Chapter 4: Functions, Linking, and Scopes

Functions are the verbs of C++; they are the primary mechanism for creating reusable, modular, and abstracted blocks of code. This chapter explores how to define functions, how the linker connects them across files, and the rules governing their visibility.

4.1 Function Signatures and Overloading

- **Function Declaration (Prototype):** A declaration introduces a function's name to the compiler. It specifies the function's name, return type, and parameter types. It acts as a contract, promising the compiler that a definition exists somewhere else. This is what you put in header files.
- **Function Definition:** The actual implementation of the function, including the code that executes when it is called.

Function Overloading

Why overload? Overloading is a form of compile-time polymorphism. It allows you to create a single, intuitive "verb" (the function name) that can operate on different types of "nouns" (the parameters). This makes an API cleaner and more expressive. Instead of `print_int(5)` and `print_string("hello")`, you can just have `print(5)` and `print("hello")`.

What makes a function unique? The compiler uses a process of **overload resolution** to select the best function to call. It does this by looking at the function's **signature**, which, for overloading purposes, consists of: 1. The function's name. 2. The number, order, and type of its parameters.

Crucially, the **return type is not** part of the signature for overloading. You cannot have two functions that differ only by their return type.

```cpp
// A set of overloaded functions
void print(int i) { /*...*/ }
void print(double d) { /*...*/ }
void print(const std::string& s) { /*...*/ }

// An ambiguous call - will not compile!
// A `char` could be promoted to `int` or `double`.
// The compiler doesn't know which is "better".
// print('A'); // ERROR: call to print is ambiguous
```

The underlying mechanism that makes this possible is **name mangling**, as discussed in Chapter 0. The compiler gives each overload a unique name for the linker (e.g., `_Z5printi`, `_Z5printd`), so the linker sees them as completely different functions.

Theory: Function Linkage

Why does linkage matter? Linkage is the rule that determines whether a symbol (a function or global variable name) is visible across different translation units (`.cpp` files). It's what makes modular programming—splitting your code into multiple files—possible.

- **External Linkage (the default):** By default, all non-`static` functions and global variables have external linkage. This means they are visible to the linker from any file in the project. This is how `main.cpp` can call a function defined in `utils.cpp`.
- **Internal Linkage:** A symbol with internal linkage is private to its own translation unit. The linker will not see it. In C++, the modern way to achieve this is with an **unnamed namespace**. The older C way was with the `static` keyword. Using `static` on a global function or variable is still valid, but the unnamed namespace is preferred as it can also apply to types (`class`, `struct`).

How it works in practice:

```cpp
// --- utils.h ---
void public_function(); // Declaration of a function with external linkage

// --- utils.cpp ---
#include "utils.h"
#include <iostream>

namespace { // Unnamed namespace makes this private to utils.cpp
    void private_helper() {
        std::cout << "Private helper running...\n";
    }
}

void public_function() {
    std::cout << "Public function running...\n";
    private_helper();
}

// --- main.cpp ---
#include "utils.h"

int main() {
    public_function(); // OK! Linker will find this in utils.o
    // private_helper(); // ERROR! This name is not visible outside of utils.cpp
}
```

4.2 Default Arguments and Inlining

Default Arguments

Why use them? Default arguments are a powerful tool for API design. They allow you to add new functionality (via new parameters) to a function without breaking all the existing code that calls it. It provides a simpler interface for common use cases while still allowing full control for advanced cases.

What are the rules? Default arguments must be the trailing (right-most) parameters in the function declaration. You cannot have a parameter with a default value followed by a parameter without one.

```cpp
// A flexible function for creating a UI element
void create_button(const std::string& text, int width = 100, int height = 30, bool is_visible = true) {
    // ...
}

// Common case: just need a button with some text
create_button("OK");

// Advanced case: need full control
create_button("Cancel", 150, 40, false);
```

The `inline` Keyword

Why does `inline` exist? 1. **Historical Reason (Performance):** For very small, frequently-called functions, the overhead of a function call (setting up a stack frame, jumping to the function's address, and returning) can be significant. `inline` was a *hint* to the compiler to replace the function call with the actual body of the function, eliminating the overhead. 2. **Modern Reason (Linkage):** This is now the most important use of `inline`. The One Definition Rule (ODR) states that every function must have exactly one definition. If you define a function in a header file, and that header is included in multiple `.cpp` files, you will get a "multiple definition" linker error. Marking the function as `inline` relaxes this rule. It tells the linker, "I know this function might be defined in multiple places. They are all identical, so just pick one and discard the rest."

What you need to know today: * Compilers are now extremely good at deciding when to inline for performance. They will often ignore your `inline` hint for complex functions and will inline simple functions even without the keyword. * **The main purpose of `inline` is to allow function definitions in header files.** * Any function defined entirely inside a `class` or `struct` definition is implicitly `inline`.

```cpp
// Inside a class, this is implicitly inline. This is necessary because the class
// definition itself is in a header file, and this function's definition comes with it.
class MyWidget {
public:
    int get_id() const { return id_; } // Implicitly inline
private:
    int id_;
};
```

Projects for Chapter 4

Project 1: The Overloaded Data Logger

- **Problem Statement:** Create a logging utility with a function named `log_data`. This function should be overloaded to handle different data types: `int`, `double`, `const char*`, and `std::string`. Each overload should print a descriptive prefix to the console, e.g., `"LOG (int): 42"`, `"LOG (string): Hello, World!"`.
- **Core Concepts to Apply:** Function overloading for different parameter types.
- **Hint:** You will have four separate definitions for the `log_data` function, each with a different parameter type.

Project 2: The Linkage Experiment

- **Problem Statement:** Create a three-file project: `main.cpp`, `config.cpp`, and `config.h`.
  1. In `config.h`, declare a function `void load_config();`.
  2. In `config.cpp`, define `load_config()`. Inside this file, also define a helper function `static bool parse_line(const std::string& line);`. The `load_config` function should call `parse_line`.
  3. In `main.cpp`, include `config.h` and call `load_config()`. Then, on the next line, try to call `parse_line()`. Observe the compiler/linker error. In comments in `main.cpp`, explain why the call to `load_config()` works but the call to `parse_line()` fails, using the concepts of external and internal linkage.
- **Core Concepts to Apply:** External vs. internal linkage, `static` keyword for functions, multi-file project compilation.

Project 3: The Flexible Message Box

- **Problem Statement:** Design a function `void show_message_box(const std::string& title, const std::string& message, const std::string& button_text = "OK", bool is_modal = true);`. This function should simply print the properties of the message box it would create. In `main`, demonstrate your function's flexibility by calling it in multiple ways: providing only the title and message, providing a custom button text, and creating a non-modal message box.
- **Core Concepts to Apply:** Default arguments, API design.
- **Hint:** Remember that parameters with default arguments must come after parameters without them.