorphism

```
class Shape {
public:
    virtual double area() const = 0;
};

class Circle : public Shape {
public:
    Circle(double radius) : radius(radius) {}
    double area() const override { return 3.14159265359 * radius * radius; }
private:
    double radius;
};

auto compute_area = [](const Shape& s) { return s.area(); };
// 'auto' here resolves to std::function<double(const Shape&)>.

Circle circle(5.0);
std::cout << "Circle area: " << compute_area(circle) << std::endl;
```

A vector of functions

`std::function` can wrap any kind of **callable** object.

```
int func(int, int); // A free function.

class F2 { // A functor.
public:
    int operator()(int, int) const;
};

// A vector of functions.
std::vector<std::function<int(int, int)>> tasks;
tasks.push_back(func); // Wraps a function.
F2 func2{};
tasks.push_back(func2); // Wraps a functor.
tasks.push_back([](int x, int y){ return x * y; }); // Wraps a lambda function.

for (auto t : tasks)
    std::cout << t(3, 4) << endl;
```

It prints the result of `func(3, 4)`, `func2(3, 4)`, and `12` (3×4).

`std::bind` and function adapters

`std::bind` provides flexibility and reusability in code by decoupling function logic from its arguments and context, making it easier to work with functions as first-class objects.

```
int add(int a, int b) {
    return a + b;
}

// Create a new function based on 'add' where argument 1 is set equal to 5.
std::function<int(int)> add5 = std::bind(add, 5, std::placeholders::_1);
const int result = add5(3);
```

In modern C++, lambda functions often offer a more concise and readable alternative to `std::bind`, which still remains valuable for complex binding scenarios or when you need to reuse a set of bound arguments.

Generic programming and templates

Generic programming

What is generic programming?

- Generic programming is a programming paradigm that aims to write code in a way that's independent of data types.
- It focuses on creating reusable and versatile code by using templates or type abstractions.
- The goal is to develop algorithms and data structures that work with various data types.

Why use generic programming?

- **Reusability:** Write code once, use it with multiple data types.
- **Type safety:** Ensures that type-related errors are caught at compile-time.
- **Performance:** Optimized code for specific data types without code duplication.
- **Expressiveness:** Code that's concise and easier to read and maintain.

Examples of generic programming

- **STL (Standard Template Library):** Offers a collection of generic data structures and algorithms.
- **Function templates:** Write functions that work with various data types.
- **Class templates:** Create versatile, type-safe data structures.

Use cases of generic programming

1. **Containers:** Generic data structures like vectors, stacks, and queues.
2. **Algorithms:** Generic sorting, searching, and transformation algorithms.
3. **Math operations:** Functions for arithmetic operations that work with multiple numeric types.
4. **Custom data structures:** Building generic trees, graphs, or linked lists.

Function templates

Motivation

Consider the following example:

```
bool less_than(char x1, char x2) {
    return (x1 < x2);
}

bool less_than(double x1, double x2) {
    return (x1 < x2);
}

// ...
```

You could end up with multiple overloads of the same function: they all have the same implementation, but only differ by few details (such as argument types).

What are function templates?

- Function templates are a feature in C++ that allows you to define generic functions.
- They enable you to write a function once and use it with different data types.
- Function templates are defined using the `template` keyword, followed by type parameters enclosed in angle brackets.

```
template <typename T>
bool less_than(T x1, T x2) {
    return (x1 < x2);
}
```

Creating and using function templates

Function template declaration (and definition)

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

Function template instantiation

```
const int result1 = add<int>(5, 3);
const int result2 = add(5, 3); // T automatically deduced as int.

const double result3 = add<double>(2.5, 3.7);
const double result4 = add(2.5, 3.7); // T automatically deduced as double.
```

Default template parameters

You can give defaults to the rightmost parameters.

```
template <typename T, typename U = double>
multiply_and_add(T a, U b, T c) {
    return a * b + c;
}

// Uses default type double for the second parameter.
const int result1 = multiply_and_add(5, 2.5, 3);

// Uses double for both 'T' and 'U'.
const double result2 = multiply_and_add(2.5, 3.7, 1.2);

// Uses float for the first parameter, int for the second.
const float result3 = multiply_and_add<float, int>(2.5, 3, 1.0);
```

Function template specialization

Template specialization allows you to define different behavior for specific template arguments.

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

// When 'T' is 'char', this version is invoked.
template <>
char max(char a, char b) {
    return (std::toupper(a) > std::toupper(b)) ? a : b; // max('A', 'b') is 'b'.
}
```

Example: vector sum using function template

The following function works for any type `T` for which `operator+` is defined.

For instance, this function can concatenate a vector of strings.

```
template <typename T>
T vector_sum(const std::vector<T>& vec) {
    // Initialize sum using T's default constructor
    // (e.g., 0 for numbers, empty for strings.

    T sum{};

    for (const T& elem : vec) {
        sum += elem;
    }

    return sum;
}
```

Important facts about templates

Templates serve as models for generating functions (or classes) once the template parameters are associated with actual types or values at the instance of the template.

This has two important implications:

1. Actual compilation occurs **only** when the template is instantiated (i.e., when it is actually used in your code). Only then can the compiler deduce the template arguments and have the necessary information to produce the machine code.
2. Thus, some compilation errors may only appear when the template is used!

Constant values as template parameters

You can give defaults to the rightmost parameters (this applies also).

A template parameter may also be an **integral** value.

```
template <int N, int M = 3>
constexpr int multiply() {
    return N * M;
}

constexpr int result1 = multiply<5>();    // Calculates 5 * 3 at compile-time.
constexpr int result2 = multiply<2, 7>(); // Calculates 2 * 7 at compile-time.
```

Only integral types can be used (e.g., integers, enumerations, pointers, ...).

`constexpr` can be applied to variables, functions, and constructors, to ensure that they are evaluated at **compile time**.

Pros of generic programming

- **Code reusability:** Reduced need to write similar code for different data types.
- **Type safety:** Compile-time checks to ensure type correctness.
- **Efficiency:** Optimized code for specific data types.
- **Readability:** Cleaner and more expressive code.

Cons of generic programming

- **Complexity:** Generic code may be more complex due to abstraction.
- **Compile-time overhead:** Template instantiation can lead to longer compile times.
- **Debugging:** Template error messages can be challenging to decipher.

Class templates

Introduction to class templates

- Class templates allow you to define generic classes.
- Syntax:

```
template <typename T>
class ClassName { /* ... */ };
```

- Type parameterization enables the class to work with different data types.

Advantages of class templates

- **Encapsulation** of data and behavior for a specific data type.
- **Code reusability**: Define a class structure once and use it with various types.
- **Type safety**: Prevents mixing incompatible types.

Creating and using class templates

Class template declaration (and definition)

```
template <typename T>
class List {
public:
    // ...
private:
    T value;
    List *next;
};
```

Class template instantiation

```
List<int> list_int;       // T is int.
List<double> list_double; // T is double.
// ...
```

Class template specialization

Template specialization allows you to define different behavior for specific template arguments.

```
template <typename T>
class Vector {
    // Implementation of a dynamic array for type T.
};

// Partial specialization for 'std::string'.
template <>
class Vector<std::string> {
    // Specialized behavior for strings.
};
```

Partial specialization

Partial specialization refines specialized behavior for specific subsets of template arguments.

```
// Generic template
template <typename T, int N>
class Array {
private:
    T data[N];
};

// Partial specialization for arrays of size 1.
template <typename T>
class Array<T, 1> {
private:
    T element; // No need to store an array for a single variable!
};

Array<int, 3> arr1;  // Uses the generic template for arrays of size 3
Array<char, 1> arr2; // Uses the partially specialized template for arrays of size 1
```

Template alias

Template aliases are a versatile feature that simplifies code by allowing you to create more concise and expressive names for complex template types.

```
template <typename T, int N>
class Array {
private:
    T data[N];
};

// Template alias to create an array of integers.
template <int N>
using IntArray = Array<int, N>;

IntArray<5> arr; // Creates an array of integers with 5 elements.
```

Example: templated stack class (LIFO)

```
template <typename T>
class Stack {
public:
    void push(const T& value) { elements.push_back(value); }

    T pop() {
        if (elements.empty()) {
            std::cerr << "Stack is empty" << std::endl;
            std::exit(1);
        }

        T top = elements.back();
        elements.pop_back();
        return top;
    }

private:
    std::vector<T> elements;
};
```

Best practices

- Use descriptive type parameter names.
- Avoid unnecessary template code duplication.
- Templatize functions for mathematical operations.

For class templates

- Use class templates for generic data structures.
- Define meaningful class names and method names.
- Leverage template specialization for customized behavior.

Notes on code organization

Template instantiation and linkage

- The compiler produces the code corresponding to function templates and class template members that are **instantiated** in each translation unit.
- It means that all translation units that contain, for instance, an instruction of the type `std::vector<double> a;` produce the machine code corresponding to the default constructor of a `std::vector<double>`. If we then have `a.push_back(5.0)`, the code for the `push_back` method is produced, and so on.
- If the same is true in other compilation units, the **same machine code** is produced several times. It is the **linker** that eventually produces the executable by selecting only one instance.

⚠️ **There is an (almost inevitable) waste of compilation time.**

The problem

- **Template definitions need to be available at the point of instantiation**. When a template is used with specific type arguments, the compiler needs to see the template definition to generate the code for that particular instantiation. Placing the template definition in a source file would make it unavailable for instantiation in other source files.
- If you place template definitions in source files and use the template in multiple source files, you may encounter linker errors due to multiple definitions of the same template. **Placing the template definitions in a header file** ensures that the definition is available for all source files that include it, and the linker can consolidate the definitions as needed.

Possible file organizations with templates

1. Leave everything in a **header file**. However, if the functions/methods are long, it may be worthwhile, for the sake of clarity, to separate definitions from declarations. You can put declarations at the beginning of the file and only short definitions. Then, at the end of the file, add the long definitions for readability.
2. **Separate** declarations (`module.hpp`) and definitions (`module.tpl.hpp`) when templates are long and complex. Then add `#include "module.tpl.hpp"` at the end of `module.hpp` (before closing its header guard).
3. **Explicitly instantiation** for a specific list of types. Only in this case, definitions can go to a source file. But if you instantiate a template for other types not explicitly instantiated, the compiler will not have access to the definition, leading to linker errors.

Explicit instantiation

We can tell the compiler to produce the code corresponding to a template function or class using **explicit instantiation**. If a source file contains, for instance:

```
template double func(const double &);
template class MyClass<double>;
template class MyClass<int>;
```

then the corresponding object file will contain the code corresponding to the template function `double func<T>(const T &)` with `T=double` and that of **all methods** of the template class `MyClass<T>` with `T=double` and `T=int`.

This can be useful to save compile time when debugging template classes (since the code for all class methods is generated).

Advanced template techniques and concepts

Type deduction and the `auto` keyword

- Type deduction allows the compiler to determine the data type of variables and return values automatically.
- The `auto` keyword simplifies code and improves readability.

```
auto sum1 = add(5, 3);        // int
auto sum2 = add(2.5, 3.7);    // double
auto sum3 = add(1.0f, 2.0f); // float
```

Type deduction with `auto` is particularly useful when you want to write more generic code that adapts to different data types without explicitly specifying them.

⚠️ Don't overuse `auto`!

The use of `this` in templates (1/2)

In a class template, derived names are resolved only when a template class is instantiated (only then the compiler knows the actual template argument). Other names are resolved immediately.

```
void my_fun() { ... }

template <typename T> class Base {
public:
    void my_fun(); // Not a good idea!
};

template <typename T>
class Derived : Base<T> {
public:
    void foo() { my_fun(); ... } // Which 'myfun'?
};
```

In this case, the free function `my_fun()` is used.

The use of `this` in templates (2/2)

Solution: use `this` :

```
template <typename T>
class Derived : Base<T> {
public:
    void foo() { this->my_fun(); ... }
}
```

or the fully-qualified name:

```
template <typename T>
class Derived : Base<T> {
public:
    void foo() { Base<T>::my_fun(); ... }
}
```

In this case, the compiler understands that `my_fun()` depends on the template parameter `T` and will resolve it only at the instance of the template class.

Template template parameters

- Template template parameters allow you to define templates as template arguments.
- Used for defining higher-order templates that accept template classes.

```
template <typename T, template <typename> class C = std::complex>
class MyClass {
private:
    C<T> a;
};

MyClass<double, std::vector> x; // 'a' is a std::vector<double>.

MyClass<int> x; // 'a' is a std::complex<int>.
```

This feature allows to write expressions like `std::vector<std::complex<double>>`.

Template metaprogramming

- Template metaprogramming is a way to perform computations at compile time.
- It involves using template features to generate code during compilation.
- Common use cases: constant expressions, type traits, and generic algorithms.

SFINAE (Substitution Failure Is Not An Error)

- SFINAE is a C++ rule that allows you to enable or disable function templates based on the validity of their substitutions.
- Used for type traits and selective function overloading.

Example: Fibonacci with template metaprogramming

```
template <int N>
class Fibonacci {
public:
    static constexpr int value = Fibonacci<N - 1>::value + Fibonacci<N - 2>::value;
};

template <>
class Fibonacci<0> {
public:
    static constexpr int value = 0;
};

template <>
class Fibonacci<1> {
public:
    static constexpr int value = 1;
};

constexpr int n = Fibonacci<10>::value; // Calculated at compile-time.
```

Example: type traits with SFINAE

```cpp
template <typename T>
class has_print {
public:
    template <typename U>
    static std::true_type test(decltype(U::print)*);

    template <typename U>
    static std::false_type test(...);

    static constexpr bool value = decltype(test<T>(0))::value;
};

class MyType {
public:
    void print() {}
};

std::cout << std::boolalpha;
std::cout << has_print<MyType>::value << std::endl; // true
std::cout << has_print<int>::value << std::endl;    // false
```

Variadic templates

- Variadic templates allow functions and classes to accept a variable number of arguments. The `...` syntax is used to define them.
- Useful for handling functions with multiple arguments of varying types.

Example: recursive sum function

```
template <typename T>
T sum(T value) {
    return value;
}

template <typename T, typename... Args>
T sum(T first, Args... rest) {
    // Consume the first argument, then recurse over remaining arguments.
    return first + sum(rest...);
}
```

CRTP (Curiously Recurring Template Pattern)

- CRTP is a design pattern where a derived class inherits from a base class template with itself as the template argument.
- Used to achieve static polymorphism and code reuse.

Traits classes and policy-based design

- Traits classes are used to encapsulate properties and behaviors of types.
- Policy-based design involves creating classes or functions with interchangeable policies to customize behavior.

Example: CRTP for static polymorphism

```
template <typename Derived>
class Shape {
public:
    double area() {
        return static_cast<Derived*>(this)->area();
    }
};

class Circle : public Shape<Circle> {
public:
    double area() {
        // Compute area of a circle.
    }
};

Circle c; c.area();
```

CRTP allows the `Shape` class to know the interface of its derived class at compile time, enabling **static** (compile-time) **polymorphism** and avoiding runtime overhead.

Example: traits for type information

```cpp
#include <type_traits>

template <typename T>
void process_type(T value) {
    if constexpr (std::is_integral_v<T>) {
        // Process integral types.
    } else if constexpr (std::is_floating_point_v<T>) {
        // Process floating-point types.
    } else {
        // Default behavior.
    }
}
```

⚠️ **`if constexpr` available since C++17. It evaluates conditions at compile-time.**

The Standard Template Library.