Chapter 5: Structs, Enums, and Code Grouping

This chapter focuses on C++'s fundamental tools for creating custom, composite data types: `structs` and `enums`. These constructs allow you to aggregate related data and to create type-safe, readable constants.

5.1 Structs and Data Aggregation

Why use structs? A `struct` (structure) is C++'s way of letting you create a new data type by bundling together other variables. This is the foundation of data aggregation. Instead of passing around a loose collection of variables (`string name`, `int id`, `double gpa`), you can group them into a single logical unit (`Student student`), making your code cleaner, more organized, and easier to understand.

By default, all members of a `struct` are `public`, meaning they can be accessed directly.

```
struct Point {
    double x;
    double y;
};

Point p = {10.5, -20.0}; // Aggregate initialization
std::cout << "X: " << p.x << ", Y: " << p.y << std::endl;
```

`struct` vs. `class` in C++

This is a common point of confusion. The **only** technical difference is the default access level: * `struct`: Default member and inheritance access is **`public`**. * `class`: Default member and inheritance access is **`private`**.

So which should you use? The C++ community has established a strong convention: * Use **`struct`** for **Plain Old Data (POD)** or simple data aggregates whose primary purpose is to bundle data. You are signaling that it's okay to directly access the members. * Use **`class`** when you need to enforce **invariants**—rules about the state of your object. By making data `private` and providing `public` methods, you control how the data is modified, which is the principle of encapsulation.

Plain Old Data (POD) and Standard Layout

Why does POD matter? This concept is crucial for interoperability, especially with C libraries. C code doesn't understand C++ features like constructors, private members, or virtual functions. A POD `struct` has a simple, predictable memory layout that C can understand, so you can safely pass a pointer to one to a C function.

In modern C++, the concept is refined into **Standard-Layout Types**. A type is considered standard-layout if it (among other rules): * Has no virtual functions or virtual base classes. * Has all its non-static data members with the same access control (`public`, `private`, etc.). * Has no user-provided constructors or destructors (in many cases).

Memory Padding and Alignment

Why does this happen? For performance. A CPU can read a 4-byte `int` or an 8-byte `double` much more efficiently if it is located at a memory address that is a multiple of its size (e.g., a 4-byte `int` at address 1000, not 1001). This is called **alignment**.

To enforce this, the compiler will insert unused bytes, called **padding**, between or after the members of a struct to ensure each member is correctly aligned.

How it works:

```
// On a typical 64-bit system where double is 8-byte aligned
struct BadLayout {
    char a;     // 1 byte
    // 7 bytes of padding inserted by compiler
    double b;   // 8 bytes
    int c;      // 4 bytes
    // 4 bytes of padding inserted for overall alignment
};
// sizeof(BadLayout) will likely be 24!

struct GoodLayout {
    double b;   // 8 bytes
```

```
    int c;      // 4 bytes
    char a;     // 1 byte
    // 3 bytes of padding inserted for overall alignment
};
// sizeof(GoodLayout) will likely be 16!
```

By reordering members from largest to smallest, you can often minimize the amount of padding the compiler needs to add, resulting in a smaller memory footprint.

5.2 Enumerations and Scoped Enums

Why use enums? Enums solve the "magic number" problem. Instead of writing `if (errorCode == 2)`, which is unreadable, you can write `if (errorCode == Error::NotFound)`, which is self-documenting and far less error-prone.

Traditional C-style `enum`

This is the original enum from C. It's essentially a set of named integer constants.

What are its flaws? 1. **Scope Pollution:** The enumerator names (`RED`, `GREEN`) leak into the surrounding scope. This can easily cause name collisions in large projects. 2. **Implicit Conversion to `int`:** They automatically convert to integers, which allows for nonsensical comparisons between unrelated types.

```
enum Color { RED, GREEN, BLUE };
enum Fruit { BANANA, APPLE };

if (RED == BANANA) { // Compiles! Both convert to 0. This is a logical error.
    // ...
}
```

`enum class` (Scoped Enums)

C++11 introduced **scoped enums** to fix these problems. They are the modern, preferred way to create enumerations.

Why are they better? 1. **Scoped:** The enumerator names are contained within the enum's scope. You must use `Color::Red`, which avoids name collisions. 2. **Strongly Typed:** They do not implicitly convert to `int` or any other type. This prevents the nonsensical comparisons seen above.

```
enum class Color { Red, Green, Blue };
enum class Fruit { Banana, Apple };

// if (Color::Red == Fruit::Banana) { // COMPILE ERROR! No conversion between types.
//     // ...
// }

// To get the integer value, you must be explicit:
int color_value = static_cast<int>(Color::Red); // OK
```

Specifying the Underlying Type

Why do this? By default, the underlying type of an `enum class` is `int`. You can specify a different integral type to control its memory size. This is useful for interoperating with hardware registers that expect a specific size, or for making data structures more compact.

```
// This enum will use only 1 byte of memory instead of the default 4 (for an int).
enum class Status : std::uint8_t {
    Pending,
    Processing,
    Complete,
    Failed
};
```

Projects for Chapter 5

Project 1: The RGB Color Mixer

- **Problem Statement:** Create a `struct` named `Color` that contains three `unsigned char` members: `r`, `g`, and `b`. Write a function `Color mix_colors(const Color& c1, const Color& c2)` that takes two colors and returns a new `Color` where each component is the average of the input colors. In `main`, create red (`{255, 0, 0}`) and blue (`{0, 0, 255}`) colors, mix them, and print the `r, g, b` values of the resulting purple color.
- **Core Concepts to Apply:** `struct` definition, member access, aggregate initialization, passing structs to functions.

Project 2: The Alignment Investigator

- **Problem Statement:** Create two structs, `InefficientLayout` and `EfficientLayout`. Both should contain the same three members: `char status;`, `double value;`, and `int id;`. In `InefficientLayout`, order them as listed. In `EfficientLayout`, reorder them from largest to smallest. In `main`, print the `sizeof` each struct and observe the difference. Add comments to your code explaining why padding causes `InefficientLayout` to be larger.
- **Core Concepts to Apply:** Memory padding, alignment, `sizeof` operator on structs.

Project 3: The Vending Machine State Machine

- **Problem Statement:** You are modeling a simple vending machine. Create a scoped enumeration `enum class VendingState : std::uint8_t { Idle, AcceptingMoney, DispensingItem };`. Write a function `void handle_state(VendingState current_state)` that takes a state and prints a different message for each one (e.g., "Idle: Please insert money.", "Dispensing: Your item is on its way."). In `main`, create a `VendingState` variable, cycle it through all three states, and call your function for each one.
- **Core Concepts to Apply:** `enum class` for type-safe states, specifying the underlying type, using enums in functions.