Part II: Core Constructs, Classes, and Basic I/O

This part delves into the foundational elements of C++ programming. You will explore the core data types, initialization methods, and the nuances of type inference using `auto`. The section covers essential control flow constructs such as conditionals and loops, as well as the intricacies of scoping and storage duration. Moving forward, you'll gain a solid understanding of functions, including overloading, parameter passing strategies, and the importance of `const` correctness for safer code. The basics of references and console input/output are also introduced. Finally, the part concludes with an in-depth look at classes—how to define them, manage access to their members, and utilize constructors and destructors—laying the groundwork for effective data abstraction and object-oriented design in C++.

Table of Contents

---

3. Data Types and Control Flow

As an experienced developer, you are comfortable with basic data types and control structures. However, C++ handles these primitives with greater precision and a deeper connection to the underlying hardware than managed languages. This chapter will focus on the C++ specifics, particularly how **initialization** prevents common bugs and how **storage duration** dictates resource management.

3.1 Built-in Types, Initialization, and Type Inference (`auto`)

Built-in Types

C++ types are defined by the ISO standard, but their exact size (the number of bits) is often **implementation-defined** (left up to the compiler) to ensure the best performance on the target architecture. This is a critical difference from C#, where sizes are generally fixed by the CLR specification.

Type Category	Type	Guaranteed Minimum Range (Example)	Common Size	Note
Integers	`int`	± 32767 (16-bit)	32-bit	Preferred general integer type.
	`long long` (C++11)	$\pm 9 \times 10^{18}$ (64-bit)	64-bit	The only guaranteed 64-bit integer.
Floats	`float`		32-bit (Single precision)	Use for performance where precision is secondary.
	`double`		64-bit (Double precision)	Preferred floating-point type.
Character	`char`		8-bit	Used for ASCII/byte data. Not guaranteed to be signed or unsigned.
Boolean	`bool`	`true` or `false`	Usually 1 byte	

For applications requiring precise, portable integer sizes, the Standard Library provides **fixed-width integer types** via the `<cstdint>` header:

- `std::int32_t`: Guaranteed 32-bit signed integer.
- `std::uint64_t`: Guaranteed 64-bit unsigned integer.

Best Practice: In professional C++, always use fixed-width types (`std::intN_t`) or the language's native types (`int`, `long long`) when the required size is important for portability or correctness.

The Dangers of Initialization Styles

C++ features three main ways to initialize variables. The choice is crucial because only one guarantees safety against **narrowing conversions**.

1. Copy Initialization (Legacy)

```
int x = 5;
```

2. Direct Initialization (Function-call style)

```
int y(5);
```

3. List Initialization (Uniform Initialization - The Modern Standard)

```
int z {5};
```

List initialization, using braces (`{}`), is the preferred modern style because it is the most consistent and, critically, it forbids **narrowing conversions**—conversions that risk data loss or a change in magnitude.

```
double big_num = 1.0e10;

// This succeeds, but data loss may occur (Narrowing Conversion)
int a = big_num; // a might not hold the full value of big_num

// This is a COMPILER ERROR because it is a narrowing conversion
int b {big_num}; // Error: initializer may be too large for target type
```

Takeaway: Always prefer **Uniform Initialization (`{}`)** for non-aggregate types to benefit from its narrowing check guarantees.

Type Inference with `auto`

C++'s `auto` keyword, introduced in C++11, is used for **type inference**, similar to C#'s `var`.

```
// C#: var list = new List<string>();
// C++: auto list = std::vector<std::string>{};

auto result = calculate_complex_value(10); // result is automatically the return type
```

Unlike C#, where `var` is often reserved for local, simple types, C++ encourages the use of `auto`, especially for iterator types or complex template returns, to keep code clean and maintainable. This prevents verbosity without sacrificing type safety, as the type is still strictly enforced at compile time.

3.2 Operators, Expressions, and Type Promotion

Most arithmetic, relational, and logical operators ($\texttt{+}, \texttt{=}, \texttt{<}, \texttt{&&}$) operate identically to C#. However, C++'s close relationship with C introduces specific rules regarding **implicit conversions** in expressions.

Integer Promotion and Type Conversions

When operands of different types are used in an expression, C++ automatically promotes the "smaller" type to the "larger" type to perform the operation. This is called **Integer Promotion** (or **Widening Conversion**).

```
int x = 5;
double y = 2.5;

// x is promoted to a double (5.0) before the addition.
// result is a double (7.5).
auto result = x + y;
```

The Implicit Warning (C# Developer Trap):

The danger lies in expressions involving different signed and unsigned types, or when a conversion results in silent truncation (allowed by non-uniform initialization).

```
unsigned int u = 10;
int s = -20;

// In C++, when a signed integer meets an unsigned integer,
// the signed integer is promoted to unsigned.
// -20 converted to an unsigned int is a very large positive number (e.g.,
4,294,967,276).
// The comparison is performed between (large positive) > 10, which is TRUE, not
FALSE as expected.
if (s > u) {
    // This branch executes!
}
```

Best Practice: Always use explicit type casts (Chapter 13) when mixing signed and unsigned integer types in conditional or relational expressions to avoid unexpected promotion rules.

3.3 Scoping, Lifetimes (Automatic, Static), and Storage Duration

Understanding C++ storage duration is the **most fundamental precursor to mastering C++ memory management (RAII)**. Your C# variables exist either on the stack (value types) or the heap (reference types, managed by the GC). C++ formalizes this through **Storage Duration**.

Scope defines the region of code where a variable is visible. **Storage Duration** defines how long the variable's memory exists.

Automatic Storage Duration (The Stack)

Variables declared inside a function or a block (`{}`) have **Automatic Storage Duration**.

1. **Allocation:** Allocated on the **Stack** when the execution enters the block.
2. **Destruction:** Deallocated when the execution leaves the block.

```cpp
void demonstrate_lifetime() {
    int outer_scope = 1; // Created on the stack

    { // Start of an inner block scope
        int inner_scope = 2; // Created on the stack
        // ...
    } // End of inner block: inner_scope's destructor is called; memory is freed.

} // End of function: outer_scope's destructor is called; memory is freed.
```

This **deterministic destruction** on exiting a scope is the essence of the **RAII principle** (Chapter 9). Every time a variable with Automatic Storage Duration is created, a **stack frame** is pushed, and when it is destroyed, the stack frame is popped.

Static Storage Duration

Variables with **Static Storage Duration** exist for the entire lifetime of the program.

1. **`static` Local Variables:** A variable inside a function marked `static` is initialized the *first* time the function is called, and its value persists across all subsequent calls.

    ```cpp
    void count_calls() {
        static int counter = 0; // Initialized only once
        counter++;
        std::cout << "Call count: " << counter << "\n";
    }
    // Calling count_calls() three times prints 1, 2, then 3.
    ```

2. **Global/Namespace Variables:** Variables declared outside any function or class are also static, initialized before `main()` runs, and destroyed after `main()` finishes.

Dynamic Storage Duration (The Heap)

Variables explicitly allocated using `new` (Chapter 6) have **Dynamic Storage Duration**. They persist until explicitly deallocated using `delete` or until their owner (a smart pointer, Chapter 9) automatically deallocates them.

3.4 Control Structures (`if`, `switch`, loops)

C++ provides standard control structures, but modern C++ standards have added useful syntactic sugar to simplify scoping and resource handling.

`if` and `switch` with Initializers (C++17)

The C++17 standard introduced the ability to include an initialization statement directly in the `if` and `switch` condition. This keeps the scope of the variable strictly local to the control structure, preventing accidental reuse later in the function.

```cpp
#include <iostream>
#include <optional>

std::optional<int> get_value() {
    // std::optional is covered in Chapter 16
    return 42;
}

void modern_if() {
    // 1. Initialize 'data' with the result of get_value()
    // 2. Test if 'data' has a value (i.e., is not empty)
    if (auto data = get_value(); data.has_value()) {
        std::cout << "Value found: " << data.value() << "\n";
    }
    // 'data' is NOT accessible here, correctly limiting its lifetime and scope.

    // Similarly for switch:
    switch (int val = data.value(); val) {
        case 42:
            std::cout << "The Answer is 42!\n";
            break;
        default:
            break;
    }
    // 'val' is NOT accessible here.
}
```

This idiom is highly encouraged in modern code, especially when dealing with resource acquisition (e.g., locks) or optional values.

The Range-Based `for` Loop (C++11)

Similar to C#'s `foreach`, the range-based `for` loop simplifies iterating over collections.

```cpp
#include <vector>
#include <iostream>

void iterate_modern() {
    std::vector<int> numbers = {10, 20, 30};

    // By value (creates a copy of each element)
    for (int n : numbers) {
        std::cout << n << " "; // Output: 10 20 30
    }

    // By reference (most efficient; allows modification)
    for (int& n : numbers) {
        n += 1; // Modifies the original vector elements
    }
}
```

The use of `int& n` (reference, Chapter 4.4) is the idiomatic way to loop through large objects to avoid the performance cost of copying each item.

Key Takeaways

- **Fixed vs. Implementation-Defined:** C++ built-in types have implementation-defined sizes; use fixed-width types like `std::int32_t` for portability when size matters.
- **Uniform Initialization:** Always use **braces (`{}`)** for initialization to benefit from the compiler's guarantee against dangerous **narrowing conversions**.
- **Type Promotion:** Be wary when mixing signed and unsigned types in expressions, as the signed type will be implicitly converted to unsigned, potentially leading to incorrect logic.
- **Automatic Storage is the Stack:** Variables declared within a scope have **Automatic Storage Duration**; their memory is allocated and, critically, **deterministically destroyed** on the stack when the scope exits. This is the foundation of RAII.
- **Modern Control Flow:** Embrace **`if` with initializer (C++17)** to limit the scope and lifetime of temporary variables used for checks, keeping your code cleaner and safer.

Exercises

1. **Initialization Safety Test:** Write a small C++ program that attempts to initialize a `short` (16-bit integer) with the value 50,000 using both `short s = 50000;` and `short s {50000};`.

   - *Task:* Describe which one generates a compiler error and which one generates a warning or silence.
   - *Hint:* The compiler should reject the `{}` initialization because 50,000 is too large for a 16-bit signed integer.

2. **Scope and Lifetime:** Write a function `demonstrate_static()` that uses a `static int` counter and a regular `int` counter. Call the function three times in `main()`.

   - *Task:* Explain why the two counters behave differently upon each function call.
   - *Hint:* The static variable has static storage duration, persisting across calls, while the regular variable has automatic storage duration, being re-created and re-initialized each time.

3. **Range-based `for` Loop Efficiency:** Given a class `BigData` that prints a message in its copy constructor. Write two `for` loops that iterate over a `std::vector<BigData>`.

   - *Task:* Use a reference (`BigData&`) in the first loop and a copy (`BigData`) in the second loop, and explain why one loop prints the copy constructor message repeatedly while the other does not.
   - *Hint:* Iterating by value (`T item : container`) involves a copy; iterating by reference (`T& item : container`) avoids the copy.

4. **`if` with Initializer Refactoring:** Imagine a variable `std::string status = find_status();` is used only in the immediate `if (status == "ERROR")` condition that follows it.

   - *Task:* Refactor this code into a single, idiomatic C++17 `if` with initializer statement, ensuring the `status` variable is no longer accessible outside the `if` block.
   - *Hint:* The syntax is `if (initializer; condition)`.

4. Functions, `const` Correctness, and Basic References

Functions are the building blocks of C++ programs, serving the same role as methods in C#. While the syntax is largely familiar, C++ introduces two fundamental concepts—**`const` correctness** and **references**—that are essential for writing correct, efficient, and idiomatic C++ code. Mastering these concepts is a crucial step in your transition from a managed environment.

4.1 Function Overloading and Signature Matching

C++ supports **function overloading**, allowing you to define multiple functions with the same name as long as their **function signature** is unique. The compiler uses a process called **overload resolution** to determine which function to call based on the number and type of the arguments.

A C++ function signature consists of the function's name and its parameters' types. The return type is **not** part of the signature, so you cannot overload functions that differ only in their return type.

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

// Overload 1: takes two integers
int add(int a, int b) {
    return a + b;
}

// Overload 2: takes a string and an integer
std::string add(const std::string& str, int val) {
    return str + " " + std::to_string(val);
}

int main() {
    // The compiler matches the call to the signature
    int sum = add(5, 10); // Calls Overload 1
    std::string text = add("Result:", 20); // Calls Overload 2

    std::cout << sum << "\n";
    std::cout << text << "\n";
    return 0;
}
```

The compiler performs this matching at compile time.

4.2 Parameter Passing: By Value and the Cost of Copying

By default, C++ passes all function parameters **by value**. This means that a **full copy** of the argument is created on the stack when the function is called.

For simple, small types like `int` or `char`, this is a non-issue. For user-defined types (classes and structs), especially large ones, passing by value can incur a significant performance penalty.

Consider a class that manages a large block of memory:

```cpp
#include <iostream>

class LargeObject {
public:
    LargeObject() {
        std::cout << "Default Constructor\n";
    }
    // A constructor that prints to show when a copy is made
    LargeObject(const LargeObject& other) {
        std::cout << "Copy Constructor\n";
    }
};

void processByValue(LargeObject obj) {
    // Function body
}

int main() {
    LargeObject my_object; // Default Constructor called
    std::cout << "Calling function...\n";
    processByValue(my_object); // Copy Constructor is called here!
    std::cout << "Function returned.\n";
    return 0;
}
```

When `processByValue` is called, a temporary copy of `my_object` is created. For a large object, this could mean allocating new memory and copying a significant amount of data, which is both slow and wasteful.

4.4 Lvalue References (Aliases) and Passing by Reference

To avoid the cost of copying, C++ provides **lvalue references**, which act as a true alias to an existing object. An lvalue reference is declared using the ampersand symbol (`&`).

An **lvalue** (location value) is an expression that identifies a location in memory. An lvalue reference must be initialized with an lvalue.

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

int main() {
    int value = 10;
    int& alias = value; // `alias` is now an alias for `value`

    std::cout << "Value: " << value << "\n"; // Prints 10
    std::cout << "Alias: " << alias << "\n"; // Also prints 10

    alias = 20; // Modifies the original 'value' variable
    std::cout << "New Value: " << value << "\n"; // Prints 20
    return 0;
}
```

This aliasing behavior is what makes **passing by reference** in a function so powerful. By passing a parameter as a reference, you give the function an alias to the original object, avoiding the copy.

```
// This function receives a reference to the original LargeObject
void processByReference(LargeObject& obj) {
    // No copy constructor is called here!
}

int main() {
    LargeObject my_object; // Default Constructor
    std::cout << "Calling function by reference...\n";
    processByReference(my_object); // No Copy!
    return 0;
}
```

However, a raw reference allows the function to modify the original object. For functions that should not have side effects, this is dangerous. This is where `const` correctness becomes essential.

4.3 `const` Correctness: Variables, Parameters, and Member Functions

The **`const`** keyword is a fundamental part of the C++ type system. It's a promise to the compiler that a value will not be modified. If that promise is broken, the compiler will refuse to compile the code.

`const` can be applied in three key ways:

1. `const` Variables

```
const double PI = 3.14159;
// PI = 3.0; // ERROR: assignment of read-only variable 'PI'
```

This is similar to C#'s `const` keyword.

2. `const` Function Parameters

By passing a parameter as a **`const` reference**, you get the efficiency of pass-by-reference *without* the danger of a function modifying your original object. This is the **most common and idiomatic way** to pass a user-defined type to a function.

```
void processByConstReference(const LargeObject& obj) {
    // obj is an alias to the original object, but it cannot be modified
    // obj = LargeObject(); // ERROR: obj is read-only
}
```

3. `const` Member Functions

A member function of a class can be marked `const`. This is a promise that the function will not modify any of the class's data members.

```
class Circle {
public:
    double radius;

    // A const member function
    double get_area() const {
        // radius = 10; // ERROR: a const member function cannot modify data
members
        return 3.14 * radius * radius;
    }
};
```

This is a critical design feature. A `const` member function can be called on a `const` object, guaranteeing immutability throughout the call chain.

4.5 Basic Console I/O (`std::cin`, `std::cout`)

The standard way to perform console input and output in C++ is through **I/O streams**, provided by the `<iostream>` header.

The `std::cout` (character output) object and `std::cin` (character input) object are used with the **stream insertion (`<<`)** and **stream extraction (`>>`)** operators. These operators are chainable.

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

int main() {
    int age;
    std::string name;

    std::cout << "Enter your name and age: ";

    // Read input from the console
    std::cin >> name >> age;

    // Use a mix of strings and variables in the output stream
    std::cout << "Hello, " << name << "!\n";
    std::cout << "You are " << age << " years old.\n";

    return 0;
}
```

The `\n` is the newline character. The `std::endl` manipulator also adds a newline and, importantly, flushes the output buffer, which can impact performance. For most cases, `'\n'` is preferred.

Key Takeaways

- **Passing by Value is Expensive:** The default pass-by-value mechanism creates a full copy of the argument, which can be inefficient for large objects.
- **Lvalue References are Aliases:** An lvalue reference (`T&`) is an alias to an existing variable. It is not a copy and does not occupy new memory for the object.
- **`const` Correctness is Critical:** Use `const` to enforce immutability. A `const` reference (`const T&`) is the standard way to pass large objects to functions when you do not need to modify them, providing both **efficiency** and **safety**.
- **`const` Member Functions:** Mark class member functions `const` if they do not modify the object's state, enabling them to be called on `const` objects.
- **Stream I/O:** Use `std::cout` and `std::cin` with the stream insertion (`<<`) and extraction (`>>`) operators for basic console I/O.

Exercises

1. **Cost of Copying:** Create a class `MyVector` that holds a `std::vector<double>` and prints a message in its copy constructor. Write a function `process(MyVector v)` that takes a `MyVector` by value.
   - *Task:* In `main()`, create a `MyVector` with 10,000 doubles and call the function. Observe how many times the copy constructor is called.
   - *Hint:* The copy constructor is called when the function is invoked.
2. **`const` Reference for Efficiency:** Refactor the previous exercise to use a `const MyVector&` parameter.
   - *Task:* Observe how many times the copy constructor is called in this version.
   - *Hint:* By using `const MyVector&`, you pass a reference, avoiding the copy altogether.
3. **Member Function `const`:** Create a simple `Rectangle` class with `double` member variables for `width` and `height`. Write a public `get_area()` method and mark it `const`.
   - *Task:* Create a `const Rectangle my_rectangle;` object and call `get_area()` on it. Then, try to write a non-`const` method that modifies `width` and call it on `my_rectangle`. What happens?
   - *Hint:* The compiler will reject the call to the non-`const` method on a `const` object.
4. **Chained I/O:** Write a program that prompts the user for their name and a favorite number. Use a single chained `std::cout` statement to print a greeting that includes both the name and number, and use a single `std::cin` statement to read them in.
   - *Task:* Show that `std::cout << "Hello, " << name << "!\n";` is valid and intuitive.
   - *Hint:* You can chain `>>` and `<<` operators together with different variable types.

---

5. Classes, Objects, and Data Abstraction

Classes and objects form the backbone of C++. They provide the mechanisms for **data abstraction** and **encapsulation**, concepts you already know well from C#. This chapter focuses on the C++ syntax and the critical role of the **destructor** in managing resources—a key difference from managed environments.

5.1 Defining a Class (Data and Member Functions)

In C++, you define a user-defined type using either the `class` or `struct` keyword.

```
// Example of a basic class definition
class Employee {
private:
    std::string name_; // Data member (often suffixed with '_')
    int id_;            // Data member

public:
    // Member function (method) declaration
    void set_name(const std::string& name);

    // Member function definition (can be defined inside or outside the class)
    int get_id() const {
        return id_;
    }
};
```

Class vs. Struct

In C++, the difference between `class` and `struct` is minimal—it's purely about the **default access specifier**:

- **`class`:** Members default to **`private`**. Used conventionally for types that enforce invariants and hide implementation details (true objects).
- **`struct`:** Members default to **`public`**. Used conventionally for simple data-holding types (like DTOs or PODs—Plain Old Data) where encapsulation is less important.

Best Practice: Use `class` when your type has methods and must enforce invariants. Use `struct` when your type is primarily a bundle of public data.

5.2 Access Specifiers (`public`, `private`, `protected`)

C++ uses access specifiers to control encapsulation, similar to C#. However, C++ access keywords define a **section** of the class definition, not a prefix for each member.

Specifier	Visibility	C# Analogy
`public:`	Accessible from anywhere.	`public`
`private:`	Accessible only by the class's own member functions.	`private`
`protected:`	Accessible by the class's member functions and member functions of derived classes.	`protected`

```
class Account {
// Everything below this line is private by default (because it's a class) or
explicitly defined.
private:
    double balance_ = 0.0; // Private data

    // Private method accessible only inside Account
    bool validate_pin(int pin) const;

protected:
    // Protected data visible to inherited classes (Chapter 11)
    std::string account_type_;

public:
    // Public interface accessible to external code
    void deposit(double amount);
};
```

You can use any access specifier multiple times within a class definition.

5.3 Introducing Constructors and Destructors

Constructors: Guaranteed Initialization

Constructors are special member functions called when an object is created. They ensure the object is placed into a valid, consistent state.

```
class Point {
private:
    int x_ = 0; // Default member initializer (C++11)
    int y_;

public:
    // Constructor
    Point(int x, int y) : x_(x), y_(y) {
        // Constructor body: usually empty if all initialization is done below
    }
};
```

The Member Initializer List

The syntax `Point(int x, int y) : x_(x), y_(y) { ... }` is called the **Member Initializer List**.

- **It is the C++ best practice for initializing members.**
- It performs **direct initialization** of members (e.g., calling the copy constructor once).
- If you assign values inside the constructor body (`x_ = x;`), the member is first **default initialized**, and then assigned—a less efficient two-step process. For primitive types this is moot, but for large objects, the Initializer List avoids an unnecessary copy or assignment operation.

Destructors: Guaranteed Deterministic Cleanup

The **destructor** is the most important concept in C++ object lifecycle management that differs from C#/Java.

- **Syntax:** Named `~ClassName()`. Takes no arguments and has no return type.
- **Purpose:** The destructor is called **deterministically** when an object's lifetime ends. Its job is to release any resource the object owns (e.g., closing file handles, releasing network sockets, or freeing dynamic memory).

```
class FileHandle {
private:
    std::string filename_;
    // Actual resource representation...
public:
    FileHandle(const std::string& name) : filename_(name) {
        std::cout << "File opened: " << filename_ << "\n";
        // Logic to acquire the actual file handle
    }

    // The Destructor
    ~FileHandle() {
        std::cout << "File closed: " << filename_ << "\n";
        // Logic to release the actual file handle
    }
};

void run_file_operation() {
    FileHandle log("debug.log"); // Object is created
    // ... use file ...
} // log goes out of scope, ~FileHandle() is called automatically and
deterministically.
```

This deterministic cleanup, tied to **Automatic Storage Duration** (Chapter 3.3), is the **Resource Acquisition Is Initialization (RAII)** principle (Chapter 9.1). The destructor is the "R" (Resource) **Release** mechanism.

5.4 Aggregate Initialization and Designated Initializers (C++20)

For simple classes (like structs) that have no user-defined constructors, no private members, and no virtual functions, C++ allows for simplified initialization using **Aggregate Initialization**.

Aggregate Initialization (Pre-C++20)

You can initialize members directly using braces, in the order they are declared:

```
struct ColorRGB {
    int r;
    int g;
    int b;
};

// Aggregate Initialization: order matters!
ColorRGB blue = {0, 0, 255};
```

```
// ERROR-Prone: If the struct order changes, this initialization breaks silently.
ColorRGB green = {0, 255, 0};
```

This is compact but error-prone because the reader must remember the declaration order.

Designated Initializers (C++20)

C++20 introduced **Designated Initializers**, a feature borrowed from C and similar to C#'s object initializer syntax, which drastically improves readability and safety for aggregates.

```
struct Vector2D {
    double x;
    double y;
};

// Designated Initializers (C++20)
Vector2D vec1 {
    .x = 10.0,
    .y = 5.0
};

// Order does not matter, improving safety and readability:
Vector2D vec2 {
    .y = 2.5,
    .x = 7.5
};
```

Best Practice: For simple C++ structures, use C++20 Designated Initializers for clear, order-independent, and safe initialization.

Key Takeaways

- **`class` vs. `struct`:** The only difference is default access (`private:` for `class`, `public:` for `struct`). Use `struct` for pure data containers.
- **Access Sections:** C++ uses access specifiers (`public:`, `private:`) to define blocks of accessibility within the class body.
- **Destructors are Deterministic:** The destructor (`~T()`) is called automatically and deterministically when an object with automatic storage duration leaves its scope. This is the **release mechanism** of RAII.
- **Member Initializer List:** Always use the **Member Initializer List** for member initialization to ensure direct construction (efficiency) rather than default construction followed by assignment.
- **Designated Initializers (C++20):** Use Designated Initializers (`.member = value`) for initializing simple structs and aggregates to improve clarity and prevent order-based bugs.

Exercises

1. **Destructor Observation:** Create a class `Resource` that prints a message in its constructor and a different message in its destructor. In `main()`, create an instance of `Resource` inside an inner `if` block

(`{ ... }`).

   - *Task:* Observe when the destructor message appears. Explain why its execution is **deterministic** (guaranteed to happen at a specific point).
   - *Hint:* The destructor is called precisely when the object is destroyed (when its scope is exited).

2. **Initializer List Refactoring:** Create a class `User` with two `std::string` members: `first_name` and `last_name`. Write a constructor that accepts two strings.

   - *Task 1 (Incorrect):* Implement the constructor using assignment in the body (`this->first_name = first;`).
   - *Task 2 (Correct):* Implement the constructor using the Member Initializer List (`: first_name(first), last_name(last)`).
   - *Hint:* The first approach involves two unnecessary default constructions and two assignments; the second involves two efficient direct initializations.

3. **Access Specifier Structure:** Define a `Configuration` class that has three sections: `public:` (for the main `load()` method), `protected:` (for configuration members that derived classes need to read), and `private:` (for a helper method `parse_internal_data()`).

   - *Task:* Structure the class using the C++ style of access specifiers as section labels.
   - *Hint:* The keywords must be followed by a colon (`:`).

4. **C++20 Designated Initialization:** Define a simple `struct EmployeeRecord` with public members `id` (int), `salary` (double), and `department` (std::string).

   - *Task:* Initialize an instance of `EmployeeRecord` using C++20 designated initializers, intentionally listing the members in a different order than their declaration.
   - *Hint:* Use the syntax `{.member = value}`.

Where to go Next

- **Part I:: The C++ Ecosystem and Foundation:** This section establishes the philosophical and technical underpinnings of C++, focusing on compilation, linking, and the modern modularization system.
- **Part II: Core Constructs, Classes, and Basic I/O:** Here, we cover the essential C++ syntax, focusing on differences in data types, scoping, **`const` correctness**, and the function of **lvalue references**.
- **Part III: The C++ Memory Model and Resource Management:** The most critical section, which deeply explores raw pointers, value categories, **move semantics**, and the indispensable role of **smart pointers** and the **RAII** idiom.
- **Part IV: Classical OOP, Safety, and Type Manipulation:** This part addresses familiar object-oriented concepts like **inheritance** and **polymorphism**, emphasizing C++'s rules for **exception safety** and type-safe casting.
- **Part V: Genericity, Modern Idioms, and The Standard Library:** Finally, we explore the advanced capabilities of **templates**, **C++20 Concepts**, **lambda expressions**, and the power of the **Standard Library containers** and **Ranges** for highly generic and expressive code.
- **Appendix:** Supplementary materials including coding style guidelines, compiler flags, and further reading.