# C++ for Beginners

readbytes

# C++ for Beginners

From Novice to Advanced Programmer

readbytes.github.io

2025-07-23

This page is intentionally left blank.

# Contents

# Chapter 1.

# Introduction to C

1. What is C++ and Why Learn It?
2. Setting Up Your Development Environment (Compilers and IDEs)
3. Writing and Running Your First C++ Program
4. Structure of a C++ Program and Basic Syntax

# 1 Introduction to C

## 1.1 What is C++ and Why Learn It?

C++ is one of the most powerful and versatile programming languages in the world. It is a general-purpose language widely used for creating software that ranges from small applications to complex systems. Understanding what C++ is and why it remains highly relevant today will help you appreciate its role in modern programming and motivate you to learn it.

### 1.1.1 A Brief History and Evolution of C

C++ was developed by Bjarne Stroustrup in the early 1980s at Bell Labs as an extension of the C programming language. The goal was to combine the efficiency and control of C with the ability to organize complex software projects better through new programming paradigms.

C, developed in the early 1970s by Dennis Ritchie, was originally designed for system programming and operating systems like UNIX. Although extremely fast and efficient, C is a procedural language — meaning it focuses on a sequence of commands and functions but lacks advanced features for modeling complex data and behaviors.

C++ introduced **object-oriented programming (OOP)** to C by adding concepts like classes, objects, inheritance, and polymorphism. This allowed developers to design software in a way that mirrors real-world objects and relationships, making code easier to manage, extend, and reuse.

Over the decades, C++ has evolved through standardization efforts, such as the C++98, C++03, C++11, C++14, C++17, and C++20 standards, each bringing new features and improvements to the language. Today, C++ remains actively developed and widely used, benefiting from a rich ecosystem of libraries and tools.

### 1.1.2 Why is C Important in Modern Software Development?

Despite being over 40 years old, C++ continues to play a crucial role in various areas of software development due to its unique combination of features:

### 1.1.3 System Programming

C++ is heavily used in system-level programming — this means software that interacts closely with hardware, such as operating systems, device drivers, and embedded systems. Its ability to manage memory manually and produce highly optimized machine code makes it ideal for performance-critical environments.

### 1.1.4 Game Development

Many popular game engines and games are written in C++ because it provides high performance and control over system resources. Games require fast processing for graphics, physics, and real-time interactions, and C++ delivers this while allowing developers to organize their code cleanly using object-oriented techniques.

### 1.1.5 Application Software

C++ is used to build large-scale applications such as financial trading systems, web browsers (e.g., parts of Google Chrome), and office software where performance and responsiveness are essential. Its portability allows applications to run on multiple platforms — from Windows and Linux to macOS and embedded devices — with minimal changes.

### 1.1.6 High-Performance Computing

In fields like scientific research, engineering, and artificial intelligence, C++ is chosen to write software that requires heavy computation. Its ability to work close to the hardware and optimize resource usage is crucial for simulations, data processing, and algorithm implementation.

### 1.1.7 Key Features That Make C Stand Out

- **Object-Oriented Programming (OOP):** C++ enables organizing code into classes and objects, promoting reusable and maintainable code through encapsulation, inheritance, and polymorphism.
- **Performance:** Unlike many higher-level languages, C++ allows direct manipulation of memory and system resources. This means programs can be extremely fast and efficient.

- **Portability:** C++ code can be compiled on many different platforms, making it easy to develop cross-platform applications.
- **Rich Standard Library:** The Standard Template Library (STL) provides ready-to-use data structures and algorithms, speeding up development.
- **Compatibility with C:** Since C++ is a superset of C, it can easily incorporate existing C code, making it easier to leverage legacy systems or libraries.

### 1.1.8 Career Opportunities and Practical Applications

Learning C++ opens doors to a wide variety of career paths. Many industries highly value C++ programmers for roles such as:

- **Software Developer:** Building desktop, mobile, or embedded applications.
- **Game Developer:** Creating graphics engines, gameplay mechanics, and real-time simulations.
- **Systems Programmer:** Working on operating systems, network software, or hardware drivers.
- **Financial Software Engineer:** Developing high-frequency trading algorithms and risk management systems.
- **Embedded Systems Engineer:** Programming microcontrollers and IoT devices.
- **Research and Development:** Implementing advanced algorithms in AI, robotics, and scientific computing.

Moreover, understanding C++ provides a strong foundation to learn other programming languages and paradigms. The concepts you master in C++—such as memory management, object-oriented design, and efficient algorithm implementation—are valuable skills across all software development.

## 1.2 Setting Up Your Development Environment (Compilers and IDEs)

Before you start writing C++ programs, it's essential to set up a proper development environment on your computer. A development environment consists mainly of a **compiler** to translate your C++ code into executable programs, and often an **Integrated Development Environment (IDE)** to help you write, compile, and debug your code more easily.

This section will guide you through choosing and installing popular C++ compilers and IDEs, with step-by-step instructions for Windows, macOS, and Linux. We will also show you how to compile and run your first simple program from both the command line and an IDE.

### 1.2.1 What Are Compilers and IDEs?

- **Compiler:** A program that converts your human-readable C++ code into machine code that your computer can execute. Common C++ compilers include:
  - **GCC (GNU Compiler Collection):** Popular on Linux and also available on Windows and macOS.
  - **Clang:** A modern compiler often used on macOS and Linux.
  - **MSVC (Microsoft Visual C++):** The Microsoft compiler integrated into Visual Studio on Windows.
- **IDE (Integrated Development Environment):** A software application that provides tools to write, edit, compile, and debug your code in one place. Examples:
  - **Visual Studio** (Windows)
  - **Code::Blocks** (cross-platform)
  - **Visual Studio Code (VS Code)** (cross-platform, lightweight editor with extensions)

### 1.2.2 Step 1: Choose and Install a Compiler

**Windows**

**Option 1: Install MSVC via Visual Studio**

1. Go to the Visual Studio download page.
2. Download the **Visual Studio Community** edition (free).
3. Run the installer.
4. During setup, select the **"Desktop development with C++"** workload. This will install MSVC compiler and the necessary tools.
5. Complete the installation.

*MSVC* integrates seamlessly with Visual Studio IDE.

**Option 2: Install GCC using MinGW-w64**

1. Visit MinGW-w64.
2. Download the latest version of the MinGW-w64 installer.
3. Run the installer and choose the architecture (usually x86_64 for 64-bit systems).
4. Follow the prompts to install.
5. Add the path to the `bin` folder inside MinGW-w64 to your system's **PATH** environment variable. This lets you run `g++` from the command line.

**macOS**

**Option 1: Install Xcode Command Line Tools (includes Clang)**

1. Open **Terminal**.
2. Type the following command and press Enter:
   ```
   xcode-select --install
   ```
3. Follow the prompts to install Xcode Command Line Tools, which include the `clang` compiler.
4. You can verify installation by typing:
   ```
   clang --version
   ```

**Option 2: Install GCC with Homebrew**

1. If you don't have Homebrew, install it by pasting this command in Terminal:
   ```
   /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
   ```
2. Install GCC using Homebrew:
   ```
   brew install gcc
   ```

**Linux**

Most Linux distributions come with GCC pre-installed. To check or install it:

- On **Ubuntu/Debian**:
  ```
  sudo apt update
  sudo apt install build-essential
  ```
  This installs GCC, G++, and related tools.
- On **Fedora**:
  ```
  sudo dnf groupinstall "Development Tools"
  ```
- Verify installation by running:
  ```
  g++ --version
  ```

### 1.2.3 Step 2: Choose and Install an IDE or Text Editor

You can write C++ programs in any text editor, but IDEs provide many features like syntax highlighting, auto-completion, and debugging.

### 1.2.4 Visual Studio (Windows)

- If you installed MSVC via Visual Studio, you already have a powerful IDE.
- Launch Visual Studio, create a new **Console App** project, and you're ready to write C++.

### 1.2.5 Code::Blocks (Cross-platform)

1. Go to the Code::Blocks download page.
2. Download the version with the **MinGW** compiler bundled for Windows, or just the IDE for macOS/Linux.
3. Install and launch Code::Blocks.
4. Configure your compiler if prompted.

### 1.2.6 Visual Studio Code (VS Code) (Cross-platform)

1. Download VS Code from code.visualstudio.com.
2. Install VS Code.
3. Add the **C/C++ extension** by Microsoft from the Extensions Marketplace.
4. Configure your compiler in VS Code settings or via tasks.

VS Code is a lightweight editor, ideal if you prefer a customizable setup.

### 1.2.7 Step 3: Compile and Run Your First C Program

**Using the Command Line**

1. Open your terminal or command prompt.
2. Create a file named `hello.cpp` with this code:

   "'cpp run #include

   int main() { std::cout « "Hello, C++ World!" « std::endl; return 0; } "'

3. Navigate to the folder containing `hello.cpp`.

**Compile and run with GCC or Clang:**

- Compile:

  ```
  g++ hello.cpp -o hello
  ```

- Run:
  - On Windows:
    ```
    hello.exe
    ```
  - On macOS/Linux:
    ```
    ./hello
    ```

**Compile and run with MSVC (Developer Command Prompt):**

- Open **Developer Command Prompt for Visual Studio**.
- Compile:
  ```
  cl hello.cpp
  ```
- Run:
  ```
  hello.exe
  ```

You should see the output:

```
Hello, C++ World!
```

**Using an IDE**

**Visual Studio:**

- Open Visual Studio.
- Create a new Console App project.
- Replace the default code with the above example or write your own.
- Press **Ctrl + F5** to build and run.

**Code::Blocks:**

- Create a new Console Application project.
- Add or replace with the `hello.cpp` code.
- Click **Build and Run** (green play button).

**VS Code:**

- Open the folder with `hello.cpp`.
- Use the terminal inside VS Code to compile the program.
- Alternatively, configure build tasks for one-click compilation and debugging.

## 1.3 Writing and Running Your First C++ Program

Welcome! Now that you have your C++ development environment set up, it's time to write and run your very first program — the classic **"Hello, World!"**. This simple program is a tradition in programming, serving as a gentle introduction to writing, compiling, and executing code.

Let's walk through every step together.

### 1.3.1 Writing Your First C Program: Hello, World!

Open your text editor or IDE and create a new file called `hello.cpp`. Type or paste the following code:

Full runnable code:

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

### 1.3.2 What Does This Code Do?

Let's break down each line:

- `#include <iostream>` This line tells the compiler to include the **iostream** library, which provides input and output functionality. We need this to use `std::cout` to display text on the screen.
- `int main() {` This is the **main function**, the starting point of every C++ program. When you run your program, execution begins here. The `int` means this function returns an integer value to the operating system when it finishes.
- `std::cout << "Hello, World!" << std::endl;` This line outputs the text `Hello, World!` to the console (your screen).
  - `std::cout` is the standard output stream.
  - `<<` is the insertion operator, which sends data to the output stream.
  - `"Hello, World!"` is a string literal — the text you want to display.
  - `std::endl` inserts a newline character, moving the cursor to the next line.
- `return 0;` This tells the program to return the value `0` to the operating system, signaling that the program finished successfully.
- `}` This closes the main function.

### 1.3.3 Compiling and Running Your Program

Using the Command Line

1. Save your file as `hello.cpp`.
2. Open your terminal or command prompt.
3. Navigate to the folder containing `hello.cpp`. For example, if your file is in `C:\Projects` on Windows:

   ```
   cd C:\Projects
   ```

   On macOS/Linux:

   ```
   cd ~/Projects
   ```

4. Compile the program using your installed compiler.

- If you're using **GCC** or **Clang**, type:

  ```
  g++ hello.cpp -o hello
  ```

  This compiles the program and creates an executable named `hello` (or `hello.exe` on Windows).

- If you're using **MSVC** in the Developer Command Prompt, type:

  ```
  cl hello.cpp
  ```

5. Run the executable:

- On Windows (Command Prompt):

  ```
  hello.exe
  ```

- On macOS/Linux (Terminal):

  ```
  ./hello
  ```

You should see:

```
Hello, World!
```

### 1.3.4 Using an IDE

If you are using an IDE like Visual Studio, Code::Blocks, or VS Code:

- Create a new Console Application or open a new file named `hello.cpp`.
- Paste the code.
- Build and run your program using the IDE's build/run button or menu option.
- The output console should display your message.

### 1.3.5 Troubleshooting Common Issues

- **Compiler not found or command not recognized:** Make sure your compiler is installed and its `bin` directory is added to your system's PATH environment variable. Refer back to Chapter 1, Section 2 for setup instructions.
- **Syntax errors:** Check for missing semicolons (`;`), unmatched braces (`{}`), or misspelled keywords like `int` or `return`. The compiler error messages usually point to the exact line.
- **Program runs but no output:** Ensure you used `std::cout` correctly and included `#include <iostream>`. Also, confirm you ran the latest compiled executable.

### 1.3.6 Experimenting with Your First Program

Once your program runs successfully, feel free to experiment!

Try modifying the string to print different messages. For example:

```
std::cout << "Welcome to C++ programming!" << std::endl;
```

Or add multiple output lines:

```
std::cout << "Hello!" << std::endl;
std::cout << "This is my first program." << std::endl;
```

Try changing the program to perform simple tasks, like printing numbers:

```
std::cout << "The answer is: " << 42 << std::endl;
```

This experimentation helps you get comfortable with syntax and output, and builds confidence as you prepare for more complex programs.

## 1.4 Structure of a C++ Program and Basic Syntax

Now that you have written and run your first C++ program, it's important to understand the fundamental structure and syntax rules that form the backbone of every C++ application. This section will introduce you to the basic components of a C++ program, how code is organized, and important syntax conventions you need to follow. Understanding these will help you write clean, correct, and maintainable programs.

### 1.4.1 The Fundamental Structure of a C++ Program

Every C++ program typically consists of the following parts:

1. **Preprocessor Directives (Headers)**
2. **The main() Function**
3. **Statements**
4. **Blocks**

Let's look at each of these with simple examples.

### 1.4.2 Preprocessor Directives (Headers)

At the very top of your program, you often see lines beginning with `#`, such as:

```
#include <iostream>
```

This is called a **preprocessor directive**. It instructs the compiler to include the contents of a library or header file before compilation. Headers provide declarations of functions, classes, and variables you want to use.

- `#include <iostream>` allows you to use input/output features like `std::cout` for printing text.
- You can include multiple headers if you need functionalities from different libraries.

### 1.4.3 The main() Function: The Entry Point

Every C++ program must have exactly one **main** function, which is the starting point where execution begins:

```
int main() {
    // code statements go here
    return 0;
}
```

- `int` indicates the function returns an integer value to the operating system when the program ends.
- The parentheses `()` after `main` indicate that it is a function. In this basic form, it takes no parameters.
- Curly braces `{}` define the **body** or **block** of the function, which contains the code statements to execute.
- `return 0;` tells the OS the program finished successfully.

### 1.4.4 Statements

A **statement** is a single instruction that the computer executes. Each statement in C++ ends with a **semicolon ;**:

```
std::cout << "Hello, World!" << std::endl;
```

- Missing the semicolon causes a compilation error because the compiler won't know where the statement ends.
- You can have multiple statements inside a function or block.

### 1.4.5 Blocks and Braces {}

Curly braces {} group multiple statements into a **block**, defining a scope for variables and control structures.

Example:

```
int main() {
    std::cout << "First line." << std::endl;
    std::cout << "Second line." << std::endl;
    return 0;
}
```

The two output statements are enclosed in the braces, indicating they belong to the `main` function.

Blocks can also be nested inside other blocks, which is important for controlling program flow.

### 1.4.6 Basic Syntax Rules and Conventions

#### Semicolons ;

- Every statement must end with a semicolon.
- Forgetting a semicolon is a common error that will cause the compiler to fail and produce error messages.

Example:

```
int a = 5;     // Correct
int b = 10     // Missing semicolon - causes error
```

#### Whitespace and Formatting

- C++ ignores extra **whitespace** (spaces, tabs, newlines) except when it separates tokens (like keywords and variable names).
- You can use whitespace freely to improve readability.

These are all valid and equivalent:

```
int a=5;
int a = 5;
int     a    =    5;
```

Good formatting and indentation help others (and yourself) read your code.

### Case Sensitivity

C++ is **case sensitive**, meaning that uppercase and lowercase letters are treated as different characters.

For example:

```
int Number = 5;
int number = 10;

std::cout << Number << std::endl;  // Prints 5
std::cout << number << std::endl;  // Prints 10
```

- `Number` and `number` are two distinct variables.
- Keywords such as `int`, `return`, and `main` must be typed in lowercase.

### Comments

Comments are notes for humans and are ignored by the compiler. They help explain your code.

- **Single-line comment:**

```
// This is a comment
```

- **Multi-line comment:**

```
/*
  This is a
  multi-line comment
*/
```

Use comments to clarify complex code or mark sections.

### 1.4.7 Common Pitfalls to Avoid

- **Missing semicolons:** Always end statements with `;`.
- **Mismatched braces:** Each `{` must have a corresponding `}`. IDEs usually help highlight matching braces.
- **Incorrect capitalization:** Write keywords exactly as required, e.g., `int` not `Int`.
- **Using undeclared variables:** Declare variables before use (we'll cover this in detail later).
- **Including headers incorrectly:** Use angle brackets `< >` for standard libraries (e.g., `<iostream>`) and quotes `" "` for your own files.

### 1.4.8 Small Example Putting It All Together

Full runnable code:

```
#include <iostream>     // Include input/output library

int main() {            // Main function starts here
    std::cout << "Welcome to C++!" << std::endl;  // Print message
    return 0;           // Indicate successful program termination
}
```

### 1.4.9 Summary

Understanding the basic structure and syntax of a C++ program is key to writing correct code. Remember these points:

- Start with **header includes** to access libraries.
- Your program must have one **main() function** where execution starts.
- Use **statements** to tell the computer what to do; end each with a **semicolon**.
- Group statements inside **blocks** using curly braces `{}`.
- Pay attention to **case sensitivity** and use **whitespace** to make your code readable.
- Add **comments** to explain your code.

# Chapter 2.

# Basic Syntax and Data Types

1. Variables and Constants
2. Fundamental Data Types (`int`, `char`, `float`, `double`, `bool`)
3. Input and Output (`cin`, `cout`)
4. Comments and Naming Conventions

# 2 Basic Syntax and Data Types

## 2.1 Variables and Constants

In programming, **variables** and **constants** are fundamental concepts that allow you to store and manipulate data. Understanding how to declare, initialize, and use variables and constants is one of the first steps to writing meaningful C++ programs.

### 2.1.1 What Are Variables?

A **variable** is a named storage location in your computer's memory that holds a value. Think of a variable as a labeled box where you can put information, and later retrieve or change it.

For example, if you need to store a person's age, you can create a variable called `age` and assign a value to it.

### 2.1.2 Declaring Variables

Before you use a variable, you must **declare** it. Declaration tells the compiler the variable's **name** and **type** (what kind of data it will hold).

Example:

```
int age;
```

Here, `int` is the type (integer numbers), and `age` is the variable's name. At this point, the variable exists but does not have a value yet.

### 2.1.3 Initializing Variables

You can **initialize** a variable by assigning a value when you declare it:

```
int age = 25;
```

This sets the variable `age` to hold the value 25 from the start.

You can also declare first and assign a value later:

```
int age;
age = 25;
```

### 2.1.4 Rules for Naming Variables

When naming variables in C++, follow these rules:

- Names can contain letters (A–Z, a–z), digits (0–9), and underscores (`_`).
- Names must **begin with a letter or underscore**, not a digit.
- Names are **case sensitive** (`Age` and `age` are different variables).
- Avoid using C++ **keywords** (like `int`, `return`, `for`) as variable names.
- Choose descriptive names that reflect the variable's purpose, e.g., `totalScore` instead of `x`.

Example of valid names:

```
int score;
float temperature;
bool isFinished;
int _count;
```

Example of invalid names:

```
int 3dModel;      // Starts with a digit - invalid
int total-score;  // Hyphens not allowed
int return;       // Keyword - invalid
```

### 2.1.5 What Are Constants?

Sometimes, you need values that **never change** throughout your program. These are called **constants**. Using constants improves code readability and helps prevent accidental modification of important values.

In C++, you declare constants using the **`const`** keyword.

Example:

```
const double PI = 3.14159;
```

Here, `PI` is a constant holding the value of pi. Trying to change `PI` later in the code will cause a compilation error.

### 2.1.6 Using `const` and `constexpr`

- **`const`** marks a variable as constant — once initialized, it cannot be changed.
- **`constexpr`** indicates that the value is a **compile-time constant**, meaning its value can be evaluated by the compiler before the program runs. This can enable certain optimizations.

Example with `constexpr`:

```cpp
constexpr int maxUsers = 100;
```

This tells the compiler that `maxUsers` is a constant known at compile time.

Use `constexpr` when the value should be constant and usable in contexts that require compile-time constants, such as array sizes.

### 2.1.7 Practical Examples: Declaring and Using Variables and Constants

Example 1: Using Variables

Full runnable code:

```cpp
#include <iostream>

int main() {
    int age = 30;               // Declare and initialize an integer variable
    double height = 1.75;       // Declare a double variable for height in meters
    bool isStudent = true;      // Boolean variable to indicate status

    std::cout << "Age: " << age << std::endl;
    std::cout << "Height: " << height << " meters" << std::endl;
    std::cout << "Is student: " << isStudent << std::endl;

    age = 31;                   // Change the value of age
    std::cout << "Next year age: " << age << std::endl;

    return 0;
}
```

**Output:**

```
Age: 30
Height: 1.75 meters
Is student: 1
Next year age: 31
```

Note: In C++, `true` prints as `1` and `false` as `0` when output using `std::cout`.

Example 2: Using Constants

Full runnable code:

```
#include <iostream>

int main() {
    const double PI = 3.14159;       // Constant value of Pi
    constexpr int maxScore = 100;    // Compile-time constant

    int score = 85;
    double circumference = 2 * PI * 10;  // Calculate circumference for radius 10

    std::cout << "Score: " << score << "/" << maxScore << std::endl;
    std::cout << "Circumference of circle with radius 10: " << circumference << std::endl;

    // Uncommenting the following line will cause an error:
    // PI = 3.14;    // Error! Cannot modify a const variable

    return 0;
}
```

### 2.1.8 Why Use Constants?

- **Safety:** Prevents accidental changes to important values.
- **Readability:** Makes code easier to understand when values have meaningful names.
- **Maintainability:** If you need to change the value, update it in one place rather than multiple locations.
- **Optimization:** Some compilers can optimize `constexpr` values better during compilation.

### 2.1.9 Summary

- **Variables** are named storage locations in memory that can hold data and change over time.
- Variables must be **declared** with a type and can be optionally **initialized** with a value.
- Follow naming rules: start with a letter or underscore, case-sensitive, avoid keywords.
- **Constants** hold values that do not change, declared with `const` or `constexpr`.
- Using constants improves code safety, clarity, and sometimes performance.
- Practice declaring and using both variables and constants to solidify your understanding.

With these basics of variables and constants, you are ready to start storing and manipulating data in your C++ programs. In the next section, we will explore the fundamental data types that variables can hold, such as integers, characters, and floating-point numbers.

## 2.2 Fundamental Data Types (`int`, `char`, `float`, `double`, `bool`)

Understanding data types is essential in C++ because they define what kind of data a variable can store and how much memory it uses. C++ offers several **fundamental data types** that serve as the building blocks for all programming tasks. In this section, we will explore the most common fundamental types: **integers**, **characters**, **floating-point numbers**, and **booleans**. You will learn about their purpose, storage sizes, typical ranges, and important nuances such as type modifiers and pitfalls to watch for.

### 2.2.1 Integer Types (`int`)

Integers are whole numbers without any fractional part. They can be positive, negative, or zero.

### 2.2.2 Declaration and Usage

```cpp
int age = 25;
int year = 2025;
int temperature = -10;
```

Here, `int` declares variables to store integer values.

### 2.2.3 Storage Size and Range

- The size of `int` typically depends on your system but is often **4 bytes (32 bits)**.
- A signed 32-bit integer ranges approximately from **-2,147,483,648** to **2,147,483,647**.

### 2.2.4 Type Modifiers: `signed` and `unsigned`

- **Signed integers** (default) can store negative and positive numbers.
- **Unsigned integers** store only non-negative numbers but can represent roughly twice the positive range.

Example:

```cpp
unsigned int positiveNumber = 4000000000;  // large positive number
signed int negativeNumber = -100;
```

### 2.2.5 Important Considerations: Overflow

If you assign a value outside the allowed range, **overflow** occurs, which leads to unexpected results.

Example:

```
unsigned int maxVal = 4294967295;  // Max value for 32-bit unsigned int
maxVal = maxVal + 1;               // Overflow! Wraps around to 0
```

**Tip:** Use `unsigned` only when you know the variable will never be negative.

### 2.2.6 Character Type (`char`)

The `char` type stores a single character, such as a letter, digit, or symbol. Characters are internally represented by numeric codes based on the ASCII (or Unicode) standard.

### 2.2.7 Declaration and Usage

```
char grade = 'A';
char symbol = '#';
char digit = '7';
```

- Characters are enclosed in **single quotes `' '`**.
- You cannot assign a string (multiple characters) to a `char`.

### 2.2.8 Storage Size and Range

- A `char` typically takes **1 byte (8 bits)**.
- It can represent values from **-128 to 127** (signed `char`) or **0 to 255** (unsigned `char`).

### 2.2.9 Using `char` as Numbers

Since `char` stores numeric codes, you can use them in arithmetic operations:

```
char letter = 'A';            // ASCII code 65
int code = letter + 1;        // 66
char nextLetter = static_cast<char>(code);  // 'B'
```

### 2.2.10 Floating-Point Types (`float` and `double`)

What Are Floating-Point Numbers?

These types store numbers with fractional parts, such as decimals or scientific notation values.

### 2.2.11 `float`

- Single precision floating-point.
- Usually **4 bytes (32 bits)**.
- Approximate range: $\pm 1.5 \times 10$ to $\pm 3.4 \times 10^{3}$ .
- About 6-7 digits of precision.

Example:

```
float temperature = 36.6f;
```

**Note:** The `f` suffix indicates the literal is a `float` (not double).

### 2.2.12 `double`

- Double precision floating-point.
- Usually **8 bytes (64 bits)**.
- Approximate range: $\pm 5.0 \times 10^{32}$ to $\pm 1.7 \times 10^{3}$ .
- About 15-16 digits of precision.

Example:

```
double pi = 3.141592653589793;
```

Use `double` when more precision or range is needed.

### 2.2.13 Boolean Type (`bool`)

A `bool` variable holds one of two possible values: **true** or **false**. It is used to represent logical conditions and control program flow.

### 2.2.14 Declaration and Usage

```cpp
bool isFinished = false;
bool hasPassed = true;
```

When output via `std::cout`, `true` prints as `1` and `false` as `0`.

### 2.2.15 Storage Size

- Typically stored as **1 byte**, but this can vary.

### 2.2.16 Default Values

Variables declared inside functions without explicit initialization have **undefined default values** and may contain garbage data. Always initialize your variables to avoid unpredictable behavior.

Example:

```cpp
int count;        // Uninitialized, value is undefined
int total = 0;    // Initialized to zero
```

Global or static variables are initialized to zero by default.

### 2.2.17 Summary Table of Common Fundamental Types

| Type | Size (Typical) | Range (Signed) | Range (Unsigned) | Example Value |
|---|---|---|---|---|
| `int` | 4 bytes | -2,147,483,648 to 2,147,483,647 | 0 to 4,294,967,295 | `int age = 30;` |
| `char` | 1 byte | -128 to 127 | 0 to 255 | `char letter = 'A';` |
| `float` | 4 bytes | $\pm 1.5 \times 10$ to $\pm 3.4 \times 10^3$ | N/A | `float temp = 36.6f;` |
| `double` | 8 bytes | $\pm 5.0 \times 10^{32}$ to $\pm 1.7 \times 10^3$ | N/A | `double pi = 3.14159;` |
| `bool` | 1 byte (typ.) | `false` or `true` | N/A | `bool flag = true;` |

### 2.2.18 Examples Using Fundamental Data Types

Full runnable code:

```cpp
#include <iostream>

int main() {
    int age = 28;
    char grade = 'B';
    float height = 1.82f;
    double weight = 75.5;
    bool isStudent = false;

    std::cout << "Age: " << age << std::endl;
    std::cout << "Grade: " << grade << std::endl;
    std::cout << "Height: " << height << " meters" << std::endl;
    std::cout << "Weight: " << weight << " kg" << std::endl;
    std::cout << "Is student? " << isStudent << std::endl;

    return 0;
}
```

### 2.2.19 Potential Pitfalls

- **Overflow and Underflow:** Assigning values outside the allowed range causes wrap-around or data loss.
- **Precision Loss:** Floating-point numbers cannot represent all decimals exactly; operations may introduce small errors.
- **Signed vs. Unsigned Mixing:** Mixing signed and unsigned types in expressions can lead to unexpected results.
- **Uninitialized Variables:** Always initialize variables before use to avoid unpredictable behavior.

### 2.2.20 Conclusion

Mastering C++ fundamental data types is crucial because they define how data is stored and manipulated. Choosing the appropriate type for your data affects program correctness, efficiency, and readability.

- Use `int` for whole numbers.
- Use `char` for individual characters.
- Use `float` or `double` for decimal numbers, preferring `double` when precision matters.
- Use `bool` for true/false conditions.

In upcoming chapters, we will build upon these fundamentals to introduce more complex data types and operations.

## 2.3 Input and Output (`cin`, `cout`)

One of the fundamental abilities of any program is to **interact with the user** — getting input and displaying output. In C++, this is typically done using the **standard input and output streams** provided by the language: `cin` for input and `cout` for output.

In this section, you will learn how to use these streams to perform basic input and output (I/O) operations, understand the syntax, and see practical examples covering different data types. We'll also discuss some common pitfalls and tips to handle user input correctly.

### 2.3.1 Output with `cout`

`cout` stands for **console output** and is used to send data from your program to the console (usually your terminal or command prompt window).

### 2.3.2 The Insertion Operator `<<`

To send data to `cout`, use the **insertion operator `<<`**, which "inserts" the data into the output stream.

Example:

Full runnable code:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Here:

- `"Hello, World!"` is sent to the output.
- `std::endl` inserts a newline character and flushes the output buffer, moving the cursor to the next line.

You can chain multiple insertions:

```cpp
int age = 30;
std::cout << "Age: " << age << std::endl;
```

This prints:

```
Age: 30
```

### 2.3.3 Input with `cin`

`cin` stands for **console input** and reads data entered by the user from the keyboard.

### 2.3.4 The Extraction Operator `>>`

To get data from the user, use the **extraction operator `>>`**, which "extracts" data from the input stream into variables.

Example:

Full runnable code:

```cpp
#include <iostream>

int main() {
    int age;
    std::cout << "Enter your age: ";
    std::cin >> age;
    std::cout << "You entered: " << age << std::endl;
    return 0;
}
```

The program:

1. Prints the prompt `"Enter your age: "` without moving to a new line.
2. Waits for the user to type a number and press Enter.
3. Stores the entered value into the variable `age`.
4. Prints the value back.

### 2.3.5 Working with Different Data Types

You can use `cin` and `cout` with many basic data types, such as `int`, `double`, `char`, and `bool`.

Example:

Full runnable code:

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

```cpp
int main() {
    int number;
    double price;
    char grade;
    bool isRegistered;

    std::cout << "Enter an integer: ";
    std::cin >> number;

    std::cout << "Enter a price (decimal): ";
    std::cin >> price;

    std::cout << "Enter a grade (single character): ";
    std::cin >> grade;

    std::cout << "Are you registered? (1 for yes, 0 for no): ";
    std::cin >> isRegistered;

    std::cout << "\nYou entered:" << std::endl;
    std::cout << "Integer: " << number << std::endl;
    std::cout << "Price: " << price << std::endl;
    std::cout << "Grade: " << grade << std::endl;
    std::cout << "Registered: " << std::boolalpha << isRegistered << std::endl;

    return 0;
}
```

Notes:

- `std::boolalpha` makes `cout` print `true` or `false` instead of `1` or `0` for booleans.
- Input expects matching data types. Typing invalid input may cause errors or unexpected behavior.

### 2.3.6 Common Issues and Tips with Input

Input Buffering and Newline Characters

When you enter data and press Enter, the input buffer may contain leftover characters (like the newline `\n`) that can interfere with subsequent inputs, especially when mixing different types or reading characters and strings.

Example problem:

```cpp
char grade;
int age;

std::cin >> age;      // User types: 25 [Enter]
std::cin >> grade;    // May skip input because of leftover newline
```

**Solution:**

Use `std::cin.ignore()` to discard unwanted characters:

```cpp
std::cin >> age;
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Clear input buffer
std::cin >> grade;
```

This clears the input buffer up to the next newline.

### 2.3.7 Input Validation and Error Checking

Users may enter invalid data, causing `cin` to fail.

Example:

```cpp
int age;
std::cin >> age;
if(std::cin.fail()) {
    std::cout << "Invalid input! Please enter a number." << std::endl;
}
```

If `cin` fails (e.g., user types letters instead of numbers):

- `std::cin.fail()` returns `true`.
- The input stream enters a **fail state** and stops further input operations.

**How to recover:**

```cpp
std::cin.clear();  // Clear error flags
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Discard invalid input
```

Then prompt the user to try again.

### 2.3.8 Reading Strings with Spaces

`std::cin >>` reads input until the first whitespace (space, tab, newline), so it cannot read multi-word strings.

Example:

```cpp
std::string name;
std::cin >> name;   // If user types "John Doe", only "John" is stored
```

To read entire lines including spaces, use `std::getline`:

```cpp
std::string fullName;
std::getline(std::cin, fullName);
```

### 2.3.9 Summary of `cin` and `cout` Operators

| Operation | Operator | Description |
|---|---|---|
| Output | << | Inserts data into output stream |
| Input | >> | Extracts data from input stream |

### 2.3.10 Simple Complete Example

Full runnable code:

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

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

    std::cout << "What is your name? ";
    std::getline(std::cin, name);  // Read full name with spaces

    std::cout << "How old are you? ";
    std::cin >> age;

    std::cout << "Hello, " << name << "! You are " << age << " years old." << std::endl;

    return 0;
}
```

### 2.3.11 Conclusion

The `cin` and `cout` streams are your primary tools for interacting with users in console applications. Mastering the insertion (`<<`) and extraction (`>>`) operators enables you to easily read inputs and display outputs of various types.

Keep in mind:

- Use `std::cin` with `>>` for simple, whitespace-delimited input.
- Use `std::getline` for reading entire lines or strings with spaces.
- Always check and handle input errors for robust programs.
- Remember to manage the input buffer carefully to avoid skipped inputs.

## 2.4 Comments and Naming Conventions

Writing clear, readable, and maintainable code is just as important as writing code that works correctly. Two key practices help achieve this goal: **using comments effectively** and **following good naming conventions**. In this section, we will explore how comments improve your code and how to choose meaningful names for variables, functions, and classes.

### 2.4.1 The Importance of Comments

Comments are notes written inside your source code that the compiler **ignores**. They help explain what the code does, why certain decisions were made, or how complex logic works. Good comments improve readability and make it easier for you and others to maintain and update the code later.

### 2.4.2 Types of Comments in C

C++ supports two kinds of comments:

1. **Single-line comments:** Start with `//` and continue to the end of the line.

```
int age = 25;  // Store the user's age
```

2. **Multi-line comments:** Start with `/*` and end with `*/`. They can span multiple lines.

```
/*
  This function calculates the factorial
  of a given non-negative integer using recursion.
*/
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

### 2.4.3 Best Practices for Using Comments

- **Explain why, not what:** Good comments clarify the reasoning behind the code, not just restate what the code clearly shows.

```
// Bad comment:
int total = a + b;  // Add a and b

// Good comment:
int total = a + b;  // Combine scores from two players to get final score
```

- **Keep comments up to date:** Outdated comments can mislead readers and cause confusion.
- **Avoid obvious comments:** If code is self-explanatory, extra comments may be unnecessary.
- **Use comments to separate sections or highlight TODOs:**

```
// Initialize variables
int count = 0;
int maxCount = 100;

// TODO: Optimize this loop for large datasets
```

### 2.4.4 Naming Conventions

Choosing good names for variables, functions, and classes makes your code **self-documenting**, meaning the names themselves convey meaning and purpose.

### 2.4.5 General Naming Rules

- Use only letters (`A-Z`, `a-z`), digits (`0-9`), and underscores (`_`).
- Names **must start** with a letter or underscore.
- Names are **case sensitive** (`totalScore` and `totalscore` are different).
- Avoid using reserved keywords like `int`, `class`, or `return`.
- Use **descriptive** names that explain the role or purpose of the item.

### 2.4.6 Variables

- Use **lowercase** letters.
- For multi-word names, use either:
  - **camelCase** (first word lowercase, subsequent words capitalized): `totalScore`, `userAge`
  - **snake_case** (words separated by underscores): `total_score`, `user_age`

**Examples:**

```
int totalScore = 0;        // Good: descriptive and camelCase
int user_age = 25;         // Good: descriptive and snake_case
int x;                     // Poor: non-descriptive
int t;                     // Poor: ambiguous
```

### 2.4.7 Functions

- Use **verbs or verb phrases** describing the action.
- Use **camelCase** or **snake_case**, matching your project's style.

**Examples:**

```
void calculateTotal();     // Good: clearly describes action
int getUserAge();          // Good: starts with a verb
void doStuff();            // Poor: vague function name
int f();                   // Poor: unclear purpose
```

### 2.4.8 Classes and Types

- Use **PascalCase** (also known as UpperCamelCase): each word starts with a capital letter.
- Class names are usually nouns.

**Examples:**

```
class UserProfile {        // Good: descriptive and PascalCase
class AccountManager {     // Good
class data {               // Poor: lowercase, not PascalCase
class x {                  // Poor: unclear
```

### 2.4.9 Consistency Is Key

Whatever naming style you choose, **be consistent** throughout your codebase. This consistency helps you and others read and maintain the code more easily.

### 2.4.10 Examples: Good vs. Poor Naming

```
// Poor Naming
int a;                  // What does 'a' represent?
void func();            // What action does this function perform?
class d;                // What does 'd' represent?

// Good Naming
int itemCount;          // Number of items in a list
void printReport();     // Prints a report to the screen
class DataProcessor;    // Class that processes data
```

### 2.4.11 Summary

- Use **comments** to clarify intent, explain tricky code, and provide context.
- Use `//` for short comments and `/* ... */` for longer explanations.
- Avoid obvious or outdated comments.
- Choose **meaningful, descriptive names** for variables, functions, and classes.
- Follow naming conventions:
  - Variables/functions: `camelCase` or `snake_case`
  - Classes/types: `PascalCase`
- Keep naming consistent throughout your projects.

Good commenting and naming habits make your code **self-explanatory** and easy to maintain, which is invaluable as programs grow more complex and as you collaborate with others.

# Chapter 3.

# Operators and Expressions

# 3 Operators and Expressions

## 3.1 Arithmetic, Relational, and Logical Operators

Operators are special symbols in C++ that perform operations on variables and values. Understanding how to use **arithmetic**, **relational**, and **logical** operators is essential because they form the building blocks of expressions, conditions, and decision-making in programs.

This section explains these three groups of operators with examples and exercises to help you master their use.

### 3.1.1 Arithmetic Operators

Arithmetic operators allow you to perform basic mathematical calculations.

| Operator | Description | Example | Result |
|---|---|---|---|
| `+` | Addition | `5 + 3` | `8` |
| `-` | Subtraction | `10 - 4` | `6` |
| `*` | Multiplication | `6 * 7` | `42` |
| `/` | Division | `20 / 5` | `4` |
| `%` | Modulus (remainder) | `17 % 5` | `2` |

### 3.1.2 Examples

```cpp
int a = 10;
int b = 3;

int sum = a + b;          // 13
int difference = a - b;  // 7
int product = a * b;      // 30
int quotient = a / b;     // 3 (integer division truncates decimal)
int remainder = a % b;    // 1
```

### 3.1.3 Important Notes

- **Integer division:** When both operands are integers, division truncates the decimal part. For example, `10 / 3` results in `3`, not `3.33`.
- To get floating-point division, use at least one floating-point operand:

```
double result = 10.0 / 3;  // 3.3333...
```

### 3.1.4 Relational Operators

Relational operators compare two values and return a **boolean** result: `true` or `false`.

| Operator | Meaning | Example | Result |
|---|---|---|---|
| `==` | Equal to | `5 == 5` | `true` |
| `!=` | Not equal to | `5 != 3` | `true` |
| `<` | Less than | `4 < 7` | `true` |
| `>` | Greater than | `9 > 12` | `false` |
| `<=` | Less than or equal to | `6 <= 6` | `true` |
| `>=` | Greater than or equal to | `8 >= 10` | `false` |

### 3.1.5 Examples

```
int x = 5;
int y = 8;

bool result1 = (x == y);  // false
bool result2 = (x != y);  // true
bool result3 = (x < y);   // true
bool result4 = (x >= y);  // false
```

These operators are commonly used in **if statements** and loops to control program flow.

### 3.1.6 Logical Operators

Logical operators allow you to combine or invert boolean expressions. They are essential when making complex decisions.

| Operator | Meaning | Description | Example | Result |
|---|---|---|---|---|
| `&&` | Logical AND | True if **both** operands are true | `(true && false)` | `false` |

| Operator | Meaning | Description | Example | Result | | |
|---|---|---|---|---|---|---|
| ‘ | | ‘ | Logical OR | True if **at least one** operand is true | ‘(true | false)‘ `true` |
| ! | Logical NOT | Inverts the boolean value | `!(true)` | `false` | | |

### 3.1.7 Examples

```
bool a = true;
bool b = false;

bool andResult = a && b;   // false, because b is false
bool orResult = a || b;    // true, because a is true
bool notResult = !a;       // false, because a is true
```

### 3.1.8 Using Logical Operators with Relational Expressions

Logical operators are often used to combine relational conditions:

```
int age = 20;
bool isAdult = (age >= 18) && (age < 65);  // true if age between 18 and 64

bool eligibleForDiscount = (age < 18) || (age >= 65); // true if age less than 18 or 65 and older
```

### 3.1.9 Combining Operators in Expressions

Operators can be combined to create more complex expressions, respecting **operator precedence** (covered in a later section).

Example:

```
int a = 10;
int b = 20;
int c = 5;

bool check = (a < b) && ((b / c) > 3);
// (10 < 20) is true
// (20 / 5) > 3 → 4 > 3 is true
// true && true is true
```

### 3.1.10 Exercises

Try solving these simple problems using arithmetic, relational, and logical operators:

1. **Check if a number is even:**

```cpp
int num;
// Input num from user
bool isEven = (num % 2 == 0);
```

2. **Determine if a person is a teenager (age between 13 and 19 inclusive):**

```cpp
int age;
// Input age from user
bool isTeenager = (age >= 13) && (age <= 19);
```

3. **Check if a number is within a range (10 to 50, inclusive):**

```cpp
int value;
// Input value
bool inRange = (value >= 10) && (value <= 50);
```

4. **Check if a character is a vowel (a, e, i, o, u):**

```cpp
char ch;
// Input character
bool isVowel = (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u' ||
                ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U');
```

### 3.1.11 Summary

| Operator Type | Operators | Purpose |
|---|---|---|
| Arithmetic | +, -, *, /, % | Perform mathematical calculations |
| Relational | ==, !=, <, >, <=, >= | Compare values, produce boolean result |
| Logical | &&, \|\|, ! | Combine or invert boolean values |

Mastering these operators allows you to build expressions that perform calculations, make decisions, and control the flow of your C++ programs. Practice combining them to solve different problems and write clear, concise conditional statements.

## 3.2 Assignment and Compound Assignment Operators

In C++, **assignment** operators are used to store values into variables. This is a fundamental operation that allows your program to remember data and update it as needed. In addition to the basic assignment operator `=`, C++ provides **compound assignment operators** that combine arithmetic operations with assignment, making your code shorter and easier to read.

### 3.2.1 The Assignment Operator =

The **assignment operator** `=` assigns the value on the right-hand side to the variable on the left-hand side.

### 3.2.2 Syntax

```
variable = expression;
```

### 3.2.3 Example

```
int x;
x = 10;          // Assigns 10 to x
x = x + 5;       // Adds 5 to x, then stores the result back in x (x becomes 15)
```

When you write `x = x + 5;`, the right side (`x + 5`) is evaluated first, then the result is stored back into `x`.

### 3.2.4 Compound Assignment Operators

Compound assignment operators combine an arithmetic operation with assignment, making the code more concise and often easier to read.

### 3.2.5 List of Common Compound Assignment Operators

| Operator | Meaning | Example | Equivalent To |
|---|---|---|---|
| `+=` | Add right operand to left operand and assign | `x += 5;` | `x = x + 5;` |
| `-=` | Subtract right operand from left operand and assign | `x -= 3;` | `x = x - 3;` |
| `*=` | Multiply left operand by right operand and assign | `x *= 2;` | `x = x * 2;` |

| Operator | Meaning | Example | Equivalent To |
|---|---|---|---|
| `/=` | Divide left operand by right operand and assign | `x /= 4;` | `x = x / 4;` |
| `%=` | Take modulus of left operand by right operand and assign | `x %= 3;` | `x = x % 3;` |

### 3.2.6 Examples

```
int x = 10;

x += 5;   // x = 10 + 5 → x becomes 15
x -= 3;   // x = 15 - 3 → x becomes 12
x *= 2;   // x = 12 * 2 → x becomes 24
x /= 4;   // x = 24 / 4 → x becomes 6
x %= 5;   // x = 6 % 5 → x becomes 1
```

Each operation updates the value of `x` directly, reducing redundancy.

### 3.2.7 Why Use Compound Assignment Operators?

- **Conciseness:** Shorter code with fewer characters.
- **Readability:** Clear intention to update the variable.
- **Maintainability:** Less chance of errors when updating a variable repeatedly.

For example, compare:

```
count = count + 1;  // Verbose
count += 1;         // More concise and clear
```

### 3.2.8 How Expressions Are Evaluated

In a compound assignment like `x += y`, the expression is evaluated as:

1. Evaluate the right-hand side (`y`).
2. Perform the arithmetic operation (`x + y`).
3. Assign the result back to `x`.

This happens in a single, atomic step from the programmer's perspective.

### 3.2.9 Operator Precedence in Assignments

Assignment operators, including compound assignment operators, have **lower precedence** than arithmetic operators.

### 3.2.10 Example

```
int x = 5;
int y = 10;

x += y * 2;  // Multiplication happens before assignment
```

Here's how this expression is evaluated:

1. `y * 2` is evaluated first → `10 * 2 = 20`
2. Then `x += 20` → `x = x + 20` → `5 + 20 = 25`

So, `x` becomes 25.

### 3.2.11 Parentheses Can Control Precedence

```
x += (y * 2);   // Same as above, parentheses clarify grouping

x = (x + y) * 2;  // Different: adds x and y first, then multiplies by 2
```

### 3.2.12 Practical Example: Counting Occurrences

Suppose you want to count how many times a certain event occurs:

```
int count = 0;

// Each time an event happens:
count += 1;  // Increment count by 1
```

Using compound assignment makes this simple and readable.

### 3.2.13 Summary

| Operator Type | Syntax Example | Meaning |
|---|---|---|
| Assignment | `x = 10;` | Assign 10 to `x` |
| Compound Addition | `x += 5;` | Add 5 to `x` and assign |
| Compound Subtraction | `x -= 3;` | Subtract 3 from `x` and assign |
| Compound Multiplication | `x *= 2;` | Multiply `x` by 2 and assign |
| Compound Division | `x /= 4;` | Divide `x` by 4 and assign |
| Compound Modulus | `x %= 3;` | `x` modulo 3 and assign |

### 3.2.14 Practice Exercise

Rewrite the following code using compound assignment operators to make it shorter:

```
int total = 100;
total = total + 50;
total = total - 20;
total = total * 2;
total = total / 5;
total = total % 3;
```

**Answer:**

```
int total = 100;
total += 50;
total -= 20;
total *= 2;
total /= 5;
total %= 3;
```

Understanding assignment and compound assignment operators will help you write cleaner and more efficient C++ code. These operators are fundamental in many programming tasks, from simple calculations to complex loops and algorithms.

## 3.3 Increment and Decrement

In C++, **increment** and **decrement** operators provide a shorthand way to increase or decrease the value of a variable by one. These operators are commonly used in loops, counters, and many algorithms, making your code more concise and readable.

This section explains the two types of increment/decrement operators—**pre** and **post**—how they behave differently in expressions, and their typical use cases.

### 3.3.1 The Increment (++) and Decrement (--) Operators

- **Increment operator (++)**: increases the value of a variable by 1.
- **Decrement operator (--)**: decreases the value of a variable by 1.

These operators can be applied in two forms:

| Form | Syntax | Meaning |
|---|---|---|
| Pre-increment | `++x` | Increment `x` **before** using it |
| Post-increment | `x++` | Use `x` first, then increment it |
| Pre-decrement | `--x` | Decrement `x` **before** using it |
| Post-decrement | `x--` | Use `x` first, then decrement it |

### 3.3.2 How Pre- and Post-Increment Differ

Pre-increment (`x`)

- Increments the variable's value by 1.
- Then **returns the updated value**.
- Used when you want the incremented value immediately.

Post-increment (`x`)

- Returns the current value of the variable.
- Then increments the variable **afterwards**.
- Used when you need the original value before increment.

### 3.3.3 Examples to Illustrate

```cpp
int x = 5;

int a = ++x;  // Pre-increment: x becomes 6, then a is assigned 6
int b = x++;  // Post-increment: b is assigned 6, then x becomes 7

std::cout << "a = " << a << std::endl;  // Outputs: a = 6
std::cout << "b = " << b << std::endl;  // Outputs: b = 6
std::cout << "x = " << x << std::endl;  // Outputs: x = 7
```

**Explanation:**

- After `++x`, `x` is immediately increased to 6, so `a` is assigned 6.
- After `x++`, `b` gets the current value of `x` (which is 6), then `x` increases to 7.

### 3.3.4 Pre- vs Post-Decrement

Similarly, the decrement operators work in the same way but subtract 1 instead.

```
int y = 10;

int c = --y;  // Pre-decrement: y becomes 9, c = 9
int d = y--;  // Post-decrement: d = 9, then y becomes 8

std::cout << "c = " << c << std::endl;  // Outputs: c = 9
std::cout << "d = " << d << std::endl;  // Outputs: d = 9
std::cout << "y = " << y << std::endl;  // Outputs: y = 8
```

### 3.3.5 Common Use Cases

#### Using Increment/Decrement in Loops

These operators are often used in `for` or `while` loops to control iteration:

```
for (int i = 0; i < 5; ++i) {
    std::cout << i << " ";
}
// Output: 0 1 2 3 4
```

- Pre-increment (`++i`) is slightly more efficient in some cases but functionally equivalent to post-increment (`i++`) in loops.
- Use whichever you prefer, but consistency matters.

#### When Expression Result Matters

When the increment operation is part of a larger expression, the difference between pre- and post-increment affects the outcome.

Example:

```
int x = 3;
int y = x++ + 5;  // y = 3 + 5 = 8, x becomes 4 afterward

int a = 3;
int b = ++a + 5;  // a becomes 4, b = 4 + 5 = 9
```

### 3.3.6 Potential Pitfalls

- **Using post-increment in complex expressions can be confusing** or lead to bugs if you don't carefully track the order of evaluation.

Example:

```
int i = 1;
int j = i++ + i++;
```

- The result depends on the compiler and order of evaluation, which is undefined behavior in C++. Avoid writing such expressions.
- **Avoid multiple increments on the same variable in one expression**.

### 3.3.7 Summary

| Operator Type | Syntax | Description | Example | Result |
|---|---|---|---|---|
| Pre-increment | `++x` | Increment then return new value | `++x` when `x=5` → 6 | `x` becomes 6 immediately |
| Post-increment | `x++` | Return value then increment | `x++` when `x=5` → 5 | `x` becomes 6 after expression |
| Pre-decrement | `--x` | Decrement then return new value | `--x` when `x=5` → 4 | `x` becomes 4 immediately |
| Post-decrement | `x--` | Return value then decrement | `x--` when `x=5` → 5 | `x` becomes 4 after expression |

### 3.3.8 Best Practices

- Use **pre-increment/decrement (`++x`, `--x`)** when the updated value is needed immediately.
- Use **post-increment/decrement (`x++`, `x--`)** when the current value should be used before changing.
- Avoid complex expressions with multiple increments/decrements on the same variable to prevent undefined or confusing behavior.
- In loops, prefer pre-increment/decrement for slightly better performance and clarity, although both work fine.

Increment and decrement operators are simple but powerful tools that make counting and iterative operations concise. Understanding the subtle difference between pre- and post-forms is crucial for writing correct, bug-free code.

## 3.4 Operator Precedence and Associativity

When writing expressions in C++, you often combine multiple operators—arithmetic, relational, logical, and more—in a single statement. But how does the compiler decide **which operator to evaluate first**? This is where **operator precedence** and **associativity** come into play.

Understanding these rules is essential to predict how expressions will be evaluated and to avoid unexpected results.

### 3.4.1 What Is Operator Precedence?

**Operator precedence** defines the priority or order in which different operators are evaluated in an expression without parentheses. Operators with higher precedence are evaluated before those with lower precedence.

For example, multiplication (`*`) has higher precedence than addition (`+`), so in the expression:

```
int result = 2 + 3 * 4;
```

the multiplication happens first:

- `3 * 4` equals `12`
- Then `2 + 12` equals `14`

If you need to change this order, you can use parentheses:

```
int result = (2 + 3) * 4;  // Now addition happens first, result is 20
```

### 3.4.2 What Is Associativity?

When multiple operators of the **same precedence** appear together, **associativity** determines the order in which they are evaluated.

- **Left-to-right associativity** means evaluation starts from the left side and moves right.
- **Right-to-left associativity** means evaluation starts from the right side and moves left.

For example, most binary operators like addition and subtraction have **left-to-right** associativity:

```
int x = 10 - 5 - 2;  // Evaluated as (10 - 5) - 2 = 3
```

However, the assignment operator `=` has **right-to-left** associativity:

```
int a, b;
a = b = 5;           // Evaluated as a = (b = 5);
```

### 3.4.3 Operator Precedence Table (Selected Operators)

| Precedence Level | Operators | Associativity | Description | Example | Explanation |
|---|---|---|---|---|---|
| 1 (highest) | `++`, `--` (postfix) | Left to Right | Post-increment/decrement | `x++` | `x` used, then incremented |
| 2 | `++`, `--` (prefix), `+`, `-` (unary), `!` | Right to Left | Unary plus/minus, logical NOT | `++x`, `-y`, `!flag` | Increment before usage, negation |
| 3 | `*`, `/`, `%` | Left to Right | Multiplication, division, modulus | `a * b / c` | Evaluated left to right |
| 4 | `+`, `-` | Left to Right | Addition, subtraction | `a + b - c` | Evaluated left to right |
| 5 | `<`, `<=`, `>`, `>=` | Left to Right | Relational comparisons | `x < y && y > z` | Comparisons before logical AND |
| 6 | `==`, `!=` | Left to Right | Equality and inequality | a == b\|\| c != d‘ | Equality checks before logical OR |
| 7 | `&&` | Left to Right | Logical AND | `a && b && c` | Evaluated left to right |
| 8 | \|\| | Left to Right | Logical OR | a\|\| b\|\| c | Evaluated left to right |
| 9 (lowest) | `=` (assignment), `+=`, `-=`, `*=`, `/=`, `%=` | Right to Left | Assignment operators | `a = b = c` | Evaluated right to left |

*Note:* This table covers common operators. C++ has many others with their own precedence rules.

Examples of Precedence and Associativity in Action

### 3.4.4 Example 1: Multiplication before Addition

```
int result = 4 + 5 * 3;
// Multiplication has higher precedence, so 5*3=15 first
// Then 4 + 15 = 19
```

### 3.4.5 Example 2: Left-to-right Associativity with Subtraction

```
int value = 20 - 5 - 3;
// Evaluated as (20 - 5) - 3 = 12
```

### 3.4.6 Example 3: Right-to-left Associativity with Assignment

```
int a, b;
a = b = 10;
// Equivalent to a = (b = 10);
// b assigned 10, then a assigned b's value (10)
```

### 3.4.7 Example 4: Combining Logical Operators

```
bool result = true || false && false;
// `&&` has higher precedence than `||`
// So `false && false` evaluates to false
// Then `true || false` evaluates to true
```

### 3.4.8 Using Parentheses to Control Evaluation

Parentheses `()` are the most reliable way to explicitly define the order of evaluation and override precedence rules.

Example:

```
int value = (4 + 5) * 3;  // Forces addition first, then multiplication
// Result is 27, not 19
```

Using parentheses makes your code easier to read and less prone to errors, especially in complex expressions.

### 3.4.9 Tips for Writing Clear Expressions

- **Use parentheses liberally** to clarify complex expressions—even when not strictly required by precedence.
- **Avoid chaining too many operators** in a single expression to prevent confusion.
- **Test expressions carefully** if you're not sure about the evaluation order.

### 3.4.10 Summary

- **Operator precedence** determines which operator is evaluated first.
- **Associativity** determines the order of evaluation when operators have the same precedence.
- Most binary arithmetic and relational operators have **left-to-right** associativity.
- Assignment operators have **right-to-left** associativity.
- Use **parentheses** to control evaluation explicitly and improve code clarity.

Understanding precedence and associativity ensures your expressions behave as intended and your programs produce correct results.

# Chapter 4.

# Control Flow Statements

1. `if`, `else if`, and `else`
2. `switch` Statement
3. Loops: `for`, `while`, `do-while`
4. Control Flow Modifiers: `break`, `continue`

# 4 Control Flow Statements

## 4.1 `if`, `else if`, and `else`

In programming, making decisions is essential. **Conditional branching** allows your program to execute different blocks of code based on whether certain conditions are true or false. In C++, the primary tools for conditional branching are the `if`, `else if`, and `else` statements.

This section will explain how these statements work, their syntax, evaluation order, and practical use cases with examples. You'll also learn best practices for writing clear and readable conditional code.

### 4.1.1 The `if` Statement

The `if` statement tests a condition (an expression that evaluates to `true` or `false`). If the condition is true, the code inside the `if` block executes. If the condition is false, the program skips the block.

### 4.1.2 Syntax

```cpp
if (condition) {
    // code to run if condition is true
}
```

### 4.1.3 Example

```cpp
int age = 20;

if (age >= 18) {
    std::cout << "You are an adult." << std::endl;
}
```

Here, the message is printed only if `age` is 18 or older.

### 4.1.4 The `else` Statement

Sometimes, you want to execute alternative code if the `if` condition is false. The `else` statement allows this.

### 4.1.5 Syntax

```
if (condition) {
    // code if true
} else {
    // code if false
}
```

### 4.1.6 Example

```
int age = 16;

if (age >= 18) {
    std::cout << "You are an adult." << std::endl;
} else {
    std::cout << "You are a minor." << std::endl;
}
```

Because `age` is less than 18, the `else` block runs, printing “You are a minor.”

### 4.1.7 The `else if` Statement

When you have **multiple conditions to check**, you can chain them using `else if`. This structure tests conditions one by one until one evaluates to true. If none match, the optional `else` block runs.

### 4.1.8 Syntax

```
if (condition1) {
    // code if condition1 is true
} else if (condition2) {
    // code if condition2 is true
} else {
    // code if none of the above conditions are true
}
```

### 4.1.9 Example

```
int score = 75;
```

```cpp
if (score >= 90) {
    std::cout << "Grade: A" << std::endl;
} else if (score >= 80) {
    std::cout << "Grade: B" << std::endl;
} else if (score >= 70) {
    std::cout << "Grade: C" << std::endl;
} else {
    std::cout << "Grade: F" << std::endl;
}
```

Here, since `score` is 75, the program prints “Grade: C”.

### 4.1.10 Evaluation Order and Flow

- The conditions are evaluated **in order, from top to bottom**.
- Once a **true condition** is found, its block runs, and the rest of the chain is skipped.
- If none of the conditions are true, and there is an `else` block, it executes.
- If no `else` block exists and all conditions are false, nothing happens.

### 4.1.11 Nested `if` Statements

You can place an `if` statement **inside** another `if` or `else` block. This is useful when decisions depend on multiple layers of conditions.

### 4.1.12 Example

```cpp
int age = 20;
bool hasID = true;

if (age >= 18) {
    if (hasID) {
        std::cout << "Entry allowed." << std::endl;
    } else {
        std::cout << "ID required for entry." << std::endl;
    }
} else {
    std::cout << "Entry denied. Must be 18 or older." << std::endl;
}
```

This program checks both age and ID status before allowing entry.

### 4.1.13 Chained `if` vs. Nested `if`

- **Chained `if`** (using `else if`) is good for mutually exclusive conditions—only one block executes.
- **Nested `if`** is useful when you need to check multiple, dependent conditions.

### 4.1.14 Proper Use of Braces `{}`

Even if an `if` or `else` block contains only one statement, **always use braces**. This improves readability and prevents bugs when you later add more statements.

### 4.1.15 Without braces (can cause bugs):

```
if (age >= 18)
    std::cout << "Adult" << std::endl;
    std::cout << "Welcome!" << std::endl;  // Runs regardless of age
```

### 4.1.16 With braces (correct):

```
if (age >= 18) {
    std::cout << "Adult" << std::endl;
    std::cout << "Welcome!" << std::endl;  // Runs only if age >= 18
}
```

### 4.1.17 Example 1: Check Number Sign

```
int num = -5;

if (num > 0) {
    std::cout << "Positive number" << std::endl;
} else if (num < 0) {
    std::cout << "Negative number" << std::endl;
} else {
    std::cout << "Zero" << std::endl;
}
```

### 4.1.18 Example 2: Voting Eligibility

```cpp
int age;

std::cout << "Enter your age: ";
std::cin >> age;

if (age >= 18) {
    std::cout << "You can vote." << std::endl;
} else {
    std::cout << "You are too young to vote." << std::endl;
}
```

### 4.1.19 Summary

- Use `if` to execute code when a condition is true.
- Use `else` to execute code when the condition is false.
- Use `else if` to check multiple conditions sequentially.
- Conditions are evaluated **in order**, and only the first true condition's block runs.
- Use **braces `{}`** even for single statements to avoid bugs and improve readability.
- Nested `if` statements allow checking dependent conditions.

Mastering conditional branching with `if`, `else if`, and `else` allows your programs to make decisions and respond dynamically. Practice writing decision structures to become comfortable with these essential tools!

## 4.2 `switch` Statement

In C++, the `switch` statement offers a structured and efficient alternative to multiple `if-else if` conditions, especially when dealing with **discrete constant values**, such as integers or enumeration types. It's particularly useful when a variable needs to be compared against several distinct values, making code cleaner and more readable than a long chain of conditional statements.

### 4.2.1 Why Use `switch`?

Using many `if-else` blocks to compare a single variable against different values can quickly become unwieldy:

```cpp
if (value == 1) {
    // do something
} else if (value == 2) {
    // do something else
} else if (value == 3) {
    // another option
} else {
    // default action
}
```

This same logic can be handled more clearly with a `switch`:

```
switch (value) {
    case 1:
        // do something
        break;
    case 2:
        // do something else
        break;
    case 3:
        // another option
        break;
    default:
        // default action
}
```

### 4.2.2 Syntax of a `switch` Statement

```
switch (expression) {
    case constant1:
        // code block
        break;
    case constant2:
        // code block
        break;
    ...
    default:
        // default block
}
```

- **`expression`**: Must evaluate to an integral or enum type (e.g., `int`, `char`, `enum`).
- **`case` labels**: Must be constant expressions (e.g., `case 1:`, `case 'A':`).
- **`break`**: Exits the `switch` block. Without it, execution "falls through" to the next case.
- **`default`**: Optional block that runs if no `case` matches.

### 4.2.3 Example: Simple Menu with `switch`

```
int choice;
std::cout << "1. Start Game\n2. Load Game\n3. Quit\n";
std::cout << "Enter your choice: ";
std::cin >> choice;
```

```
switch (choice) {
    case 1:
        std::cout << "Starting new game...\n";
        break;
    case 2:
        std::cout << "Loading saved game...\n";
        break;
    case 3:
        std::cout << "Quitting game.\n";
        break;
    default:
        std::cout << "Invalid choice.\n";
}
```

### 4.2.4 Output Example:

```
1. Start Game
2. Load Game
3. Quit
Enter your choice: 2
Loading saved game...
```

### 4.2.5 The Role of `break`

The `break` statement prevents the program from executing the code in subsequent cases. Without it, control will "fall through" to the next `case`.

### 4.2.6 Example Without `break`

```
int level = 2;

switch (level) {
    case 1:
        std::cout << "Level 1\n";
    case 2:
        std::cout << "Level 2\n";
    case 3:
        std::cout << "Level 3\n";
}
```

**Output:**

```
Level 2
Level 3
```

In this example, once `case 2` matches, it continues executing through `case 3` because there are no `break` statements. This is usually **not** desired, unless fall-through behavior is intentional.

### 4.2.7 The `default` Case

The `default` label handles any values not explicitly matched by the `case` labels. It works like the `else` in an `if-else` chain.

### 4.2.8 Example:

```
char grade = 'B';

switch (grade) {
    case 'A':
        std::cout << "Excellent!\n";
        break;
    case 'B':
        std::cout << "Good job!\n";
        break;
    case 'C':
        std::cout << "You passed.\n";
        break;
    default:
        std::cout << "Invalid grade.\n";
}
```

If `grade` is `'B'`, the output will be “Good job!”. If `grade` is `'F'`, the `default` case will trigger.

### 4.2.9 Using `enum` with `switch`

`switch` pairs well with `enum` types, improving readability and maintainability.

### 4.2.10 Example:

```
enum Color { Red, Green, Blue };
Color selectedColor = Green;

switch (selectedColor) {
    case Red:
        std::cout << "You selected Red.\n";
        break;
    case Green:
        std::cout << "You selected Green.\n";
        break;
    case Blue:
        std::cout << "You selected Blue.\n";
        break;
    default:
        std::cout << "Unknown color.\n";
}
```

Enums make it easier to work with symbolic names rather than numeric codes.

### 4.2.11 When to Use `switch`

Use `switch` when:

- You're comparing the **same variable** against **multiple constant values**.
- Your values are **discrete**, not ranges.
- You want **cleaner, more efficient branching** than many `if-else if` blocks.

Avoid `switch` when:

- You're comparing ranges (e.g., `score > 90`)
- You need complex boolean expressions (e.g., `if (x > 5 && y < 10)`)

### 4.2.12 Summary

- `switch` is a multi-way branching statement for comparing a single value to multiple constants.
- Each `case` represents a value to match.
- `break` ends the case block to prevent fall-through.
- `default` handles unmatched values.
- Works best with `int`, `char`, and `enum` types.
- Improves clarity and structure when checking against many constant values.

Understanding how and when to use `switch` helps you write more organized and readable decision logic, especially in menus, state machines, and user input handling.

## 4.3 Loops: `for`, `while`, `do-while`

In programming, repetition is often necessary. Instead of copying and pasting the same code, C++ provides **loops** to execute a block of code multiple times under certain conditions. This concept is known as **iteration**.

C++ offers three primary loop types:

- `for` loop
- `while` loop
- `do-while` loop

Each has its own syntax and use cases, but they all help avoid redundancy and make programs more flexible and efficient.

### 4.3.1 The `for` Loop

Syntax

```
for (initialization; condition; update) {
    // Code to repeat
}
```

### 4.3.2 Components

- **Initialization**: Set up a loop control variable.
- **Condition**: Checked before each iteration. If false, the loop ends.
- **Update**: Changes the loop variable, usually incrementing or decrementing.

### 4.3.3 Example: Counting from 1 to 5

```
for (int i = 1; i <= 5; ++i) {
    std::cout << i << " ";
}
// Output: 1 2 3 4 5
```

### 4.3.4 When to Use

- When the number of iterations is known in advance.

- Useful for indexing arrays, performing repeated calculations, and counting.

### 4.3.5 The `while` Loop

Syntax

```
while (condition) {
    // Code to repeat
}
```

### 4.3.6 Behavior

- Checks the condition **before** each iteration.
- If the condition is false at the start, the loop doesn't execute at all.

### 4.3.7 Example: Sentinel-Controlled Loop

```
int number;
std::cout << "Enter a number (-1 to quit): ";
std::cin >> number;

while (number != -1) {
    std::cout << "You entered: " << number << std::endl;
    std::cout << "Enter a number (-1 to quit): ";
    std::cin >> number;
}
```

This loop continues until the user enters `-1`, a common sentinel value.

### 4.3.8 When to Use

- When the number of iterations is **not known** ahead of time.
- Often used for user input, reading from files, or waiting for a condition to change.

### 4.3.9 The `do-while` Loop

Syntax

```
do {
    // Code to repeat
} while (condition);
```

### 4.3.10 Behavior

- Executes the code **at least once**, regardless of the condition.
- Then checks the condition **after** the first iteration.

### 4.3.11 Example: Menu-Driven Program

```
int choice;

do {
    std::cout << "1. Say Hello\n";
    std::cout << "2. Say Goodbye\n";
    std::cout << "3. Exit\n";
    std::cout << "Enter your choice: ";
    std::cin >> choice;

    switch (choice) {
        case 1:
            std::cout << "Hello!\n";
            break;
        case 2:
            std::cout << "Goodbye!\n";
            break;
        case 3:
            std::cout << "Exiting...\n";
            break;
        default:
            std::cout << "Invalid option.\n";
    }
} while (choice != 3);
```

### 4.3.12 When to Use

- When the loop must execute **at least once**, such as displaying a menu or prompting user input.

### 4.3.13 Comparing Loop Types

| Feature | `for` Loop | `while` Loop | `do-while` Loop |
|---|---|---|---|
| Condition checked | Before loop | Before loop | After loop |
| Executes at least once | No | No | Yes |
| Ideal for | Counting, fixed loops | Indefinite loops | Menu/input-driven loops |
| Initialization & update | Inline in header | External to loop | External to loop |

### 4.3.14 Avoiding Infinite Loops

An **infinite loop** runs endlessly if the condition never becomes false. This is a common beginner mistake.

### 4.3.15 Example of an Infinite Loop

```
int i = 0;

while (i < 5) {
    std::cout << i << std::endl;
    // Missing i++
}
```

Here, `i` is never incremented, so the condition is always true. The fix:

```
int i = 0;

while (i < 5) {
    std::cout << i << std::endl;
    ++i;
}
```

**Best Practices:**

- Always ensure the loop condition can become false.
- Use a loop counter or sentinel value appropriately.
- Test loops carefully to avoid unexpected behavior.

### 4.3.16 Nested Loops

You can place one loop inside another for more complex iteration.

### 4.3.17 Example: Multiplication Table

```cpp
for (int i = 1; i <= 3; ++i) {
    for (int j = 1; j <= 3; ++j) {
        std::cout << i * j << "\t";
    }
    std::cout << std::endl;
}
```

**Output:**

```
1    2    3
2    4    6
3    6    9
```

Nested loops are useful for matrices, grids, and multi-level operations.

### 4.3.18 Summary

- **`for` loops** are best for known, fixed repetitions.
- **`while` loops** are ideal when repetition depends on a condition.
- **`do-while` loops** guarantee at least one execution before checking the condition.
- Always make sure your loop can terminate to avoid infinite loops.
- Use meaningful loop conditions and keep the logic inside the loop simple and clear.

Mastering loops allows you to build more dynamic and responsive programs. Practice with counting, input-driven tasks, and nested loops to become confident with each loop type.

## 4.4 Control Flow Modifiers: `break`, `continue`

In C++, control flow modifiers like `break` and `continue` give you more flexibility within loops and `switch` statements. They allow you to influence how and when loop iterations proceed or terminate, making your code more efficient and responsive to runtime conditions.

### 4.4.1 The `break` Statement

Purpose

The `break` statement immediately **terminates the nearest enclosing loop** (`for`, `while`, or `do-while`) or a `switch` block. Execution resumes with the first statement after the loop or switch.

### 4.4.2 Syntax

```
break;
```

### 4.4.3 Example 1: Using `break` in a `for` Loop

```
for (int i = 1; i <= 10; ++i) {
    if (i == 5) {
        break;  // Exit loop when i reaches 5
    }
    std::cout << i << " ";
}
// Output: 1 2 3 4
```

In this example, the loop exits when `i` is 5, skipping the remaining iterations.

### 4.4.4 Example 2: Using `break` in a `while` Loop

```
int n;
while (true) {
    std::cout << "Enter a number (0 to stop): ";
    std::cin >> n;

    if (n == 0) {
        break;  // Exit the loop if user enters 0
    }

    std::cout << "You entered: " << n << std::endl;
}
```

This loop runs indefinitely (`while (true)`), but the `break` provides a condition for exiting based on user input.

### 4.4.5 Example 3: `break` in a `switch` Statement

```
int choice = 2;

switch (choice) {
    case 1:
        std::cout << "Option 1\n";
        break;
    case 2:
        std::cout << "Option 2\n";
        break;
    case 3:
        std::cout << "Option 3\n";
        break;
    default:
        std::cout << "Invalid option\n";
}
```

Without `break`, control would "fall through" to the next case. `break` ensures only the matched case executes.

### 4.4.6 The `continue` Statement

The `continue` statement **skips the remaining code in the current loop iteration** and proceeds to the next iteration.

### 4.4.7 Syntax

```
continue;
```

### 4.4.8 Example 1: `continue` in a `for` Loop

```
for (int i = 1; i <= 5; ++i) {
    if (i == 3) {
        continue;  // Skip this iteration
    }
    std::cout << i << " ";
}
// Output: 1 2 4 5
```

When `i` is 3, `continue` skips the `std::cout` line and jumps to the next iteration of the loop.

### 4.4.9 Example 2: `continue` in a `while` Loop

```
int i = 0;

while (i < 5) {
    ++i;

    if (i == 2) {
        continue;  // Skip printing when i is 2
    }

    std::cout << i << " ";
}
// Output: 1 3 4 5
```

Note: The increment `++i` is placed **before** the `continue`, otherwise `i` would remain unchanged, possibly causing an infinite loop.

### 4.4.10 When to Use `break`

- Exiting a loop early when a goal is achieved (e.g., found a matching value).
- Stopping user input or processing on a sentinel value.
- Exiting from a `switch` case to avoid fall-through.

### 4.4.11 When to Use `continue`

- Skipping unnecessary iterations based on a condition.
- Filtering data in a loop without deeply nested conditionals.

### 4.4.12 Common Pitfalls

- **Misplaced `continue` or `break`** can cause logic errors or infinite loops.
- **Nested loops**: `break` or `continue` only affects the innermost loop, not outer ones.
- Overusing `break` and `continue` can make code harder to read. Prefer using clear and structured loop conditions when possible.

### 4.4.13 Nested Loops and `break`

In nested loops, `break` exits **only the inner loop** where it appears.

```
for (int i = 1; i <= 3; ++i) {
    for (int j = 1; j <= 3; ++j) {
        if (j == 2) {
            break;  // Only exits inner loop
        }
        std::cout << "(" << i << "," << j << ") ";
    }
}
// Output: (1,1) (2,1) (3,1)
```

### 4.4.14 Summary

- `break` exits the current loop or `switch` block immediately.
- `continue` skips the current iteration and moves to the next one.
- Use `break` to stop loops early based on a condition.
- Use `continue` to avoid unnecessary processing in some iterations.
- Always use these modifiers carefully to maintain readable and predictable code.

With proper use, `break` and `continue` enhance the flexibility and efficiency of your control flow logic in C++ programs.

# Chapter 5.

# Functions

1. Defining and Calling Functions
2. Function Parameters and Return Types
3. Function Overloading
4. Inline Functions and Default Arguments

# 5 Functions

## 5.1 Defining and Calling Functions

In programming, one of the key principles for writing efficient, organized, and reusable code is **modularization**. In C++, this is achieved through the use of **functions**. Functions allow you to break a large program into smaller, manageable pieces. Each function performs a specific task, making your code cleaner, more readable, and easier to maintain.

### 5.1.1 What Is a Function?

A **function** is a self-contained block of code that performs a specific task. Once defined, a function can be **called** or **invoked** from other parts of the program — including from within other functions (even itself, in the case of recursion).

Functions are especially useful for:

- Reusing code without rewriting it
- Organizing logic into meaningful chunks
- Making programs easier to read, test, and debug

### 5.1.2 Basic Structure of a Function

To use a function, you need to define it first and then call it when needed. A basic function has the following syntax:

```
return_type function_name() {
    // function body
}
```

- **`return_type`**: The type of value the function returns (e.g., `int`, `double`, `void`).
- **`function_name`**: The identifier you use to refer to the function.
- **Function body**: The block of statements enclosed in `{}` that define what the function does.

### 5.1.3 Example 1: A Simple Function with No Parameters and No Return Value

Full runnable code:

```cpp
#include <iostream>

// Function definition
void greet() {
    std::cout << "Hello, welcome to C++ programming!" << std::endl;
}

int main() {
    // Function call
    greet();
    return 0;
}
```

**Explanation:**

- `greet()` is a function that prints a greeting message.
- The return type `void` means it does **not** return any value.
- The function is called from `main()` using the statement `greet();`.

### 5.1.4 Example 2: A Function That Returns a Value

Full runnable code:

```cpp
#include <iostream>

int getNumber() {
    return 42;
}

int main() {
    int result = getNumber();
    std::cout << "The number is: " << result << std::endl;
    return 0;
}
```

**Explanation:**

- `getNumber()` returns an integer (`int`) with the value 42.
- The return value is stored in the variable `result` and printed.
- Functions that return values must use the `return` keyword followed by a value of the correct type.

### 5.1.5 Function Declaration and Definition

In larger programs, functions are often **declared** (also called **prototyped**) before they are used, then **defined** later. This helps the compiler know the function's signature before its actual implementation.

### 5.1.6 Example with Declaration

Full runnable code:

```cpp
#include <iostream>

// Function declaration
int square(int x);

int main() {
    int num = 5;
    std::cout << "Square of " << num << " is " << square(num) << std::endl;
    return 0;
}

// Function definition
int square(int x) {
    return x * x;
}
```

- The declaration `int square(int x);` tells the compiler what the function looks like.
- The definition provides the actual code.
- This pattern is useful when organizing functions across multiple files.

### 5.1.7 Calling Functions

To **call** a function, you use its name followed by parentheses:

```cpp
function_name();
```

- If the function takes arguments, pass them inside the parentheses: `function_name(arg1, arg2);`
- The program will jump to the function, execute its code, and return back to the point where it was called.

### 5.1.8 Benefits of Using Functions

1. **Code Reusability**: You can reuse a function in multiple places without rewriting the same logic.
2. **Readability**: Functions make programs easier to read and understand by dividing them into logical blocks.
3. **Maintainability**: Bugs and changes are easier to isolate and fix in small, modular functions.
4. **Debugging**: Testing functions individually makes it easier to find and fix issues.
5. **Collaboration**: In team environments, multiple developers can work on different

functions concurrently.

### 5.1.9 Example: Multiple Functions

Full runnable code:

```
#include <iostream>

void printWelcome() {
    std::cout << "Welcome!" << std::endl;
}

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

int main() {
    printWelcome();
    int result = add(3, 4);
    std::cout << "Sum is: " << result << std::endl;
    return 0;
}
```

- This program uses two functions: `printWelcome()` and `add(int, int)`.
- `add()` demonstrates a function that accepts parameters and returns a result.

### 5.1.10 Summary

- A function is a named block of code that performs a task.
- Functions help you break large programs into smaller, more manageable pieces.
- Define a function using a return type, name, and a code block.
- Use the `return` statement to send back a value (if not `void`).
- Call a function by using its name followed by parentheses.
- Function declarations can appear before the `main()` function when needed.

Understanding how to define and call functions is a fundamental skill in C++ programming. As you build more complex programs, functions will help you organize, reuse, and manage your code effectively.

## 5.2 Function Parameters and Return Types

Functions in C++ are powerful tools for breaking a program into manageable parts. To make functions flexible and reusable, you often need to pass data to them and receive results in return. This section explains how **parameters** and **return types** work in C++, covering different ways to pass arguments, return values, and the concept of scope and lifetime of variables inside functions.

### 5.2.1 What Are Function Parameters?

**Parameters** (also called **arguments**) are variables listed in a function's definition that allow you to pass data into a function when calling it.

### 5.2.2 Example:

```cpp
void greet(std::string name) {
    std::cout << "Hello, " << name << "!" << std::endl;
}

int main() {
    greet("Alice");
    return 0;
}
```

Here, `name` is a parameter. When the function is called with `"Alice"`, it prints a personalized greeting.

### 5.2.3 Function Return Types

A **return type** indicates the kind of value a function returns. If a function doesn't return any value, its return type is `void`.

### 5.2.4 Example: Returning an `int`

```cpp
int square(int number) {
    return number * number;
}

int main() {
    int result = square(4);
    std::cout << "4 squared is " << result << std::endl;
    return 0;
}
```

- `square` takes an `int` and returns its square.
- The result is stored in the variable `result`.

### 5.2.5 Example: `void` Function

```
void printLine() {
    std::cout << "-------------------" << std::endl;
}
```

This function performs an action but doesn't return anything.

### 5.2.6 Passing Arguments: By Value vs. By Reference

C++ allows you to pass arguments in two main ways:

### 5.2.7 Pass by Value

This is the default behavior in C++. The function receives a **copy** of the argument. Changes made to the parameter inside the function do **not** affect the original variable.

```
void changeValue(int x) {
    x = 100;
}

int main() {
    int num = 10;
    changeValue(num);
    std::cout << num << std::endl; // Output: 10
    return 0;
}
```

### 5.2.8 Pass by Reference

To modify the original argument, use **pass by reference** by adding an ampersand `&` to the parameter.

```
void changeValue(int& x) {
    x = 100;
}

int main() {
    int num = 10;
    changeValue(num);
    std::cout << num << std::endl; // Output: 100
    return 0;
}
```

Now `x` refers directly to `num`, so changes to `x` affect `num`.

### 5.2.9 When to Use Each:

- Use **pass by value** when you don't want to modify the original data.
- Use **pass by reference** when:
  - You need to modify the original variable.
  - You want to avoid copying large data structures (e.g., vectors, strings) for efficiency.

### 5.2.10 Returning Different Data Types

Functions in C++ can return various data types — `int`, `double`, `char`, `bool`, even complex types like `std::string` or user-defined types.

### 5.2.11 Example: Returning a `double`

```
double average(double a, double b) {
    return (a + b) / 2.0;
}
```

### 5.2.12 Example: Returning a `bool`

```
bool isEven(int num) {
    return num % 2 == 0;
}
```

### 5.2.13 Example: Returning a `std::string`

```cpp
std::string makeGreeting(std::string name) {
    return "Hello, " + name + "!";
}
```

### 5.2.14 Parameter Scope and Lifetime

Each parameter and local variable inside a function has **local scope** — meaning it only exists during the execution of the function.

### 5.2.15 Example:

```cpp
void displayMessage() {
    int temp = 42; // `temp` is local to this function
    std::cout << temp << std::endl;
}
```

Attempting to access `temp` outside `displayMessage()` will cause a compilation error.

- Once the function finishes, all its local variables are **destroyed**.
- This isolation helps prevent conflicts and bugs in larger programs.

### 5.2.16 Multiple Parameters

Functions can accept more than one parameter:

```cpp
int sum(int a, int b, int c) {
    return a + b + c;
}
```

When calling the function, the order and type of arguments must match:

```cpp
int total = sum(10, 20, 30); // total = 60
```

### 5.2.17 Summary

- Functions can take parameters to operate on specific data and return values to communicate results.
- Parameters can be passed **by value** (copy) or **by reference** (modify original).

- Functions can return any data type or `void` if no return is needed.
- Parameters and local variables have **local scope** and **limited lifetime**.
- Using parameters and return types effectively makes functions more powerful, reusable, and maintainable.

Understanding how parameters and return values work is essential to writing modular and efficient C++ code. Practice by writing your own functions that accept input, perform calculations, and return results — you'll find that even complex programs become much easier to manage.

## 5.3 Function Overloading

In C++, **function overloading** allows you to define multiple functions with the same name but different **parameter lists**. This powerful feature enables you to create functions that perform similar tasks but on different types or numbers of arguments — improving code readability and reducing the need for uniquely named variants.

### 5.3.1 What Is Function Overloading?

Function overloading means you can declare several functions in the same scope with the same name, as long as their **parameter types**, **number of parameters**, or **parameter order** are different.

The compiler uses the information from the **function call** to determine which version of the function to invoke. This process is known as **overload resolution**.

### 5.3.2 Why Use Function Overloading?

- It improves code clarity and consistency.
- It lets you use the same function name for logically similar operations.
- It reduces the need to invent many different names for similar functionality.

### 5.3.3 Basic Syntax of Overloading

Here's a simple example:

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

```
void print(int i) {
    std::cout << "Integer: " << i << std::endl;
}

void print(double d) {
    std::cout << "Double: " << d << std::endl;
}

void print(std::string s) {
    std::cout << "String: " << s << std::endl;
}

int main() {
    print(10);            // Calls print(int)
    print(3.14);          // Calls print(double)
    print("Hello");       // Calls print(string)
    return 0;
}
```

### 5.3.4 Output:

```
Integer: 10
Double: 3.14
String: Hello
```

Each version of `print` handles a different data type, but the function name remains the same. The compiler chooses the appropriate version based on the type of the argument.

### 5.3.5 Rules for Overloading

1. **Functions must differ by parameter list**, not just return type.
2. You can overload based on:
   - Number of parameters
   - Type of parameters
   - Order of parameters (if types differ)

### 5.3.6 Invalid Overload Example (Only Return Type Differs):

```
int compute();
double compute(); // Error: only differs in return type
```

This causes a compilation error because the compiler can't distinguish which function to call based solely on the return type.

### 5.3.7 Example: Overloading with Different Numbers of Parameters

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

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

int main() {
    std::cout << add(2, 3) << std::endl;       // 5
    std::cout << add(2, 3, 4) << std::endl;    // 9
    return 0;
}
```

Here, both `add` functions have the same name but differ in the number of parameters.

### 5.3.8 Example: Overloading with Different Parameter Types

```cpp
double multiply(double a, double b) {
    return a * b;
}

int multiply(int a, int b) {
    return a * b;
}
```

In this example, both functions compute a product but for different data types. The function used will match the type of arguments passed in the function call.

### 5.3.9 Compiler's Role in Overload Resolution

When you call an overloaded function, the compiler looks at:

- The number of arguments
- The types of each argument
- The best match among available overloaded functions

If no suitable match is found, or if there's ambiguity, the compiler will issue an error.

### 5.3.10 Example: Ambiguous Overload

```
void display(int x);
void display(float x);

display(3.0);  // Ambiguous: 3.0 is a double
```

In this case, the compiler doesn't know whether to convert `3.0` to `int` or `float`, resulting in an ambiguous call. To fix it, either:

- Provide a `display(double x)` version, or
- Cast the argument explicitly: `display((float)3.0);`

### 5.3.11 Overloading and Default Arguments

Be cautious when combining overloading and **default arguments**. It can lead to ambiguity if not handled carefully:

```
void greet(std::string name = "Guest");
void greet(); // Error: Ambiguous with greet("Guest")
```

Avoid defining both a no-argument version and a default-argument version unless they differ in other meaningful ways.

### 5.3.12 Summary

- **Function overloading** allows multiple functions with the same name but different parameter types or counts.
- The compiler uses **overload resolution** to choose the correct function.
- You cannot overload functions by return type alone.
- Overloading improves code readability, especially for logically similar operations.
- Care is needed when using overloading with default arguments or similar types to avoid ambiguity.

Mastering function overloading helps you write clean, consistent, and expressive C++ programs — a skill that becomes increasingly valuable as your programs grow in size and complexity.

## 5.4 Inline Functions and Default Arguments

In C++, two useful function features — **inline functions** and **default arguments** — can help you write more efficient and flexible code. While both serve different purposes, they work together to make function definitions and calls cleaner, faster, and easier to manage.

### 5.4.1 Inline Functions

What Is an Inline Function?

An **inline function** is a function in which the compiler attempts to insert the function's body directly into the code at the point of the function call, rather than performing a typical function call (which involves jumping to the function code and returning back). This can eliminate the **overhead** of a function call, making the program potentially faster, especially for small, frequently called functions.

### 5.4.2 Syntax

To suggest that a function be inlined, use the `inline` keyword before the function definition:

```
inline int square(int x) {
    return x * x;
}
```

You can also define inline functions inside class definitions — these are implicitly treated as inline.

### 5.4.3 Example: Using an Inline Function

Full runnable code:

```
#include <iostream>

inline int cube(int n) {
    return n * n * n;
}

int main() {
    std::cout << "Cube of 3 is " << cube(3) << std::endl;
    return 0;
}
```

If the compiler inlines the function, the function call `cube(3)` is replaced with `3 * 3 * 3` directly in the `main()` function.

### 5.4.4 When to Use Inline Functions

**Use inline functions when:**

- The function is small and simple (like a single-line return).
- The function is called very frequently.

**Avoid using inline when:**

- The function is large or complex (inlining may increase code size — known as code bloat).
- It uses recursion or contains loops and large logic.

  Note: The `inline` keyword is only a *suggestion* to the compiler. Modern compilers may ignore it and decide on inlining based on optimization rules.

### 5.4.5 Default Arguments

A **default argument** is a value automatically assigned to a function parameter when no argument is provided by the caller. This makes function calls shorter and more flexible, especially when most parameters commonly take the same value.

### 5.4.6 Syntax

Default arguments are specified in the function declaration or definition by assigning a value in the parameter list:

```cpp
void greet(std::string name = "Guest") {
    std::cout << "Hello, " << name << "!" << std::endl;
}
```

### 5.4.7 Example: Using Default Arguments

Full runnable code:

```cpp
#include <iostream>

void greet(std::string name = "Guest") {
    std::cout << "Hello, " << name << "!" << std::endl;
}

int main() {
    greet();             // Output: Hello, Guest!
    greet("Alice");      // Output: Hello, Alice!
    return 0;
}
```

The first call uses the default argument `"Guest"`, while the second provides a specific value.

### 5.4.8 Rules and Best Practices

1. **Default arguments must appear from right to left.** Once you specify a default value, all parameters to its right must also have default values.

   YES Valid:

   ```
   void log(std::string msg, int level = 1);
   ```

   NO Invalid:

   ```
   void log(int level = 1, std::string msg); // Error
   ```

2. **Don't repeat default values** in both declaration and definition if they are separated:

   **Declaration (with default):**

   ```
   void display(std::string text = "Default");
   ```

   **Definition (without repeating default):**

   ```
   void display(std::string text) {
       std::cout << text << std::endl;
   }
   ```

3. Default arguments can make code more readable but avoid overusing them when it introduces ambiguity or confusion.

### 5.4.9 Combining Inline and Default Arguments

You can use both features together:

```
inline int multiply(int a, int b = 2) {
    return a * b;
}

int main() {
    std::cout << multiply(5) << std::endl;    // Uses default b = 2 → 10
    std::cout << multiply(5, 3) << std::endl; // b = 3 → 15
```

```
    return 0;
}
```

This function is short and efficient, and the default argument allows flexible calls.

### 5.4.10 Summary

- **Inline functions** are suggested to the compiler for performance optimization by replacing function calls with actual code.
- Use the `inline` keyword for small, frequently used functions.
- **Default arguments** simplify function calls by allowing parameters to have default values.
- Specify default values from right to left in the parameter list.
- Avoid duplicating default values in separate declarations and definitions.
- Together, inline functions and default arguments help you write cleaner, faster, and more flexible code.

Mastering these features will make your functions more efficient and easier to use — key traits in writing professional-grade C++ code.

# Chapter 6.

# Arrays and Strings

1. Single and Multi-Dimensional Arrays
2. C-style Strings vs. `std::string`
3. Common String Operations
4. Arrays of Strings and String Manipulation Examples

# 6 Arrays and Strings

## 6.1 Single and Multi-Dimensional Arrays

In C++, **arrays** are used to store multiple values of the same data type in a single, fixed-size collection. They provide a simple and efficient way to group related data, such as a list of numbers or characters. Arrays are a foundational concept in programming and are essential for handling structured data, looping operations, and basic algorithm implementation.

### 6.1.1 What Is an Array?

An **array** is a contiguous block of memory that holds a sequence of elements of the same type. Each element in the array is accessed using an **index**, starting from `0`.

### 6.1.2 Array Declaration

To declare an array, you specify the type of elements, the array name, and the number of elements it should hold:

```cpp
int numbers[5]; // An array of 5 integers
```

This creates an array named `numbers` that can store 5 integers, indexed from `0` to `4`.

### 6.1.3 Initializing Arrays

Arrays can be initialized at the time of declaration:

```cpp
int values[3] = {10, 20, 30};
```

You can also let the compiler determine the size based on the initializer list:

```cpp
int values[] = {10, 20, 30}; // Compiler infers size as 3
```

If you provide fewer initializers than the declared size, the remaining elements are set to zero:

```cpp
int scores[5] = {90, 85}; // scores[2] to scores[4] are 0
```

### 6.1.4 Accessing Elements

Array elements are accessed using their index inside square brackets:

```
std::cout << values[0]; // Outputs the first element (10)
values[2] = 50;         // Changes the third element to 50
```

**Important:** C++ does **not** perform automatic bounds checking. Accessing elements outside the valid range causes **undefined behavior**.

```
int a[3] = {1, 2, 3};
std::cout << a[5]; // Error: out-of-bounds access
```

To avoid such bugs, always ensure your index is within the valid range (0 to size−1).

### 6.1.5 Iterating Through Arrays

You can use loops to access or manipulate all elements of an array:

```
int data[] = {2, 4, 6, 8, 10};
int size = sizeof(data) / sizeof(data[0]);

for (int i = 0; i < size; i++) {
    std::cout << "Element " << i << ": " << data[i] << std::endl;
}
```

`sizeof(data) / sizeof(data[0])` computes the number of elements in the array.

### 6.1.6 Multi-Dimensional Arrays

C++ supports **multi-dimensional arrays**, commonly used to represent tables or matrices.

### 6.1.7 Declaring a 2D Array

```
int matrix[2][3]; // 2 rows, 3 columns
```

You can also initialize it directly:

```
int matrix[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

### 6.1.8 Accessing Elements in a 2D Array

Use two indices: one for the row and one for the column.

```
std::cout << matrix[0][2]; // Outputs 3
matrix[1][1] = 10;         // Changes the value at row 1, column 1
```

### 6.1.9 Iterating Through a 2D Array

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        std::cout << matrix[i][j] << " ";
    }
    std::cout << std::endl;
}
```

This prints the 2D array row by row.

### 6.1.10 Higher-Dimensional Arrays

Although less common, arrays can have more than two dimensions. For example, a 3D array can be declared like this:

```
int cube[2][3][4]; // 2 layers, 3 rows, 4 columns
```

Manipulating higher-dimensional arrays follows the same principles, but with more indices.

### 6.1.11 Common Pitfalls

#### Out-of-Bounds Access

C++ does not automatically detect if you're trying to access beyond an array's size:

```
int arr[3] = {1, 2, 3};
arr[5] = 10; // May compile, but behavior is undefined
```

**Tip:** Always use loop counters carefully and know the array size.

#### Uninitialized Elements

If you declare an array without initializing it, its elements may contain **garbage values**:

```
int data[5]; // Elements have unpredictable values
```

Use initialization or explicitly set values to avoid surprises.

### 6.1.12 Fixed Size

Traditional C++ arrays have a **fixed size**. If you need dynamic resizing or safer bounds checking, consider using `std::vector` or `std::array` (from the C++ Standard Library).

### 6.1.13 Summary

- Arrays are fixed-size collections of elements of the same type.
- They are indexed from `0` to `size - 1`.
- You can initialize arrays during declaration or later through assignment.
- Multi-dimensional arrays represent tabular or matrix data using multiple indices.
- Always stay within bounds and avoid accessing uninitialized elements.
- Arrays are simple but powerful tools in C++ for handling collections of data.

Understanding arrays is essential for managing structured data and implementing many algorithms effectively. In the next sections, you'll learn how to work with character arrays (C-style strings) and the more flexible `std::string` class.

## 6.2 C-style Strings vs. `std::string`

In C++, strings can be represented in two primary ways: **C-style strings**, which are arrays of characters terminated by a null character (`'\0'`), and the more modern and convenient **`std::string`** class, provided by the C++ Standard Library.

Understanding the differences between these two approaches is crucial. While both are used to store and manipulate textual data, they differ significantly in terms of memory management, functionality, and safety.

### 6.2.1 C-style Strings

A **C-style string** is a sequence of characters stored in a **character array**, ending with a special null character `'\0'`. This null terminator signals the end of the string.

### 6.2.2 Declaring a C-style String

```cpp
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Or, more commonly:

```cpp
char greeting[] = "Hello"; // Automatically adds '\0'
```

### 6.2.3 Accessing and Printing

Full runnable code:

```cpp
#include <iostream>

int main() {
    char name[] = "Alice";
    std::cout << name << std::endl;
    return 0;
}
```

### 6.2.4 Common C-style String Functions

C-style strings rely on functions from the C standard library (`<cstring>`):

- `strlen(str)` – returns the length (excluding `'\0'`)
- `strcpy(dest, src)` – copies `src` into `dest`
- `strcmp(a, b)` – compares two strings
- `strcat(dest, src)` – appends `src` to `dest`

### 6.2.5 Example:

Full runnable code:

```cpp
#include <iostream>
#include <cstring>

int main() {
    char str1[20] = "Hello";
    char str2[] = "World";

    strcat(str1, str2); // str1 now holds "HelloWorld"
    std::cout << str1 << std::endl;

    return 0;
}
```

### 6.2.6 Pitfalls of C-style Strings

- **Manual memory management**: You must ensure enough space is allocated.
- **Null terminator required**: Forgetting `'\0'` leads to undefined behavior.
- **No bounds checking**: Overruns may corrupt memory.
- **Harder to use**: Function calls like `strcpy` and `strcat` are error-prone.

### 6.2.7 `std::string` The Safer Alternative

The C++ Standard Library offers `std::string`, a powerful and convenient class for handling strings. Unlike C-style strings, `std::string` manages memory automatically, grows dynamically, and provides intuitive operations.

To use `std::string`, include the `<string>` header and use the `std` namespace:

Full runnable code:

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

int main() {
    std::string name = "Alice";
    std::cout << name << std::endl;
    return 0;
}
```

### 6.2.8 Advantages of `std::string`

- Automatic memory management
- Built-in string manipulation methods
- Bounds-safe access via `.at()`
- Operator overloading (`+`, `==`, etc.)
- Easily convertible to/from C-style strings if needed

### 6.2.9 Common `std::string` Operations

```cpp
std::string s1 = "Hello";
std::string s2 = "World";

// Concatenation
std::string message = s1 + " " + s2; // "Hello World"
```

```
// Comparison
if (s1 == "Hello") { ... }

// Length
std::cout << s1.length(); // 5

// Substring
std::string part = s2.substr(1, 3); // "orl"

// Access character
char ch = s1[0];    // 'H'
char safe = s1.at(1); // 'e' (throws exception if out of range)
```

### 6.2.10 Example: Using `std::string`

Full runnable code:

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

int main() {
    std::string name;
    std::cout << "Enter your name: ";
    std::cin >> name;
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
}
```

Unlike C-style strings, `std::string` handles memory and safety internally, allowing you to focus on logic instead of low-level concerns.

### 6.2.11 Converting Between C-style Strings and `std::string`

From `std::string` to C-style

```
std::string s = "hello";
const char* cstr = s.c_str(); // read-only C-style string
```

From C-style to `std::string`

```
const char* msg = "hello";
std::string s(msg); // or: std::string s = msg;
```

### 6.2.12 When to Use Which?

**Use C-style strings if:**

- You're working in legacy systems or interfacing with C APIs.
- Memory footprint is extremely critical (e.g., embedded systems).

**Prefer `std::string` when:**

- Writing modern C++ code.
- You want safety, flexibility, and ease of use.
- You need to manipulate strings extensively (e.g., appending, searching, replacing).

### 6.2.13 Summary

| Feature | C-style Strings | `std::string` |
|---|---|---|
| Memory Management | Manual | Automatic |
| Null Terminator | Required (`'\0'`) | Handled internally |
| Bounds Checking | No | Yes (via `.at()`) |
| Operations | Via `<cstring>` | Built-in methods & operators |
| Safety & Convenience | Low | High |

In modern C++ programming, `std::string` is almost always the better choice unless you're dealing with performance-critical code or legacy C functions. Understanding both, however, gives you flexibility and the ability to work across diverse C++ environments.

## 6.3 Common String Operations

Strings are one of the most frequently used data types in any program. Whether you're displaying output, reading user input, or processing text data, you'll perform a range of operations like concatenation, comparison, measuring length, and extracting substrings. In C++, you can perform these operations using either **C-style strings** (null-terminated character arrays) or the more modern and robust **`std::string`** class.

In this section, we'll cover essential string operations using both methods and highlight best practices and potential issues to avoid.

### 6.3.1 String Concatenation

C-style Strings

Concatenating C-style strings requires using functions from the `<cstring>` library, such as `strcat`.

Full runnable code:

```cpp
#include <iostream>
#include <cstring>

int main() {
    char str1[20] = "Hello ";
    char str2[] = "World";
    strcat(str1, str2); // str1 becomes "Hello World"
    std::cout << str1 << std::endl;
    return 0;
}
```

> **Warning:** Make sure `str1` has enough space to hold the result. Otherwise, `strcat` can lead to memory corruption or crashes.

### 6.3.2 `std::string`

Concatenation with `std::string` is straightforward and safe, thanks to operator overloading:

Full runnable code:

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

int main() {
    std::string first = "Hello ";
    std::string second = "World";
    std::string result = first + second;
    std::cout << result << std::endl;
    return 0;
}
```

No need to worry about buffer sizes—`std::string` handles memory automatically.

### 6.3.3 String Comparison

#### C-style Strings

C-style strings cannot be compared using `==` directly. Instead, use `strcmp`:

Full runnable code:

```
#include <iostream>
#include <cstring>

int main() {
    char a[] = "apple";
    char b[] = "banana";

    if (strcmp(a, b) < 0) {
        std::cout << a << " comes before " << b << std::endl;
    }
    return 0;
}
```

- `strcmp(s1, s2)` returns:
  - `0` if strings are equal
  - Negative if `s1 < s2`
  - Positive if `s1 > s2`

**`std::string`**

`std::string` supports `==`, `!=`, `<`, `>`, and other comparison operators directly:

```
std::string a = "apple";
std::string b = "banana";

if (a < b) {
    std::cout << a << " comes before " << b << std::endl;
}
```

This syntax is cleaner, safer, and easier to understand.

### 6.3.4 String Length

**C-style Strings**

Use `strlen` to determine the number of characters (excluding the null terminator):

```
char word[] = "hello";
std::cout << "Length: " << strlen(word) << std::endl;
```

> WARNING Ensure the string is null-terminated. `strlen` will keep counting until it finds `'\0'`, which may lead to reading garbage memory.

**`std::string`**

Use `.length()` or `.size()` methods:

```
std::string word = "hello";
std::cout << "Length: " << word.length() << std::endl;
```

Both methods return the same value and are safe to use.

### 6.3.5 Substring Extraction

#### C-style Strings

To extract a substring, you must manually copy the desired range using `strncpy` or a loop:

Full runnable code:

```
#include <iostream>
#include <cstring>

int main() {
    char str[] = "abcdef";
    char sub[4];
    strncpy(sub, str + 2, 3); // Copy 3 characters starting from index 2
    sub[3] = '\0'; // Manually null-terminate
    std::cout << sub << std::endl; // Output: "cde"
    return 0;
}
```

WARNING Always null-terminate manually when using `strncpy`.

#### `std::string`

Use the `substr()` method:

```
std::string str = "abcdef";
std::string sub = str.substr(2, 3); // Starting at index 2, length 3
std::cout << sub << std::endl; // Output: "cde"
```

Much simpler and safer than working with character arrays.

### 6.3.6 Memory Safety with C-style Strings

When using C-style strings, the burden of memory safety is on the programmer:

- **Always** allocate enough space for the string and the null terminator.
- **Always** null-terminate strings manually if you copy characters yourself.
- Avoid buffer overflows caused by `strcpy` and `strcat` by using `strncpy` and `strncat`.
- Consider using `std::string` for any non-trivial operations, especially when safety and maintainability matter.

```
// Unsafe
char name[5];
strcpy(name, "Charles"); // May overflow!

// Safer with std::string
```

```
std::string name = "Charles";
```

### 6.3.7 Summary

| Operation | C-style Strings | `std::string` |
|---|---|---|
| Concatenation | `strcat(dest, src)` | `result = str1 + str2` |
| Comparison | `strcmp(a, b)` | `a == b`, `a < b`, etc. |
| Length | `strlen(str)` | `str.length()` |
| Substring | Use `strncpy()` or manual copying | `str.substr(start, length)` |
| Safety | Manual memory and termination required | Automatically managed, bounds-safe |

C-style string functions are fast but risky, while `std::string` offers powerful features, intuitive syntax, and safety. As a beginner and for most real-world projects, **prefer `std::string`** unless you have a specific reason to use character arrays.

Understanding both approaches gives you flexibility and helps when interacting with legacy code or system-level APIs that require C-style strings.

## 6.4 Arrays of Strings and String Manipulation Examples

In C++, it's common to work with **collections of strings**, such as lists of names, commands, or other textual data. These collections can be represented as **arrays of strings**. Depending on whether you use traditional C-style strings or the modern `std::string` class, the way you declare and manipulate these arrays differs.

In this section, you will learn how to declare arrays of strings, how to iterate and modify them, and see practical examples including simple text processing tasks.

### 6.4.1 Arrays of C-style Strings

A common way to store multiple C-style strings is as an array of **pointers to characters**:

```
const char* fruits[] = {"Apple", "Banana", "Cherry"};
```

Here, `fruits` is an array of three pointers, each pointing to the first character of a null-terminated string literal.

### 6.4.2 Accessing Elements

You can access individual strings and their characters using indices:

```
std::cout << fruits[1] << std::endl;    // Output: Banana
std::cout << fruits[2][0] << std::endl; // Output: C (first letter of "Cherry")
```

### 6.4.3 Iterating Over the Array

Using a loop to print all fruits:

```
int count = sizeof(fruits) / sizeof(fruits[0]);

for (int i = 0; i < count; i++) {
    std::cout << fruits[i] << std::endl;
}
```

### 6.4.4 Modifying C-style Strings

Strings stored as string literals (`const char*`) are **read-only**. Attempting to modify them leads to undefined behavior. If you need modifiable C-style strings, you must store them in a 2D char array:

```
char fruits[3][10] = {"Apple", "Banana", "Cherry"};

fruits[1][0] = 'b'; // Changes "Banana" to "banana"
std::cout << fruits[1] << std::endl; // Output: banana
```

But be careful to allocate enough space for each string to avoid buffer overflows.

### 6.4.5 Arrays of `std::string`

The preferred way to handle string arrays in modern C++ is using `std::string`:

Full runnable code:

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

int main() {
    std::string colors[] = {"Red", "Green", "Blue"};

    int size = sizeof(colors) / sizeof(colors[0]);
    for (int i = 0; i < size; i++) {
        std::cout << colors[i] << std::endl;
    }

    // Modifying a string
    colors[1] = "Yellow";
    std::cout << "Modified: " << colors[1] << std::endl;

    return 0;
}
```

`std::string` objects are dynamic and safe to modify. You don't need to worry about manual memory management or null terminators.

### 6.4.6 Example: Storing and Searching Names

Full runnable code:

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

int main() {
    std::string names[] = {"Alice", "Bob", "Charlie", "Diana"};
    int size = sizeof(names) / sizeof(names[0]);

    std::string search;
    std::cout << "Enter a name to search: ";
    std::getline(std::cin, search);

    bool found = false;
    for (int i = 0; i < size; i++) {
        if (names[i] == search) {
            std::cout << search << " found at index " << i << std::endl;
            found = true;
            break;
        }
    }

    if (!found) {
        std::cout << search << " not found in the list." << std::endl;
    }

    return 0;
}
```

This program uses an array of `std::string` to store names, then allows the user to search for a name in the array.

### 6.4.7 Simple Text Processing: Counting Word Lengths

Suppose you want to count and display the length of each word in a list:

Full runnable code:

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

int main() {
    std::string words[] = {"programming", "language", "array", "string"};
    int size = sizeof(words) / sizeof(words[0]);

    for (int i = 0; i < size; i++) {
        std::cout << words[i] << " has length " << words[i].length() << std::endl;
    }

    return 0;
}
```

Here, `std::string::length()` returns the length of each string, showcasing easy access to string properties in an array context.

### 6.4.8 Summary

| Feature | Arrays of C-style Strings | Arrays of `std::string` |
|---|---|---|
| Declaration | `const char* arr[] = {...}` or 2D `char` arrays | `std::string arr[] = {...}` |
| Modifiability | String literals are read-only; modifiable with 2D arrays | Fully modifiable and safer |
| Memory Management | Manual allocation, fixed size | Automatic memory management |
| Safety | Risk of buffer overflow and undefined behavior | Safer and easier to use |
| Functionality | Limited to C string functions (`strcpy`, etc.) | Rich string methods and operators |

Using arrays of `std::string` is recommended for most applications because they simplify string handling and avoid common pitfalls related to C-style strings. However, understanding arrays of C-style strings is still useful, especially when working with legacy code or interfacing with C libraries.

Mastering arrays of strings equips you to build more complex programs involving lists, menus, text processing, and beyond. As you continue, you will learn how to manipulate these arrays further and combine them with other data structures to create versatile software.

# Chapter 7.

# Pointers and References

# 7 Pointers and References

## 7.1 Pointer Basics and Syntax

In C++, **pointers** are one of the most powerful and fundamental features. A pointer is a special type of variable that **stores the memory address** of another variable. Understanding pointers is essential because they enable you to directly manipulate memory, interact with arrays efficiently, work with dynamic memory, and implement complex data structures.

### 7.1.1 What is a Pointer?

A **pointer** holds the **address** of another variable in memory, not the variable's actual value. Think of a pointer as a signpost pointing to a specific location where data is stored.

For example, if you have an integer variable `x` stored at some memory location, a pointer to `x` stores that location's address.

### 7.1.2 Declaring Pointers

To declare a pointer, you specify the data type it points to, followed by an asterisk `*`, then the pointer name:

```cpp
int* ptr;  // Pointer to an integer
```

The `int*` means "pointer to an integer." The type is important because the pointer needs to know how many bytes to access when dereferencing.

### 7.1.3 Assigning Addresses to Pointers: The Address-of Operator (&)

To make a pointer point to a variable, use the **address-of operator** `&`, which returns the memory address of a variable:

```cpp
int x = 42;
int* ptr = &x;  // ptr now holds the address of x
```

Here:

- `x` is an integer variable.
- `&x` is the address of `x`.
- `ptr` stores this address.

### 7.1.4 Dereferencing Pointers: The Dereference Operator (*)

To access or modify the value stored at the memory address a pointer holds, use the **dereference operator** `*`:

```
int value = *ptr;  // Access the value pointed to by ptr (which is 42)
*ptr = 100;        // Change the value at ptr's address to 100
```

In the example:

- `*ptr` means "the value stored at the address in `ptr`."
- Changing `*ptr` changes the value of `x` because `ptr` points to `x`.

### 7.1.5 Example: Pointer Basics in Action

Full runnable code:

```
#include <iostream>

int main() {
    int number = 10;
    int* p = &number;  // Pointer p holds the address of number

    std::cout << "Value of number: " << number << std::endl;
    std::cout << "Address of number (&number): " << &number << std::endl;
    std::cout << "Value of p (address stored): " << p << std::endl;
    std::cout << "Value pointed to by p (*p): " << *p << std::endl;

    *p = 20;  // Modify number using the pointer
    std::cout << "New value of number after *p = 20: " << number << std::endl;

    return 0;
}
```

**Output:**

```
Value of number: 10
Address of number (&number): 0x7ffcb2e1eac4
Value of p (address stored): 0x7ffcb2e1eac4
Value pointed to by p (*p): 10
New value of number after *p = 20: 20
```

### 7.1.6 Important Pointer Concepts

Pointer Type Matters

The type of pointer (`int*`, `char*`, `double*`) tells the compiler how many bytes to read or write when dereferencing.

```
double d = 3.14;
double* dp = &d;  // Points to a double (8 bytes on most systems)
```

Trying to assign incompatible pointer types without casting will cause errors.

### 7.1.7 Uninitialized Pointers

Declaring a pointer without initializing it leads to an **uninitialized pointer**, which points to an unknown memory location. Accessing or dereferencing such pointers causes **undefined behavior** and often crashes your program.

```
int* p;    // Uninitialized pointer - dangerous!
std::cout << *p;  // Undefined behavior! Don't do this.
```

**Always initialize pointers** before use:

```
int x = 5;
int* p = &x;  // Safe initialization
```

### 7.1.8 Null Pointers

A **null pointer** points to nothing. It is useful to indicate that a pointer doesn't currently point to valid memory.

In C++, use `nullptr` (introduced in C++11) to represent null pointers:

```
int* p = nullptr;
if (p == nullptr) {
    std::cout << "Pointer is null and safe to check before dereferencing." << std::endl;
}
```

Dereferencing a null pointer also causes undefined behavior, so always check for `nullptr` before dereferencing pointers that may be null.

### 7.1.9 Summary

| Concept | Description | Example |
|---|---|---|
| Pointer declaration | Variable that stores an address | `int* p;` |
| Address-of operator | Gets memory address of a variable | `p = &x;` |

| Concept | Description | Example |
|---|---|---|
| Dereference operator | Accesses/modifies value at the address stored | `*p = 10;` |
| Uninitialized pointer | Pointer without assigned address (dangerous) | `int* p;` (don't dereference) |
| Null pointer | Pointer set to point to no valid memory | `int* p = nullptr;` |

### 7.1.10 Practical Tips for Working with Pointers

- **Always initialize pointers** either with a valid address or `nullptr`.
- **Check for `nullptr`** before dereferencing pointers that might be null.
- Avoid dangling pointers (pointers to memory that has been freed).
- Use pointers carefully; improper use is a common source of bugs and crashes.
- Use references or smart pointers in advanced C++ to reduce manual pointer errors (covered later).

Pointers unlock powerful programming techniques by letting you directly manage memory and efficiently manipulate data. Mastering pointer basics is a key step toward becoming a proficient C++ programmer.

## 7.2 Pointer Arithmetic and Arrays

One of the most powerful uses of pointers in C++ is **working with arrays**. Since arrays are stored in contiguous memory locations, pointers allow you to traverse, access, and manipulate array elements efficiently by performing arithmetic on addresses.

In this section, we'll explore the relationship between pointers and arrays, understand how pointer arithmetic works, and see practical examples demonstrating how to iterate through arrays using pointers. We'll also discuss important safety considerations like avoiding out-of-bounds access.

### 7.2.1 The Relationship Between Arrays and Pointers

In C++, the name of an array acts like a **pointer to its first element**. For example, consider this array:

```cpp
int numbers[] = {10, 20, 30, 40, 50};
```

- The expression `numbers` (without an index) is equivalent to the address of the first element: `&numbers[0]`.

- You can assign it to a pointer of the same type:

```
int* ptr = numbers;  // Points to numbers[0]
```

This means `ptr` and `numbers` both point to the first element of the array.

### 7.2.2 Pointer Arithmetic Basics

Pointers hold memory addresses. Because arrays store elements contiguously, incrementing or decrementing a pointer moves it to the next or previous element of the array, not just the next byte.

- **Incrementing a pointer (`ptr++`) moves it to the next element in the array.**
- **Decrementing a pointer (`ptr--`) moves it to the previous element.**

For example:

```
int arr[] = {1, 2, 3};
int* p = arr;  // Points to arr[0]

p++;  // Now points to arr[1]
p++;  // Now points to arr[2]
p--;  // Back to arr[1]
```

The pointer moves by the size of the element type it points to. For an `int` on most systems, that's usually 4 bytes.

### 7.2.3 Accessing Array Elements Using Pointers

You can access the value a pointer points to using the **dereference operator** `*`:

```
int arr[] = {5, 10, 15};
int* p = arr;

std::cout << *p << std::endl;       // Outputs 5 (arr[0])
std::cout << *(p + 1) << std::endl; // Outputs 10 (arr[1])
std::cout << *(p + 2) << std::endl; // Outputs 15 (arr[2])
```

Here, `p + 1` points to the second element, and so on.

### 7.2.4 Iterating Through an Array Using a Pointer

Instead of using array indices, you can use pointers to traverse an array:

Full runnable code:

```cpp
#include <iostream>

int main() {
    int data[] = {100, 200, 300, 400, 500};
    int* ptr = data;                        // Points to data[0]
    int length = sizeof(data) / sizeof(data[0]);

    for (int i = 0; i < length; i++) {
        std::cout << *(ptr + i) << " ";
    }
    std::cout << std::endl;

    // Alternative: increment pointer directly
    for (int* p = data; p < data + length; p++) {
        std::cout << *p << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Both loops print the array elements. The second loop shows a common idiom: using a pointer variable `p` to traverse from the start (`data`) to the end (`data + length`).

### 7.2.5 Pointer and Array Notation Are Interchangeable

Because of the close relationship between arrays and pointers, you can use pointer arithmetic instead of array indexing:

```cpp
int arr[3] = {7, 8, 9};
int* p = arr;

std::cout << arr[1] << std::endl;  // Outputs 8
std::cout << *(p + 1) << std::endl; // Also outputs 8
```

Both `arr[1]` and `*(p + 1)` access the second element.

### 7.2.6 Caution: Avoid Out-of-Bounds Access

Unlike higher-level languages, C++ **does not check** if a pointer goes beyond the array boundaries. Accessing memory outside the valid range is **undefined behavior** and can cause crashes or data corruption.

Example of dangerous code:

```cpp
int numbers[3] = {1, 2, 3};
int* p = numbers;
```

```
std::cout << *(p + 3) << std::endl;  // Out-of-bounds! Undefined behavior.
```

Here, `p + 3` points **one element past** the array end, which is invalid.

**Always ensure your pointers stay within the array bounds when performing arithmetic.**

### 7.2.7 Example: Modifying Array Elements Using Pointers

Pointers can also be used to modify array elements:

```
int scores[] = {50, 60, 70};
int* p = scores;

for (int i = 0; i < 3; i++) {
    *(p + i) += 10;  // Increase each score by 10
}

for (int i = 0; i < 3; i++) {
    std::cout << scores[i] << " ";  // Outputs: 60 70 80
}
std::cout << std::endl;
```

The pointer-based loop increments each element's value directly.

### 7.2.8 Summary

| Concept | Description | Example |
|---|---|---|
| Array name as pointer | Name of array points to first element | `int* p = arr;` |
| Pointer arithmetic | `p++` moves to next element, `p--` moves back | `*(p + 1)` accesses next element |
| Iterating with pointers | Use pointer increments to traverse an array | `for (int* p = arr; p < arr + n; p++)` |
| Out-of-bounds danger | Accessing beyond array size causes undefined behavior | Avoid `*(p + n)` if array size is n |
| Modifying via pointers | Dereference pointer to change array contents | `*(p + i) = new_value;` |

### 7.2.9 Conclusion

Pointer arithmetic lets you efficiently traverse and manipulate arrays in C++. The close relationship between arrays and pointers is a cornerstone of C++ programming, enabling fast and flexible access to data.

However, with this power comes responsibility. Always ensure pointers remain within valid bounds to prevent undefined behavior. Mastering pointer arithmetic with arrays prepares you for advanced topics like dynamic arrays, pointer-to-pointer constructs, and low-level memory management covered in later chapters.

## 7.3 References and Reference Variables

In C++, **references** provide a convenient and safer alternative to pointers in many situations. A reference is essentially an **alias** or another name for an existing variable. Once initialized, a reference refers to the same memory location as the original variable, allowing you to access or modify that variable through the reference.

In this section, we will explore what references are, how to declare and use them, their advantages over pointers for parameter passing, and key differences between references and pointers.

### 7.3.1 What Is a Reference?

A **reference variable** is a reference (alias) to an existing variable. It must be initialized at declaration and cannot be changed to refer to another variable later.

```cpp
int x = 10;
int& ref = x;  // ref is a reference to x
```

Here, `ref` and `x` refer to the **same** memory location. Changing one affects the other:

```cpp
ref = 20;  // Changes x to 20
std::cout << x;  // Outputs 20
```

### 7.3.2 Syntax for Declaring References

The syntax for declaring a reference is:

```cpp
type& reference_name = variable;
```

- The ampersand (`&`) after the type indicates a reference.

- Initialization is mandatory — you cannot have an uninitialized reference.
- After initialization, the reference **cannot be reseated** to refer to another variable.

Example:

```
double value = 3.14;
double& refValue = value;
```

### 7.3.3 How Are References Different from Pointers?

| Feature | Reference | Pointer |
|---|---|---|
| Syntax | Declared with `&` after type | Declared with `*` after type |
| Initialization | Must be initialized immediately | Can be declared without initialization |
| Nullability | Cannot be null (must refer to valid variable) | Can be null (`nullptr`) |
| Reassignment | Cannot be reassigned to refer to another variable | Can point to different variables |
| Dereferencing | Implicit; no operator needed | Requires `*` operator |
| Usage | Acts as an alias | Holds memory address |

Because references cannot be null or reassigned, they are safer and easier to use in many cases, especially for function parameters.

### 7.3.4 Typical Use Case: Passing Arguments to Functions

References are commonly used to pass variables **by reference** to functions, allowing the function to modify the original argument without copying it.

### 7.3.5 Pass-by-value (copy):

```
void increment(int n) {
    n = n + 1;  // Modifies local copy only
}

int main() {
    int x = 5;
    increment(x);
    std::cout << x;  // Outputs 5 (unchanged)
}
```

### 7.3.6 Pass-by-reference:

```
void increment(int& n) {
    n = n + 1;  // Modifies original variable
}

int main() {
    int x = 5;
    increment(x);
    std::cout << x;  // Outputs 6 (modified)
}
```

Passing by reference avoids the overhead of copying large objects and allows functions to modify the caller's variables safely.

### 7.3.7 Simple reference aliasing

```
int a = 100;
int& refA = a;  // refA is another name for a

std::cout << a << ", " << refA << std::endl;  // Outputs: 100, 100

refA = 200;  // Modify through reference
std::cout << a << std::endl;  // Outputs: 200
```

### 7.3.8 Function modifying a variable by reference

```
void doubleValue(int& num) {
    num *= 2;
}

int main() {
    int val = 7;
    doubleValue(val);
    std::cout << val;  // Outputs 14
}
```

### 7.3.9 References to Constants (`const` References)

Sometimes you want to pass large objects by reference **without allowing modification**. This is done using `const` references:

```cpp
void printValue(const int& n) {
    std::cout << n << std::endl;
}

int main() {
    int x = 42;
    printValue(x);  // Safe; printValue cannot modify x
}
```

`const` references combine efficiency (no copying) with safety (no accidental changes).

### 7.3.10 When to Use References vs. Pointers

| Scenario | Use References | Use Pointers |
|---|---|---|
| Function parameters | Prefer references for clarity and safety | Use pointers when null or reassignment needed |
| Object aliasing | Use references when alias is permanent | Use pointers for optional or changeable aliases |
| Dynamic memory management | Not applicable | Required for managing heap-allocated memory |
| Nullable references | Not possible | Pointers can be null |

### 7.3.11 Summary

- A **reference** is an alias to an existing variable; it must be initialized and cannot be changed afterward.
- References simplify syntax by eliminating the need to dereference explicitly.
- They are safer than pointers because they cannot be null or uninitialized.
- Commonly used for **pass-by-reference** in functions, enabling efficient and clear code.
- `const` references provide read-only access to variables, useful for preventing modification while avoiding copies.

### 7.3.12 Final Note

References help make C++ code more readable and less error-prone compared to pointers in many situations. Once you are comfortable with pointers, mastering references is essential for writing clean and efficient C++ programs. In the next section, we will explore **dynamic memory allocation** and how pointers interact with memory on the heap.

## 7.4 Dynamic Memory Allocation with `new` and `delete`

In C++, **dynamic memory allocation** allows your program to request and release memory during runtime, rather than relying solely on fixed-size variables defined at compile time. This flexibility is essential for handling data whose size or lifetime cannot be determined in advance, such as user input, data structures that grow or shrink, or large objects that shouldn't be stored on the stack.

C++ provides the **`new`** and **`delete`** operators to allocate and deallocate memory on the **heap** (also called free store). This section explains how to use these operators, how to allocate both single variables and arrays dynamically, and how to avoid common pitfalls such as memory leaks and dangling pointers.

### 7.4.1 Why Dynamic Memory Allocation?

When you declare variables normally, such as:

```cpp
int x = 5;
int arr[10];
```

These variables are typically allocated on the **stack**, which is a limited, fast-access memory area. The size and lifetime of stack variables are fixed by the program structure.

However, sometimes:

- You don't know the size of an array beforehand.
- You want a variable to persist beyond the scope it was created in.
- You want to allocate large objects without overflowing the stack.

This is where **dynamic memory allocation** comes in — it lets you **allocate memory at runtime on the heap**, which is a much larger pool of memory managed manually by your program.

### 7.4.2 The `new` Operator: Allocating Memory

The `new` operator dynamically allocates memory for a variable or array and returns a **pointer** to the allocated memory.

### 7.4.3 Allocating a single variable

```
int* ptr = new int;   // Allocates memory for one int
*ptr = 42;            // Assign a value to the allocated memory

std::cout << *ptr << std::endl;  // Outputs 42
```

Here:

- `new int` requests enough memory for one `int`.
- It returns a pointer to that memory.
- You must use the pointer to access or modify the value.

You can also combine allocation and initialization:

```
int* ptr = new int(10);  // Allocate and initialize to 10
```

### 7.4.4 Allocating an array

You can allocate an array dynamically by specifying the size inside square brackets `[]`:

```
int* arr = new int[5];  // Allocates an array of 5 integers

// Initialize elements
for (int i = 0; i < 5; i++) {
    arr[i] = i * 10;
}
```

The pointer `arr` points to the first element of the dynamically allocated array.

### 7.4.5 The `delete` Operator: Releasing Memory

Memory allocated with `new` is **not automatically freed** when it goes out of scope. You must manually free it using `delete` to avoid **memory leaks** — situations where allocated memory remains reserved but inaccessible, wasting resources.

### 7.4.6 Deleting a single variable

```
delete ptr;  // Frees memory allocated with new int
ptr = nullptr;  // Good practice: set pointer to nullptr after deletion
```

### 7.4.7 Deleting an array

When deleting memory allocated for an array, use `delete[]`:

```
delete[] arr;  // Frees memory allocated with new int[]
arr = nullptr;  // Avoid dangling pointer
```

### 7.4.8 Why Matching `new` and `delete` Matters

Every `new` must have a corresponding `delete`, and every `new[]` must have a corresponding `delete[]`. Failing to do so causes:

- **Memory leaks**: Memory that's never freed, eventually exhausting available memory.
- **Undefined behavior**: Using `delete` instead of `delete[]` (or vice versa) can corrupt the heap and crash your program.

### 7.4.9 Example: Dynamic Allocation and Deallocation

Full runnable code:

```
#include <iostream>

int main() {
    int* num = new int(99);
    std::cout << "Value: " << *num << std::endl;

    delete num;  // Free memory
    num = nullptr;

    int size = 3;
    int* array = new int[size];

    for (int i = 0; i < size; i++) {
        array[i] = i * i;
    }

    for (int i = 0; i < size; i++) {
        std::cout << array[i] << " ";
    }
    std::cout << std::endl;

    delete[] array;  // Free array memory
    array = nullptr;

    return 0;
}
```

### 7.4.10 Common Pitfalls and Best Practices

#### Memory Leaks

If you lose all pointers to dynamically allocated memory without deleting it, the memory is **leaked**.

```
int* p = new int(10);
p = nullptr;  // The previously allocated memory is now lost, causing a leak
```

**Best practice:** Always delete what you allocate and avoid losing pointer references before deletion.

#### Dangling Pointers

A **dangling pointer** is a pointer pointing to memory that has been freed/deleted.

```
int* p = new int(5);
delete p;
std::cout << *p;  // Undefined behavior: p is dangling
```

**Best practice:** After `delete`, set the pointer to `nullptr` to avoid accidentally using dangling pointers:

```
delete p;
p = nullptr;
```

#### Double Deletion

Calling `delete` on the same pointer twice can cause crashes or corruption.

```
int* p = new int(3);
delete p;
delete p;  // Error: double deletion
```

**Best practice:** Setting pointers to `nullptr` after deletion prevents double deletion because deleting a null pointer is safe (does nothing):

```
delete p;
p = nullptr;
delete p;  // Safe; no effect
```

#### Mixing `new`/`delete` and `new[]`/`delete[]`

Using `delete` to free memory allocated with `new[]` (or vice versa) is undefined behavior.

Correct:

```
int* p1 = new int;
delete p1;

int* p2 = new int[5];
delete[] p2;
```

Incorrect:

```
int* p2 = new int[5];
delete p2;  // Wrong! Use delete[]
```

### 7.4.11 Summary

| Concept | Description | Example |
|---|---|---|
| Dynamic allocation | Allocate memory on the heap at runtime | `int* p = new int(10);` |
| Deallocation | Free heap memory to avoid leaks | `delete p;` or `delete[] p;` |
| Single variable | Use `new` and `delete` | `int* p = new int; delete p;` |
| Arrays | Use `new[]` and `delete[]` | `int* arr = new int[10]; delete[] arr;` |
| Memory leak | Forgetting to delete allocated memory | Causes wasted memory |
| Dangling pointer | Pointer referring to deleted memory | Access causes undefined behavior |
| Best practice | Set pointer to `nullptr` after deletion | `delete p; p = nullptr;` |

### 7.4.12 Final Thoughts

Dynamic memory allocation with `new` and `delete` gives you great power but also responsibility. Improper use can lead to subtle bugs and resource issues. As you gain experience, you will learn about safer alternatives like **smart pointers** (`std::unique_ptr`, `std::shared_ptr`) introduced in modern C++ that automate memory management and reduce errors.

For now, mastering manual dynamic memory management is essential to understand how C++ programs work "under the hood" and prepares you for advanced programming concepts ahead.

# Chapter 8.

# Structures and Enumerations

# 8 Structures and Enumerations

## 8.1 Defining and Using `struct`

In C++, a **`struct`** is a user-defined data type that groups together related variables, called **members**, into a single unit. This makes it easier to organize and manage complex data by logically associating different pieces of information.

Structs are widely used to model real-world objects or concepts where multiple attributes belong together, such as a point in 2D space, a student record, or an employee profile.

### 8.1.1 Defining a `struct`

The syntax to define a struct is:

```
struct StructName {
    // Members: variables of any type
    dataType member1;
    dataType member2;
    // ...
};
```

For example, a simple struct to represent a point in 2D coordinates:

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

Here, `Point` is a new data type with two members: `x` and `y`, both of type `int`.

### 8.1.2 Declaring and Initializing Struct Variables

Once a struct is defined, you can declare variables of that type:

```
Point p1;  // Declare a Point variable named p1
```

You can access members using the **dot operator (`.`)**:

```
p1.x = 10;
p1.y = 20;
```

### 8.1.3 Initialization

C++ supports several ways to initialize structs:

```
Point p2 = {30, 40};  // Aggregate initialization

Point p3;             // Default initialization
p3.x = 50;
p3.y = 60;
```

In modern C++ (C++11 and later), you can also use **brace initialization**:

```
Point p4{70, 80};
```

### 8.1.4 Accessing Struct Members Using Pointers

When you have a pointer to a struct, you access members with the **arrow operator (`->`)**, which dereferences the pointer and accesses the member in one step.

```
Point* ptr = &p1;

ptr->x = 100;  // Equivalent to (*ptr).x = 100;
ptr->y = 200;
```

Using the arrow operator is more concise and readable than dereferencing first.

### 8.1.5 Practical Example: Student Record

Let's create a struct to represent a student's information:

```
struct Student {
    int id;
    char name[50];
    float grade;
};
```

You can declare and initialize students like this:

```
Student s1 = {101, "Alice", 89.5f};
Student s2;

s2.id = 102;
strcpy(s2.name, "Bob");
s2.grade = 92.0f;
```

Using a pointer:

```
Student* ptr = &s1;
std::cout << "Student Name: " << ptr->name << std::endl;
```

### 8.1.6 Difference Between `struct` and `class`

In C++, `struct` and `class` are very similar, and both define user-defined types. The key differences are:

| Feature | `struct` | `class` |
|---|---|---|
| Default member access | **public** | **private** |
| Default inheritance | **public** | **private** |

For example:

```
struct A {
    int x;  // public by default
};

class B {
    int x;  // private by default
};
```

Because of this, structs are typically used for **plain data structures** with mostly public members, while classes are preferred for more complex data with encapsulation and member functions.

### 8.1.7 Summary

- A `struct` groups related variables (members) into a single user-defined type.
- Members are accessed using the **dot (`.`)** operator for variables and **arrow (`->`)** for pointers.
- You can initialize struct variables with brace initialization or by assigning members individually.
- Structs are commonly used to represent objects like points, students, or employees.
- The main difference between `struct` and `class` is default access level (`public` vs. `private`).

### 8.1.8 Example: Using Structs to Group Data

Full runnable code:

```
#include <iostream>
#include <cstring>

struct Book {
    char title[100];
    char author[50];
    int year;
};

int main() {
    Book book1 = {"The C++ Journey", "John Doe", 2023};

    Book book2;
    strcpy(book2.title, "Learning C++");
    strcpy(book2.author, "Jane Smith");
    book2.year = 2024;

    std::cout << "Book 1: " << book1.title << " by " << book1.author
              << " (" << book1.year << ")" << std::endl;

    std::cout << "Book 2: " << book2.title << " by " << book2.author
              << " (" << book2.year << ")" << std::endl;

    return 0;
}
```

By mastering structs, you can efficiently group data and build more organized, maintainable C++ programs. Next, we will explore **enumerations** (`enum` and `enum class`) that help define named sets of constants to further improve code clarity.

## 8.2 Enumerations (`enum` and `enum class`)

In programming, it's common to work with a set of related named constants — such as days of the week, colors, or states of a process. Using raw numbers for these constants can make code hard to read and maintain. C++ provides a powerful feature called **enumerations** (or **enums**) to solve this problem by allowing you to define a set of named integral constants that represent meaningful values.

In this section, we'll explore traditional C++ enums and the modern, safer alternative introduced in C++11 — **scoped enums** (`enum class`). We will see how to declare enums, assign values, and use them in your programs.

### 8.2.1 Traditional `enum`

The classic way to define an enumeration is with the `enum` keyword:

```
enum Color {
    Red,
    Green,
    Blue
};
```

Here, `Color` is an enumeration type, and `Red`, `Green`, and `Blue` are **enumerators** — named constants representing integral values.

By default:

- The first enumerator (`Red`) has the value `0`.
- Each subsequent enumerator increments by 1 automatically (`Green` is 1, `Blue` is 2).
- You can explicitly specify values if needed:

```
enum Status {
    Pending = 1,
    Approved = 5,
    Rejected = 10
};
```

### 8.2.2 Using Traditional Enums

You can declare variables of the enum type and assign enumerators:

```
Color myColor = Red;

if (myColor == Green) {
    std::cout << "Color is green.\n";
} else {
    std::cout << "Color is not green.\n";
}
```

### 8.2.3 Underlying Type and Implicit Conversion

By default, the underlying type of a traditional enum is an integral type (usually `int`), but it is **not strongly typed**. This means:

- Enumerators implicitly convert to integers.
- Integers can be assigned to enum variables without errors (though this is usually unsafe).

Example:

```
Color c = Red;
int n = c;  // Implicit conversion to int: n == 0

c = static_cast<Color>(5);  // Allowed but potentially unsafe
```

Because of these implicit conversions, traditional enums can sometimes lead to bugs when mixing enum values and integers unintentionally.

### 8.2.4 Scoped Enumerations (`enum class`)

To address the shortcomings of traditional enums, C++11 introduced **scoped enumerations** using the syntax:

```cpp
enum class Color {
    Red,
    Green,
    Blue
};
```

Scoped enums have several important advantages:

1. **Strong typing:** They do not implicitly convert to integers.
2. **Scoped names:** Enumerators are accessed with the enum name as a prefix, avoiding polluting the enclosing namespace.
3. **Custom underlying type:** You can specify the underlying integral type explicitly.

### 8.2.5 Using Scoped Enums

You must use the **scope resolution operator** `::` to access enumerators:

```cpp
Color color = Color::Green;

if (color == Color::Red) {
    std::cout << "Red color selected.\n";
}
```

Attempting to assign an integer directly will cause a compile-time error:

```cpp
Color c = 1;           // Error: cannot convert int to Color
int n = c;             // Error: no implicit conversion to int

int n = static_cast<int>(c);  // Explicit conversion allowed
```

### 8.2.6 Specifying Underlying Type

You can specify the underlying integral type for a scoped enum explicitly:

```cpp
enum class Status : unsigned int {
    Pending = 1,
    Approved = 5,
    Rejected = 10
};
```

Specifying the type can be useful for memory optimization or interfacing with external APIs.

### 8.2.7 Practical Example: Using `enum` and `enum class`

Traditional enum example

```
enum Direction { North, East, South, West };

Direction d = North;

if (d == East) {
    std::cout << "Going East\n";
}
```

Scoped enum example

```
enum class Direction { North, East, South, West };

Direction d = Direction::North;

if (d == Direction::East) {
    std::cout << "Going East\n";
}
```

Trying to compare `Direction::North` to `0` or any integer will cause an error, which improves type safety.

### 8.2.8 Summary

| Feature | Traditional `enum` | Scoped `enum class` |
|---|---|---|
| Namespaced enumerators | No; enumerators are global | Yes; must use `EnumName::` |
| Implicit conversions | Yes, to/from integers | No; explicit cast required |
| Strong typing | No | Yes |
| Specify underlying type | Limited support | Fully supported |
| Usage safety | Potentially unsafe | Safer and recommended |

### 8.2.9 When to Use Which?

- Use **traditional enums** when you want quick and simple named constants without concern for namespace pollution or strong typing.
- Prefer **scoped enums (`enum class`)** for safer, clearer, and more maintainable code, especially in modern C++ projects.

### 8.2.10 Final Notes

Enumerations make your code more readable and maintainable by giving meaningful names to integral constants. The introduction of scoped enums in C++11 enhances type safety and avoids subtle bugs common with traditional enums.

In your projects, adopting `enum class` is a best practice whenever possible. They improve clarity by requiring explicit scope and conversions, making your code easier to understand and less error-prone.

### 8.2.11 Example: Scoped Enum with Switch Statement

Full runnable code:

```cpp
#include <iostream>

enum class Day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };

void printDay(Day day) {
    switch(day) {
        case Day::Monday:
            std::cout << "Monday\n"; break;
        case Day::Tuesday:
            std::cout << "Tuesday\n"; break;
        case Day::Wednesday:
            std::cout << "Wednesday\n"; break;
        case Day::Thursday:
            std::cout << "Thursday\n"; break;
        case Day::Friday:
            std::cout << "Friday\n"; break;
        case Day::Saturday:
            std::cout << "Saturday\n"; break;
        case Day::Sunday:
            std::cout << "Sunday\n"; break;
    }
}

int main() {
    Day today = Day::Wednesday;
    printDay(today);
    return 0;
}
```

This code clearly shows how scoped enums improve safety and readability in typical scenarios like switch statements.

With this knowledge, you can confidently use enumerations in your C++ programs, improving both code clarity and robustness. Next, we will explore **type aliases** with `typedef` and `using` to simplify complex type names.

## 8.3 Using typedef and using Aliases

In C++, type names can sometimes become long or complex, especially when dealing with pointers, function pointers, or template types. To make your code easier to read, maintain, and write, C++ provides two ways to create **type aliases**: the traditional `typedef` keyword and the modern `using` keyword (introduced in C++11). Both serve the purpose of giving a new name to an existing type.

In this section, we will explore how to use `typedef` and `using` to simplify type declarations, along with practical examples and advice on when to use them.

### 8.3.1 What is a Type Alias?

A **type alias** lets you create an alternative name for an existing type. This can:

- Simplify complex type names.
- Make code more descriptive and readable.
- Reduce repetitive type declarations.
- Help when you want to change the underlying type easily in one place.

### 8.3.2 Using typedef

The `typedef` keyword has been part of C and C++ since the beginning. The syntax is:

```
typedef existing_type new_alias;
```

For example, to create an alias for `unsigned int`:

```
typedef unsigned int uint;
```

You can now use `uint` instead of `unsigned int`:

```
uint age = 25;
```

### 8.3.3 Examples of typedef

- Aliasing pointers:

```
typedef int* IntPtr;
IntPtr p1, p2;  // p1 and p2 are both pointers to int
```

- Aliasing function pointers:

```
typedef void (*FuncPtr)(int, double);
```

This declares `FuncPtr` as a pointer to a function taking `int` and `double` parameters and returning `void`.

- Aliasing template instantiations:

```
#include <vector>
typedef std::vector<int> IntVector;
```

Now, `IntVector` can be used as a shorter name for `std::vector<int>`.

### 8.3.4 Using `using` Keyword (Modern Type Aliases)

C++11 introduced the `using` keyword for type aliases, which is often clearer and more versatile than `typedef`. The syntax is:

```
using new_alias = existing_type;
```

For example:

```
using uint = unsigned int;
```

This behaves similarly to the `typedef` example but is more readable, especially for complex types.

### 8.3.5 Examples with `using`

- Pointers:

```
using IntPtr = int*;
```

- Function pointers:

```
using FuncPtr = void(*)(int, double);
```

- Template aliases (this is something `typedef` cannot do):

```
template <typename T>
using Vec = std::vector<T>;
```

You can now use `Vec<int>` instead of `std::vector<int>`.

### 8.3.6 Comparison Between `typedef` and `using`

| Feature | `typedef` | `using` |
|---|---|---|
| Syntax | `typedef old_type new_type;` | `using new_type = old_type;` |

| Feature | `typedef` | `using` |
|---|---|---|
| Template aliases | Not possible | Possible (much more powerful) |
| Readability | Can be confusing with complex types | Generally clearer and easier to read |
| Introduced | C and C++ | C++11 |

### 8.3.7 Practical Examples

Simplifying a Pointer to Function

Using `typedef`:

```
typedef int (*CompareFunc)(const void*, const void*);
```

Using `using`:

```
using CompareFunc = int(*)(const void*, const void*);
```

Now you can declare variables of this type:

```
CompareFunc comp = nullptr;
```

### 8.3.8 Template Alias Example

Instead of writing:

```
std::vector<std::pair<int, int>> vecPairs;
```

You can define an alias:

```
using Pair = std::pair<int, int>;
using PairVector = std::vector<Pair>;

PairVector vecPairs;
```

This improves code readability by giving meaningful names to complex types.

### 8.3.9 When and Why to Use Type Aliases

- **Improve code clarity:** Use aliases to give descriptive names to complicated types, especially in APIs or libraries.
- **Simplify repetitive types:** If a type appears frequently, aliasing reduces typing and mistakes.

- **Make code easier to update:** Changing the alias updates all uses, simplifying maintenance.
- **Enhance generic programming:** Template aliases with `using` make templates easier to manage and understand.

### 8.3.10 Summary

- `typedef` and `using` both create type aliases.
- `typedef` is the older form, with slightly awkward syntax for complex types.
- `using` is more readable and powerful, especially with templates.
- Use aliases to improve readability, reduce code duplication, and make maintenance easier.
- Prefer `using` in modern C++ for clearer, more expressive code.

By mastering `typedef` and `using` aliases, you will write cleaner and more maintainable C++ programs, especially as your codebase grows in complexity.

Next, we will continue exploring other ways to organize and simplify your C++ code.

# Chapter 9.

# Classes and Objects: Introduction

1. Defining Classes and Creating Objects
2. Data Members and Member Functions
3. Constructors and Destructors
4. Access Specifiers (`public`, `private`, `protected`)

# 9 Classes and Objects: Introduction

## 9.1 Defining Classes and Creating Objects

In C++, **classes** are fundamental building blocks of object-oriented programming (OOP). They serve as **blueprints** or templates that define the structure and behavior of objects. A class encapsulates data (called **data members**) and functions (called **member functions** or **methods**) that operate on that data. By defining classes, you can create objects—instances of these blueprints—that represent real-world entities or abstract concepts in your programs.

### 9.1.1 What is a Class?

Think of a class as a blueprint for a house. The blueprint itself isn't a house you can live in; it's a plan that specifies what the house should look like, how many rooms it has, where the doors and windows are, and so on. When you actually build a house based on that blueprint, you get an **object**. You can build many houses from the same blueprint, each independent and possibly different in some details.

Similarly, a class defines:

- **Data members**: Variables representing the state or properties of an object.
- **Member functions**: Functions that define behaviors or actions the object can perform.

Each **object** created from the class has its own copy of data members but shares the behavior defined by the member functions.

### 9.1.2 Syntax for Defining a Class

Here's the basic syntax for defining a class in C++:

```cpp
class ClassName {
    // Access specifier (usually public or private)
    access_specifier:
        // Data members (variables)
        dataType memberVariable1;
        dataType memberVariable2;

        // Member functions (methods)
        returnType memberFunctionName(parameters) {
            // Function body
        }
};
```

### 9.1.3 Example: Defining a Simple Class

Let's define a class named `Car` with data members to represent its brand and speed:

```cpp
class Car {
public:  // Access specifier indicating public members
    std::string brand;
    int speed;

    void display() {
        std::cout << "Brand: " << brand << ", Speed: " << speed << " km/h\n";
    }
};
```

- `brand` and `speed` are **data members** representing the state of a car.
- `display()` is a **member function** that prints the car's information.

### 9.1.4 Creating Objects (Instances) of a Class

Once you have defined a class, you can create objects of that class. An object is a variable of the class type, holding its own copy of the data members.

```cpp
Car car1;  // Create an object named car1 of type Car
Car car2;  // Another object car2
```

You can access and modify the data members of each object separately:

```cpp
car1.brand = "Toyota";
car1.speed = 120;

car2.brand = "Honda";
car2.speed = 150;
```

Calling a member function on an object:

```cpp
car1.display();  // Output: Brand: Toyota, Speed: 120 km/h
car2.display();  // Output: Brand: Honda, Speed: 150 km/h
```

### 9.1.5 Objects in Different Scopes

You can create objects in different scopes depending on your program's needs:

- **Local objects:** Declared inside functions, they exist only while the function executes.

```cpp
void test() {
    Car localCar;
    localCar.brand = "Ford";
    localCar.speed = 100;
    localCar.display();
}  // localCar is destroyed when test() ends
```

- **Global objects:** Declared outside any function, accessible throughout the program.

```
Car globalCar;  // Exists for the program's lifetime
```

- **Dynamic objects:** Created using pointers and dynamic memory allocation (covered in a later chapter).

### 9.1.6 Example Program: Class and Objects

Full runnable code:

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

class Car {
public:
    std::string brand;
    int speed;

    void display() {
        std::cout << "Brand: " << brand << ", Speed: " << speed << " km/h\n";
    }
};

int main() {
    Car car1;  // Creating an object
    car1.brand = "Tesla";
    car1.speed = 200;

    Car car2;  // Another object
    car2.brand = "BMW";
    car2.speed = 180;

    car1.display();
    car2.display();

    return 0;
}
```

### 9.1.7 Key Points to Remember

- **Class** is a blueprint; **object** is an instance of that blueprint.
- A class encapsulates data (members) and behavior (member functions).
- Objects have their own copies of data members.
- Access members using the **dot operator (.)** on the object.
- Objects can be created in local, global, or dynamic scope.

### 9.1.8 Benefits of Using Classes

Using classes allows you to:

- **Organize code:** Group related data and functions logically.
- **Reuse code:** Define a class once and create multiple objects.
- **Model real-world entities:** Represent complex entities with properties and behaviors.
- **Improve maintainability:** Encapsulate functionality, making code easier to manage and extend.

With this foundational understanding of classes and objects, you are ready to dive deeper into how to define data members and member functions, use constructors to initialize objects, and control access with specifiers — topics that we will cover in the upcoming sections.

If you are excited to bring your programs to life with powerful object-oriented concepts, you are off to a great start!

## 9.2 Data Members and Member Functions

In C++, **classes** are designed to encapsulate both **data members** (also called attributes or fields) and **member functions** (also known as methods). This combination allows an object to hold its own state and define behaviors that operate on that state. Understanding how to define, declare, and access data members and member functions is fundamental to mastering object-oriented programming.

### 9.2.1 Data Members: The Attributes of a Class

Data members are variables declared inside a class that represent the state or properties of an object. Each object you create from a class contains its own copy of these variables.

### 9.2.2 Declaring Data Members

Data members are typically declared within the class body, often grouped together for clarity:

```cpp
class Person {
public:
    std::string name;
    int age;
};
```

Here, `name` and `age` are data members describing a person's attributes.

### 9.2.3 Member Functions: The Behaviors of a Class

Member functions are functions defined inside the class that can access and manipulate data members. These functions define what operations can be performed on an object.

### 9.2.4 Declaring Member Functions

Member functions are declared inside the class and can be defined either inside or outside the class body.

Example — declaring and defining inside the class:

```cpp
class Person {
public:
    std::string name;
    int age;

    void display() {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};
```

You can also define member functions outside the class using the **scope resolution operator (::)**:

```cpp
class Person {
public:
    std::string name;
    int age;
    void display();  // Declaration only
};

// Definition outside the class
void Person::display() {
    std::cout << "Name: " << name << ", Age: " << age << std::endl;
}
```

### 9.2.5 Accessing Data Members and Member Functions

To access data members and member functions, you use the **dot operator (.)** with an object:

```cpp
Person p;
p.name = "Alice";
p.age = 30;
p.display();  // Calls the display() function
```

If you have a pointer to an object, use the **arrow operator (->)** to access members:

```
Person* ptr = &p;
ptr->name = "Bob";
ptr->age = 25;
ptr->display();
```

### 9.2.6 Encapsulation: Grouping Data and Behavior

Encapsulation is a key concept of OOP. It means bundling data and functions that operate on the data within one unit — the class — and restricting direct access to some of the object's components.

While we'll dive deeper into access control later, the idea here is that **member functions control how data members are accessed or modified**. This allows you to enforce rules, validate data, or hide the internal state from outside interference.

### 9.2.7 The this Pointer

Inside a member function, you can access the current object using a special pointer called `this`. It points to the object on which the member function was called.

Example:

```
class Person {
public:
    std::string name;
    int age;

    void setName(std::string name) {
        // To resolve the name conflict, use 'this->name' for the data member
        this->name = name;
    }

    void display() {
        std::cout << "Name: " << this->name << ", Age: " << this->age << std::endl;
    }
};
```

Here, the parameter `name` shadows the data member `name`. Using `this->name` clarifies that you want to assign the parameter to the object's member variable.

### 9.2.8 Practical Example

Let's put it all together:

Full runnable code:

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

class Person {
public:
    std::string name;
    int age;

    void setName(std::string name) {
        this->name = name;  // Assign parameter to data member
    }

    void setAge(int age) {
        if (age >= 0) {
            this->age = age;
        } else {
            std::cout << "Age cannot be negative!" << std::endl;
        }
    }

    void display() {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    Person person1;

    person1.setName("John");
    person1.setAge(28);

    person1.display();  // Output: Name: John, Age: 28

    person1.setAge(-5); // Output: Age cannot be negative!

    return 0;
}
```

In this example:

- `name` and `age` are data members.
- `setName()`, `setAge()`, and `display()` are member functions.
- `this` pointer is used to refer explicitly to the object's data members when parameters shadow member names.
- The `setAge()` method includes basic validation, demonstrating encapsulation by controlling how `age` is set.

### 9.2.9 Summary

- Classes contain **data members** (attributes) and **member functions** (methods).

- Data members store the object's state; member functions define its behavior.
- Use the dot (`.`) operator to access members through an object, and arrow (`->`) for pointers.
- The **`this` pointer** inside member functions refers to the current object and is useful when parameter names conflict with data member names.
- Encapsulation helps protect and control access to an object's data through member functions.

Mastering how to define and use data members and member functions is crucial for writing clear, organized, and maintainable C++ code that leverages the power of object-oriented design.

Next, we'll explore how to create objects with properly initialized states using **constructors and destructors**.

## 9.3 Constructors and Destructors

In C++, **constructors** and **destructors** are special member functions that play crucial roles in the lifecycle of an object. They help manage initialization and cleanup, ensuring your objects start in a valid state and release resources properly when they are no longer needed. In this section, we will explore what constructors and destructors are, their types, syntax, and practical examples to help you understand how to use them effectively.

### 9.3.1 What is a Constructor?

A **constructor** is a special member function that is automatically called when an object is created. Its primary purpose is to **initialize** the object's data members or perform setup tasks.

### 9.3.2 Key Characteristics of Constructors:

- **Same name as the class**.
- No return type—not even `void`.
- Automatically invoked when an object is instantiated.
- Can be overloaded (multiple constructors with different parameters).

### 9.3.3 Default Constructor

The simplest form of a constructor is the **default constructor**, which takes no parameters.

Full runnable code:

```cpp
#include <iostream>
#include <string>
class Person {
public:
    std::string name;
    int age;

    // Default constructor
    Person() {
        name = "Unknown";
        age = 0;
        std::cout << "Default constructor called\n";
    }

    void display() {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    Person p;  // Default constructor is called here
    p.display();  // Output: Name: Unknown, Age: 0

    return 0;
}
```

If you don't define any constructor, the compiler provides an implicit default constructor that does nothing (default initialization).

### 9.3.4 Parameterized Constructors

Constructors can take parameters to allow initializing objects with specific values when created.

Full runnable code:

```cpp
#include <iostream>
#include <string>
class Person {
public:
    std::string name;
    int age;

    // Parameterized constructor
    Person(std::string n, int a) {
        name = n;
        age = a;
        std::cout << "Parameterized constructor called\n";
    }

    void display() {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    Person p("Alice", 25);  // Calls parameterized constructor
    p.display();  // Output: Name: Alice, Age: 25

    return 0;
}
```

You can define multiple constructors with different parameters for flexibility—this is called **constructor overloading**.

### 9.3.5 Copy Constructor

The **copy constructor** is a special constructor that creates a new object as a copy of an existing object. It takes a reference to an object of the same class as its parameter.

Full runnable code:

```
#include <iostream>
#include <string>
class Person {
public:
    std::string name;
    int age;

    // Parameterized constructor
    Person(std::string n, int a) : name(n), age(a) {}

    // Copy constructor
    Person(const Person& other) {
        name = other.name;
        age = other.age;
        std::cout << "Copy constructor called\n";
    }

    void display() {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

int main() {
    Person p1("Bob", 30);
    Person p2 = p1;  // Copy constructor is called here

    p1.display();  // Output: Name: Bob, Age: 30
    p2.display();  // Output: Name: Bob, Age: 30

    return 0;
}
```

The copy constructor is invoked in several cases:

- When passing an object by value to a function.
- When returning an object from a function by value.
- When initializing one object with another of the same type.

### 9.3.6 What is a Destructor?

A **destructor** is a special member function that is called automatically when an object **goes out of scope** or is explicitly deleted. Its main purpose is to perform **cleanup tasks**, such as releasing dynamically allocated memory, closing files, or freeing other resources.

### 9.3.7 Key Characteristics of Destructors:

- Same name as the class, but preceded by a tilde (~).
- No return type and no parameters.
- Only one destructor per class (cannot be overloaded).
- Automatically invoked when the object's lifetime ends.

### 9.3.8 Syntax of Destructor

Full runnable code:

```
#include <iostream>
#include <string>
class Person {
public:
    std::string name;
    int age;

    Person(std::string n, int a) : name(n), age(a) {
        std::cout << "Constructor called\n";
    }

    ~Person() {  // Destructor
        std::cout << "Destructor called for " << name << std::endl;
    }
};
```

```
int main() {
    {
        Person p("Charlie", 40);
    }  // p goes out of scope here, destructor is called automatically

    return 0;
}
```

Output:

```
Constructor called
Name: Charlie, Age: 40
Destructor called for Charlie
```

### 9.3.9 Why Are Constructors and Destructors Important?

- **Constructors** ensure objects start with valid and meaningful data.
- **Destructors** help prevent **resource leaks** by cleaning up when objects are destroyed.
- Both are essential for managing resource acquisition and release (the **RAII** principle: Resource Acquisition Is Initialization).

### 9.3.10 Best Practices and Tips

- Use **member initializer lists** for efficiency, especially with non-primitive data members:

```
Person(std::string n, int a) : name(n), age(a) {}
```

- If your class manages dynamic memory or other resources, define a destructor to free them.
- Always ensure every `new` has a corresponding `delete` in the destructor to avoid memory leaks (covered in later chapters).
- Consider implementing or explicitly deleting the copy constructor and assignment operator if your class manages resources to avoid shallow copies (advanced topic).

### 9.3.11 Summary

| Concept | Purpose | Syntax Example |
|---|---|---|
| Default Constructor | Initialize object with default values | `ClassName();` |

| Concept | Purpose | Syntax Example |
|---|---|---|
| Parameterized Constructor | Initialize object with given values | `ClassName(type param1, type param2);` |
| Copy Constructor | Initialize a new object as a copy of another | `ClassName(const ClassName& other);` |
| Destructor | Clean up resources when object is destroyed | `~ClassName();` |

### 9.3.12 Final Thoughts

Constructors and destructors are powerful tools that let you control how your objects come into existence and how they clean up after themselves. Understanding these concepts will help you write safer, more efficient, and easier-to-maintain C++ programs.

In the next section, we will explore **access specifiers** (`public`, `private`, `protected`) to learn how to control access to your class members and further improve your class design.

## 9.4 Access Specifiers (`public`, `private`, `protected`)

One of the core principles of object-oriented programming (OOP) is **encapsulation**—the practice of bundling data and methods inside a class while controlling access to the internals of that class. In C++, **access specifiers**—`public`, `private`, and `protected`—are keywords that determine how the members (data and functions) of a class can be accessed from outside the class.

Understanding access specifiers is essential for designing robust and secure classes that protect their internal state from unintended or unauthorized modifications. Let's explore each of these access levels, how they work, and how to use them effectively.

### 9.4.1 The Three Access Specifiers in C

**`public`**

- Members declared as `public` are **accessible from anywhere** in your program.
- There are no restrictions on accessing public members, so they can be read or modified by any code that has an object of the class.
- Typically used for the **interface** of the class — the functions and data members you want users of the class to interact with directly.

**Example:**

Full runnable code:

```
#include <iostream>
#include <string>
class Rectangle {
public:
    int width;
    int height;

    int area() {
        return width * height;
    }
};

int main() {
    Rectangle rect;
    rect.width = 5;   // Accessible because 'width' is public
    rect.height = 3;  // Accessible because 'height' is public
    std::cout << "Area: " << rect.area() << std::endl;
    return 0;
}
```

### `private`

- Members declared as `private` are **only accessible within the class itself**.
- Neither code outside the class nor derived classes can access private members directly.
- This is the most restrictive access level and is commonly used to **hide internal data** and helper functions to prevent unintended interference.
- Private access supports **data hiding**, one of the pillars of OOP, improving reliability and security.

**Example:**

Full runnable code:

```
#include <iostream>
#include <string>
class Rectangle {
private:
    int width;
    int height;

public:
    void setDimensions(int w, int h) {
        if (w > 0 && h > 0) { // Validating inputs
            width = w;
            height = h;
        }
    }

    int area() {
        return width * height;
    }
};

int main() {
    Rectangle rect;
    // rect.width = 5;    // Error: 'width' is private
    rect.setDimensions(5, 3);  // Allowed via public setter method
    std::cout << "Area: " << rect.area() << std::endl;
    return 0;
}
```

In this example, direct access to `width` and `height` is blocked to prevent invalid values. Instead, a **public setter function** controls how these members are modified, ensuring data integrity.

`protected`

- Members declared as `protected` are accessible **within the class itself and its derived (subclass) classes**, but not outside these.
- It's a middle ground between `private` and `public`.
- Useful when you want derived classes to have access to base class internals, but still restrict direct access from general outside code.

**Example:**

Full runnable code:

```
#include <iostream>
#include <string>
class Shape {
protected:
    int colorCode;  // Accessible to derived classes

public:
    void setColor(int c) {
        colorCode = c;
    }
};

class Circle : public Shape {
public:
    void showColor() {
        std::cout << "Color code: " << colorCode << std::endl;  // Access allowed here
    }
};

int main() {
    Circle c;
    c.setColor(5);
    c.showColor();  // Output: Color code: 5
    // std::cout << c.colorCode;  // Error: 'colorCode' is protected, not accessible here
}
```

### Default Access Specifiers

- In a **`class`**, members are `private` by default if no access specifier is provided.
- In a **`struct`**, members are `public` by default.

Example:

```
class MyClass {
    int x;  // private by default
};

struct MyStruct {
    int x;  // public by default
};
```

### 9.4.2 Why Use Access Specifiers?

#### Encapsulation and Data Hiding

By restricting access to sensitive data members (using `private`), you prevent external code from setting invalid or inconsistent states. Instead, data can only be modified through well-defined interfaces (public member functions), where you can validate inputs and control behavior.

### 9.4.3 Improved Maintainability

When internal details are hidden, you can change the implementation without affecting the code that uses your class, as long as the public interface remains the same.

#### Controlled Inheritance

`protected` lets you allow subclasses to access and extend the base class functionality safely, without exposing internals to the outside world.

### 9.4.4 Practical Example Combining All Three

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

class BankAccount {
private:
    double balance;

protected:
```

```cpp
    int accountNumber;

public:
    std::string owner;

    BankAccount(std::string name, int accNum, double initialBalance)
        : owner(name), accountNumber(accNum), balance(initialBalance) {}

    void deposit(double amount) {
        if (amount > 0) balance += amount;
    }

    void withdraw(double amount) {
        if (amount > 0 && amount <= balance) balance -= amount;
    }

    double getBalance() {
        return balance;
    }
};

class SavingsAccount : public BankAccount {
public:
    SavingsAccount(std::string name, int accNum, double initialBalance)
        : BankAccount(name, accNum, initialBalance) {}

    void showAccountNumber() {
        std::cout << "Account Number: " << accountNumber << std::endl;  // Allowed: protected access
    }
};

int main() {
    SavingsAccount sa("Jane Doe", 12345, 1000.0);
    sa.deposit(500);
    sa.withdraw(200);
    std::cout << sa.owner << "'s balance: $" << sa.getBalance() << std::endl;
    sa.showAccountNumber();
    // std::cout << sa.accountNumber;  // Error: protected member, not accessible here
    // std::cout << sa.balance;        // Error: private member, not accessible here

    return 0;
}
```

### 9.4.5 Summary Table of Access Specifiers

| Access Specifier | Accessible From | Typical Use Case |
|---|---|---|
| `public` | Everywhere | Interface functions and accessible data |
| `private` | Only within the class | Internal data and helper functions |
| `protected` | Class and derived classes | Members for inheritance access control |

### 9.4.6 Tips for Effective Use

- Keep data members `private` to maintain control over the object's state.
- Expose only the necessary functions as `public` to interact with the class.
- Use `protected` sparingly to expose internals only to derived classes.
- Avoid making data members `public` unless you have a very good reason.
- Consistently use access specifiers to improve **code readability** and **encapsulation**.

### 9.4.7 Conclusion

Access specifiers are a vital tool to **enforce encapsulation and data hiding** in your C++ programs. They define how your class members can be accessed and help maintain **clean, secure, and maintainable code** by controlling the interaction between objects and external code. By mastering these keywords, you will build more robust classes and lay a solid foundation for advanced object-oriented design.

In the next chapter, you will learn how to use **inheritance** to create new classes from existing ones, further expanding your understanding of object-oriented programming in C++.

# Chapter 10.

# Operator Overloading

# 10 Operator Overloading

## 10.1 Overloading Arithmetic and Assignment Operators

Operator overloading is a powerful feature in C++ that allows you to redefine the behavior of built-in operators (like `+`, `-`, `*`, `/`, and `=`) for your own user-defined types such as classes and structs. This capability makes your custom types easier to use and more intuitive, enabling expressions like `a + b` or `v1 += v2` to work seamlessly, much like they do with fundamental types such as `int` or `double`.

In this section, we'll explore how to overload arithmetic and assignment operators for user-defined types, the syntax involved, the differences between member and non-member operator functions, and when to use friend functions. We'll also provide practical examples using classes like `Complex` (for complex numbers) and `Vector` (for mathematical vectors) to demonstrate the concepts.

### 10.1.1 Why Overload Operators?

Imagine you have a class representing a mathematical vector:

```cpp
class Vector {
    double x, y, z;
};
```

Without operator overloading, adding two `Vector` objects would require calling a function explicitly, such as:

```cpp
Vector v3 = add(v1, v2);
```

This can become verbose and less readable, especially when you have many operations. Operator overloading lets you write:

```cpp
Vector v3 = v1 + v2;
```

which reads naturally and improves code clarity.

### 10.1.2 Operators That Can Be Overloaded

In C++, almost all operators can be overloaded, including:

- Arithmetic operators: `+`, `-`, `*`, `/`, `%`
- Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- Comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- Stream operators: `<<`, `>>`
- And many more.

In this section, we focus on **arithmetic and assignment operators**.

### 10.1.3 Syntax for Overloading Operators

Operator overloading is done by defining a special function called an **operator function**. The syntax depends on whether the function is a **member function** of the class or a **non-member (usually friend) function**.

### 10.1.4 Member Function Overloading

- The operator function is defined inside the class.
- The left-hand operand is implicitly the object the function is called on (`*this`).
- The right-hand operand (if any) is passed as a parameter.

Example syntax for overloading the addition operator `+` as a member function:

```cpp
class Vector {
public:
    Vector operator+(const Vector& other) const;
};
```

### 10.1.5 Non-Member Function Overloading

- The operator function is defined outside the class.
- Both operands are passed as parameters.
- Sometimes declared as `friend` inside the class to access private members.

Example syntax:

```cpp
class Vector {
    double x, y, z;
public:
    friend Vector operator+(const Vector& lhs, const Vector& rhs);
};
```

### 10.1.6 Overloading Arithmetic Operators: Example with `Complex` Numbers

Let's create a simple `Complex` class and overload `+`, `-`, `*`, and `/` operators.

```cpp
#include <iostream>

```

```cpp
class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Overload + operator as a member function
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imag + other.imag);
    }

    // Overload - operator as a member function
    Complex operator-(const Complex& other) const {
        return Complex(real - other.real, imag - other.imag);
    }

    // Overload * operator as a member function
    Complex operator*(const Complex& other) const {
        return Complex(
            real * other.real - imag * other.imag,
            real * other.imag + imag * other.real
        );
    }

    // Overload / operator as a member function
    Complex operator/(const Complex& other) const {
        double denominator = other.real * other.real + other.imag * other.imag;
        return Complex(
            (real * other.real + imag * other.imag) / denominator,
            (imag * other.real - real * other.imag) / denominator
        );
    }

    void display() const {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};

int main() {
    Complex c1(4, 3);
    Complex c2(2, -1);

    Complex sum = c1 + c2;
    Complex diff = c1 - c2;
    Complex prod = c1 * c2;
    Complex quot = c1 / c2;

    std::cout << "Sum: "; sum.display();
    std::cout << "Difference: "; diff.display();
    std::cout << "Product: "; prod.display();
    std::cout << "Quotient: "; quot.display();

    return 0;
}
```

**Explanation:**

- Each operator function returns a new `Complex` object with the computed result.
- They are marked `const` because they do not modify the original objects.
- This example allows you to write arithmetic expressions involving `Complex` objects naturally.

### 10.1.7 Overloading Assignment Operators

The **assignment operator** `=` and compound assignments (`+=`, `-=`, `*=`, `/=`) can also be overloaded. Unlike arithmetic operators, assignment operators modify the object on the left side and typically return a reference to the current object (`*this`) to allow chaining.

### 10.1.8 Overloading the Assignment Operator =

C++ provides a default assignment operator, but you might want to customize it when your class manages resources like dynamic memory. Here's a simple example:

```cpp
class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Overload assignment operator
    Complex& operator=(const Complex& other) {
        if (this != &other) { // protect against self-assignment
            real = other.real;
            imag = other.imag;
        }
        return *this;
    }
};
```

### 10.1.9 Overloading Compound Assignment Operators

Compound assignments combine an operation and assignment. For example, `a += b` means `a = a + b`. You can overload `+=` to optimize this:

```cpp
class Complex {
    // ... existing members ...

public:
    // Overload += operator
    Complex& operator+=(const Complex& other) {
        real += other.real;
        imag += other.imag;
        return *this;
    }

    // Similarly, you can overload -=, *=, /=
};
```

### 10.1.10 Example: Overloading Operators for a `Vector` Class

Full runnable code:

```
#include <iostream>

class Vector {
private:
    double x, y, z;

public:
    Vector(double xVal=0, double yVal=0, double zVal=0) : x(xVal), y(yVal), z(zVal) {}

    // Overload + operator as member function
    Vector operator+(const Vector& other) const {
        return Vector(x + other.x, y + other.y, z + other.z);
    }

    // Overload += operator
    Vector& operator+=(const Vector& other) {
        x += other.x;
        y += other.y;
        z += other.z;
        return *this;
    }

    void display() const {
        std::cout << "Vector(" << x << ", " << y << ", " << z << ")\n";
    }
};

int main() {
    Vector v1(1, 2, 3);
    Vector v2(4, 5, 6);

    Vector v3 = v1 + v2;
    v3.display();  // Output: Vector(5, 7, 9)

    v1 += v2;
    v1.display();  // Output: Vector(5, 7, 9)

    return 0;
}
```

### 10.1.11 Member vs. Non-Member Functions: When to Use Which?

- **Member functions**: Use when the left-hand operand is an object of your class, and the operator logically modifies or interacts with that object.
- **Non-member functions**: Required when the left-hand operand is not of your class (e.g., overloading `operator+` to add `int + Complex`).
- **Friend functions**: Non-member functions that require access to private members. Declared `friend` inside the class.

Example of non-member friend function:

```cpp
class Complex {
    double real, imag;

public:
    Complex(double r=0, double i=0) : real(r), imag(i) {}

    friend Complex operator+(const Complex& lhs, const Complex& rhs) {
        return Complex(lhs.real + rhs.real, lhs.imag + rhs.imag);
    }
};
```

### 10.1.12 Important Rules and Tips

- Do **not** create new operators; you can only overload existing ones.
- Keep operator overloading intuitive; for example, `+` should perform addition-like behavior.
- Avoid overloading operators in ways that confuse readers.
- Overload operators in pairs when appropriate; e.g., if you overload `==`, consider overloading `!=`.
- Remember to handle self-assignment in assignment operators.
- Return by reference (`*this`) for assignment operators to allow chaining (`a = b = c`).

### 10.1.13 Summary

- Operator overloading lets you redefine built-in operators for your classes, improving code readability and usability.
- Arithmetic operators (`+`, `-`, `*`, `/`) typically return new objects representing the result.
- Assignment operators (`=`, `+=`, etc.) modify the current object and return a reference to it.
- You can implement operator overloading as member or non-member functions; friend functions allow access to private data when necessary.
- Practical examples like `Complex` and `Vector` classes illustrate how operator overloading enables natural syntax for custom types.

Mastering operator overloading makes your classes more powerful and easier to use, a valuable skill for modern C++ programming. In the next section, we'll explore overloading comparison operators to support intuitive object comparisons.

## 10.2 Overloading Comparison Operators

In C++, comparison operators allow you to compare values and make decisions based on those comparisons. For built-in types like `int`, `double`, or `char`, these operators (`==`, `!=`, `<`, `>`, `<=`, `>=`) work out of the box. However, for **user-defined types** such as classes and structs, you need to define how objects should be compared. This is where **operator overloading** comes in.

Overloading comparison operators enables your classes to be compared intuitively and consistently, making your code more readable and expressive.

### 10.2.1 Why Overload Comparison Operators?

Imagine a `Date` class representing calendar dates. You want to check if two dates are equal or whether one date is earlier than another. Without operator overloading, you'd need to write custom functions like:

```cpp
bool areEqual(const Date& d1, const Date& d2);
bool isEarlier(const Date& d1, const Date& d2);
```

With operator overloading, you can write natural comparisons like:

```cpp
if (d1 == d2) { ... }
if (d1 < d2) { ... }
```

This not only improves clarity but also integrates your objects seamlessly into standard algorithms and containers.

### 10.2.2 Relational Operators Overview

The common relational operators you can overload include:

| Operator | Meaning |
|---|---|
| `==` | Equality |
| `!=` | Inequality |
| `<` | Less than |
| `>` | Greater than |

| Operator | Meaning |
|---|---|
| `<=` | Less than or equal to |
| `>=` | Greater than or equal to |

### 10.2.3 Best Practices for Overloading Comparison Operators

1. **Implement `==` and `!=` Together**

   Always provide consistent behavior for equality (`==`) and inequality (`!=`) operators. Usually, `!=` can be implemented simply as the negation of `==`.

2. **Maintain Logical Consistency**

   If you overload ordering operators (`<`, `>`, `<=`, `>=`), they should be consistent with each other to avoid confusion and bugs. For example, if `a < b` is true, then `b > a` should also be true.

3. **Consider Implementing `operator<` Only**

   In some cases, implementing only the less-than operator (`<`) and the equality operator (`==`) is sufficient because other relational operators can be derived from these two.

4. **Use `const` and Pass Parameters by Reference**

   To avoid modifying the compared objects and to improve efficiency, declare comparison operators as `const` and take arguments as constant references.

5. **Make Comparison Operators Non-Member Functions**

   To support implicit conversions on both operands and improve symmetry, consider implementing comparison operators as non-member functions. Mark them as `friend` inside the class if they require access to private members.

### 10.2.4 Example 1: Overloading Comparison Operators for a `Date` Class

Let's define a simple `Date` class and overload `==`, `!=`, `<`, and `>` operators.

Full runnable code:

```
#include <iostream>

class Date {
private:
    int day, month, year;

public:
```

```cpp
    Date(int d, int m, int y) : day(d), month(m), year(y) {}

    // Equality operator
    bool operator==(const Date& other) const {
        return day == other.day && month == other.month && year == other.year;
    }

    // Inequality operator (negation of ==)
    bool operator!=(const Date& other) const {
        return !(*this == other);
    }

    // Less than operator (compare by year, then month, then day)
    bool operator<(const Date& other) const {
        if (year < other.year) return true;
        if (year > other.year) return false;
        if (month < other.month) return true;
        if (month > other.month) return false;
        return day < other.day;
    }

    // Greater than operator (can be derived from < and ==)
    bool operator>(const Date& other) const {
        return other < *this;
    }

    // Optional: less than or equal to
    bool operator<=(const Date& other) const {
        return !(other < *this);
    }

    // Optional: greater than or equal to
    bool operator>=(const Date& other) const {
        return !(*this < other);
    }
};

int main() {
    Date date1(15, 6, 2023);
    Date date2(20, 6, 2023);
    Date date3(15, 6, 2023);

    if (date1 == date3) {
        std::cout << "date1 is equal to date3\n";
    }

    if (date1 != date2) {
        std::cout << "date1 is not equal to date2\n";
    }

    if (date1 < date2) {
        std::cout << "date1 is earlier than date2\n";
    }

    if (date2 > date1) {
        std::cout << "date2 is later than date1\n";
    }
    return 0;
}
```

**Explanation:**

- `operator==` returns `true` if the day, month, and year are all equal.
- `operator!=` is implemented as the negation of `operator==`.
- `operator<` compares dates by year, then month, then day for a logical ordering.
- Other relational operators are derived from these two, ensuring consistency.

### 10.2.5 Example 2: Overloading Comparison Operators for a `Person` Class

Consider a `Person` class with `name` and `age`. We want to compare two persons by age and check equality by name and age.

Full runnable code:

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

class Person {
private:
    std::string name;
    int age;

public:
    Person(const std::string& n, int a) : name(n), age(a) {}

    // Equality operator compares name and age
    bool operator==(const Person& other) const {
        return name == other.name && age == other.age;
    }

    bool operator!=(const Person& other) const {
        return !(*this == other);
    }

    // Less than compares by age only
    bool operator<(const Person& other) const {
        return age < other.age;
    }

    bool operator>(const Person& other) const {
        return other < *this;
    }
};

int main() {
    Person alice("Alice", 30);
    Person bob("Bob", 25);
    Person aliceClone("Alice", 30);
    if (alice == aliceClone) {
        std::cout << "Alice and her clone are equal\n";
    }

    if (bob != alice) {
        std::cout << "Bob and Alice are not equal\n";
    }

    if (bob < alice) {
        std::cout << "Bob is younger than Alice\n";
    }

    if (alice > bob) {
        std::cout << "Alice is older than Bob\n";
    }

    return 0;
}
```

### 10.2.6 Summary and Key Points

- Overloading comparison operators allows your user-defined types to be compared naturally and intuitively.
- Always overload `==` and `!=` together for consistency.
- Logical ordering operators (`<`, `>`, `<=`, `>=`) should be consistent and ideally derived from a single `operator<`.
- Prefer `const` correctness and pass parameters by reference.
- Non-member functions are preferred for symmetry; use `friend` if access to private members is necessary.
- Well-implemented comparison operators improve your class usability, enabling easy integration with standard algorithms, sorting, and containers.

Mastering comparison operator overloading is essential for writing expressive, maintainable C++ code that behaves predictably in conditional expressions and algorithms. In the next section, we'll explore overloading stream insertion (`<<`) and extraction (`>>`) operators for custom input and output operations.

## 10.3 Overloading Stream Insertion and Extraction Operators

In C++, the stream insertion (`<<`) and extraction (`>>`) operators are fundamental for input and output operations, primarily used with `std::cout` and `std::cin`. These operators allow you to print data to the console or read data from the user. While they work seamlessly with built-in types like integers and strings, to support your **custom classes and objects**, you need to **overload** these operators.

This section explains how to overload `<<` and `>>` operators to enable intuitive input and output for your own classes, making your objects integrate smoothly with C++ streams.

### 10.3.1 Why Overload << and >>?

Imagine you have a class `Point` representing a 2D coordinate with `x` and `y` values. If you need to print a `Point` object directly using:

```
Point p(3, 4);
std::cout << p;
```

You need to define how `operator<<` behaves for the `Point` class because the compiler doesn't know how to convert your object into text by default.

Similarly, for input:

```
Point p;
std::cin >> p;
```

You want the ability to read values into a `Point` object directly.

### 10.3.2 Why Use Non-Member Friend Functions?

The stream insertion (`<<`) and extraction (`>>`) operators are typically overloaded as **non-member** functions rather than member functions of your class. This is because:

- The **left-hand operand** of `<<` and `>>` is a stream object (`std::ostream` or `std::istream`), not your class.
- Overloading them as **non-member friend functions** allows direct access to private members if necessary while keeping the operators symmetric and intuitive.
- This design also allows chaining operations like:
  ```
  std::cout << p1 << " " << p2;
  ```

### 10.3.3 Syntax Overview

Here is the general form of overloading these operators for a class `ClassName`:

```
#include <iostream>

class ClassName {
    // private members

    // Declare friend functions for access if needed
```

```cpp
    friend std::ostream& operator<<(std::ostream& os, const ClassName& obj);
    friend std::istream& operator>>(std::istream& is, ClassName& obj);
};

// Stream insertion operator overload (output)
std::ostream& operator<<(std::ostream& os, const ClassName& obj) {
    // Output object's members to the stream
    return os;
}

// Stream extraction operator overload (input)
std::istream& operator>>(std::istream& is, ClassName& obj) {
    // Extract object's members from the stream
    return is;
}
```

### 10.3.4 Example: Overloading for a `Point` Class

Let's define a simple `Point` class and overload the stream insertion and extraction operators.

Full runnable code:

```cpp
#include <iostream>

class Point {
private:
    int x, y;

public:
    Point() : x(0), y(0) {}  // Default constructor
    Point(int xVal, int yVal) : x(xVal), y(yVal) {}

    // Friend function to overload <<
    friend std::ostream& operator<<(std::ostream& os, const Point& pt);

    // Friend function to overload >>
    friend std::istream& operator>>(std::istream& is, Point& pt);
};

// Output operator: prints point as (x, y)
std::ostream& operator<<(std::ostream& os, const Point& pt) {
    os << "(" << pt.x << ", " << pt.y << ")";
    return os;
}

// Input operator: expects two integers separated by space
std::istream& operator>>(std::istream& is, Point& pt) {
    is >> pt.x >> pt.y;
    return is;
}

int main() {
    Point p1(5, 10);
    std::cout << "Point p1: " << p1 << std::endl;

    Point p2;
    std::cout << "Enter coordinates for point p2 (format: x y): ";
    std::cin >> p2;

    std::cout << "You entered: " << p2 << std::endl;

    return 0;
}
```

### 10.3.5 Explanation

- The `operator<<` function formats the `Point` object as `(x, y)` and inserts it into the output stream.
- The `operator>>` function reads two integers from the input stream and stores them in the `x` and `y` members.
- Both functions return the stream (`os` or `is`) by reference to allow chaining of multiple operations, e.g., `std::cout << p1 << " " << p2;`.

### 10.3.6 Handling Input Errors

When overloading `operator>>`, it's important to consider input validation and error handling.

For example, if the user inputs invalid data (like characters instead of numbers), the stream will enter a fail state, and the extraction will not modify the object.

You can check and clear the stream state outside your operator function or inside it for more robust input handling.

### 10.3.7 Overloading with Complex Classes

For more complex classes, you typically print or read all relevant members in a consistent format. For example, a `Person` class:

Full runnable code:

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

class Person {
```

```cpp
private:
    std::string name;
    int age;

public:
    Person() : name(""), age(0) {}
    Person(std::string n, int a) : name(n), age(a) {}

    friend std::ostream& operator<<(std::ostream& os, const Person& p);
    friend std::istream& operator>>(std::istream& is, Person& p);
};

std::ostream& operator<<(std::ostream& os, const Person& p) {
    os << "Name: " << p.name << ", Age: " << p.age;
    return os;
}

std::istream& operator>>(std::istream& is, Person& p) {
    std::cout << "Enter name: ";
    std::getline(is, p.name);
    std::cout << "Enter age: ";
    is >> p.age;
    is.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Clear newline
    return is;
}

int main() {
    Person person;
    std::cin >> person;
    std::cout << "Person details: " << person << std::endl;
    return 0;
}
```

### 10.3.8 Summary of Key Points

- Overload `operator<<` and `operator>>` as **non-member friend functions** to integrate your classes with C++ streams.
- Always return the stream object by reference to enable chaining of multiple operations.
- `operator<<` is for output (insertion), while `operator>>` is for input (extraction).
- Use consistent formatting in both operators for clarity and predictability.
- Handle input carefully to avoid corrupting your objects when bad data is entered.
- Overloading these operators greatly improves usability and readability when working with custom objects.

Mastering stream operator overloading enables your classes to feel like natural C++ types, simplifying input and output tasks and improving the overall user experience of your programs.

# Chapter 11.

# Inheritance and Polymorphism

# 11 Inheritance and Polymorphism

## 11.1 Base and Derived Classes

Inheritance is a cornerstone of object-oriented programming (OOP) and one of the most powerful features in C++. It allows you to create new classes—called **derived classes**—based on existing classes—called **base classes**. This mechanism promotes **code reuse**, improves organization, and models real-world relationships in a natural way.

In this section, you'll learn what base and derived classes are, how to declare inheritance in C++, how data members and member functions are inherited, and the role of access specifiers in inheritance. We'll also explore practical examples using a simple `Shape` base class and derived classes such as `Circle` and `Rectangle`.

### 11.1.1 What is Inheritance?

Inheritance means that a new class automatically contains the properties (data members) and behaviors (member functions) of another class. The derived class can:

- Use all accessible members of the base class.
- Add new members unique to itself.
- Override or extend the base class's behavior.

Think of it like creating a specialized version of something more general. For example:

- A `Shape` class might represent general properties like position and color.
- `Circle` and `Rectangle` classes inherit from `Shape` and add their own specific details (radius, width, height).

### 11.1.2 Declaring Derived Classes

In C++, inheritance is declared by specifying the base class name after a colon (:) in the derived class definition:

```
class BaseClass {
    // Base class members
};

class DerivedClass : access_specifier BaseClass {
    // Derived class members
};
```

The **access_specifier** controls how the base class's members are inherited by the derived class:

- `public` inheritance: Public and protected members of the base class keep their access levels in the derived class.
- `protected` inheritance: Public and protected members become protected in the derived class.
- `private` inheritance: Public and protected members become private in the derived class.

Most inheritance is **public**, as it models "is-a" relationships clearly.

### 11.1.3 Example: Base Class `Shape`

Let's create a simple base class `Shape`:

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

class Shape {
public:
    std::string color;

    Shape() : color("undefined") {}

    void setColor(const std::string& c) {
        color = c;
    }

    void displayColor() const {
        std::cout << "Color: " << color << std::endl;
    }
};
```

This class has a public data member `color` and two public member functions.

### 11.1.4 Example: Derived Classes `Circle` and `Rectangle`

Now, let's define `Circle` and `Rectangle` classes derived from `Shape`:

```cpp
class Circle : public Shape {
public:
    double radius;

    Circle(double r) : radius(r) {}

    double area() const {
        return 3.14159 * radius * radius;
    }

    void display() const {
        std::cout << "Circle with radius " << radius << " and ";
        displayColor();
    }
};

class Rectangle : public Shape {
public:
    double width, height;

    Rectangle(double w, double h) : width(w), height(h) {}

    double area() const {
        return width * height;
    }

    void display() const {
        std::cout << "Rectangle " << width << "x" << height << " and ";
        displayColor();
    }
};
```

Here, both `Circle` and `Rectangle` inherit `color` and the functions `setColor()` and `displayColor()` from `Shape`. They also add their own specific members and functions.

### 11.1.5 Using the Derived Classes

Here is how you can use these classes:

```
int main() {
    Circle c(5.0);
    c.setColor("Red");
    c.display();
    std::cout << "Area: " << c.area() << std::endl;

    Rectangle r(4.0, 6.0);
    r.setColor("Blue");
    r.display();
    std::cout << "Area: " << r.area() << std::endl;

    return 0;
}
```

Output:

```
Circle with radius 5 and Color: Red
Area: 78.5397
Rectangle 4x6 and Color: Blue
Area: 24
```

### 11.1.6 Key Points About Inheritance

#### Access Specifiers and Inheritance

The choice of access specifier affects what members the derived class inherits and their accessibility:

| Base Class Member | Public Inheritance | Protected Inheritance | Private Inheritance |
|---|---|---|---|
| `public` | `public` | `protected` | `private` |
| `protected` | `protected` | `protected` | `private` |
| `private` | **not inherited** | **not inherited** | **not inherited** |

- **Private members** of the base class are **never accessible** directly in derived classes.
- You can access private members indirectly through public or protected member functions of the base class.

#### Constructors and Inheritance

- Base class constructors are **not inherited** by derived classes.
- The derived class constructor can call the base class constructor explicitly to initialize base members.

Example:

```cpp
class Circle : public Shape {
public:
    double radius;

    Circle(double r, const std::string& c) : Shape() {  // call base constructor
        radius = r;
        setColor(c);
    }
};
```

### 11.1.7 Benefits of Inheritance

- **Code Reuse**: You don't rewrite common code; just add what's unique to each derived class.
- **Logical Organization**: Models real-world hierarchies naturally.
- **Maintainability**: Changes in the base class automatically reflect in derived classes.

### 11.1.8 Summary

- **Inheritance** creates a new class (derived) based on an existing class (base).

- The derived class inherits accessible data members and member functions from the base.
- Use the syntax `class Derived : public Base {}` for public inheritance.
- Access specifiers (`public`, `protected`, `private`) control member accessibility during inheritance.
- Derived classes can add new members and functions, enhancing the base functionality.
- Constructors of the base class are called explicitly or implicitly during derived class instantiation.
- Inheritance promotes **code reuse**, better **organization**, and models hierarchical relationships in your programs.

By understanding base and derived classes, you lay the foundation for more advanced concepts like method overriding, polymorphism, and abstract classes — all of which you will explore in the following sections.

## 11.2 Method Overriding and Virtual Functions

One of the core strengths of object-oriented programming is the ability of derived classes to **customize or extend** the behavior defined in their base classes. This is achieved through **method overriding**, where a derived class provides its own implementation of a function that is already declared in the base class. C++ adds a powerful feature called **virtual functions** to support **dynamic dispatch**, allowing the program to decide at runtime which function to invoke based on the actual object type, rather than the pointer or reference type.

In this section, you will learn how method overriding works in C++, the difference between **static binding** and **dynamic binding**, the role of **virtual functions**, and how to declare and use virtual functions to enable polymorphism.

### 11.2.1 Method Overriding: Specialized Behavior in Derived Classes

When a derived class declares a function with the same name, return type, and parameter list as a base class function, it **overrides** the base class version. This allows the derived class to provide specialized behavior while maintaining a consistent interface.

For example:

```
class Shape {
public:
    void draw() {
        std::cout << "Drawing a generic shape." << std::endl;
    }
};

class Circle : public Shape {
```

```
public:
    void draw() {  // Overrides Shape::draw
        std::cout << "Drawing a circle." << std::endl;
    }
};
```

In this example, `Circle` overrides the `draw()` method of `Shape` to print a more specific message.

### 11.2.2 Static Binding vs. Dynamic Binding

To understand why **virtual functions** are essential, you need to grasp the difference between **static binding** and **dynamic binding**.

- **Static binding** (also called compile-time binding) means the function to call is determined at compile time based on the type of the pointer or reference.
- **Dynamic binding** (also called runtime binding) means the function to call is determined at runtime based on the actual type of the object pointed to.

Consider the following example:

```
Shape* shapePtr = new Circle();
shapePtr->draw();
```

If `draw()` is not declared as `virtual`, the call `shapePtr->draw()` will invoke `Shape::draw()` because the compiler binds the call to the `draw()` function of the pointer type `Shape*`. This is **static binding**.

If `draw()` **is** declared as `virtual` in the base class, then `shapePtr->draw()` will invoke `Circle::draw()`, which is the overridden version in the actual object. This is **dynamic binding**, allowing polymorphic behavior.

### 11.2.3 Declaring Virtual Functions

To enable dynamic binding, you declare a base class function as `virtual` by prefixing its declaration with the `virtual` keyword:

```
class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a generic shape." << std::endl;
    }
};
```

When a function is virtual:

- The compiler creates a **vtable** (virtual method table) for the class.

- Each object stores a pointer to the vtable.
- At runtime, the program uses the vtable pointer to look up the correct function to call based on the actual type of the object.

Derived classes override the virtual function by providing their own definition with the same signature:

```cpp
class Circle : public Shape {
public:
    void draw() override {  // 'override' is optional but recommended
        std::cout << "Drawing a circle." << std::endl;
    }
};
```

Using the `override` keyword is good practice because it tells the compiler you intend to override a base class virtual function, and it will generate an error if no such function exists in the base class.

### 11.2.4 Example: Demonstrating Virtual Functions and Overriding

Here is a complete example demonstrating method overriding and virtual functions:

Full runnable code:

```cpp
#include <iostream>

class Shape {
public:
    virtual void draw() {
        std::cout << "Drawing a generic shape." << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

int main() {
    Shape* shapes[3];
    shapes[0] = new Shape();
    shapes[1] = new Circle();
    shapes[2] = new Rectangle();
    for (int i = 0; i < 3; i++) {
        shapes[i]->draw();  // Calls the appropriate draw() dynamically
    }

    // Clean up
    for (int i = 0; i < 3; i++) {
        delete shapes[i];
    }

    return 0;
}
```

Output:

```
Drawing a generic shape.
Drawing a circle.
Drawing a rectangle.
```

Each call to `draw()` is dynamically dispatched to the correct overridden function, even though the pointers are all of type `Shape*`.

### 11.2.5 What Happens Without `virtual`?

If you remove the `virtual` keyword from the base class's `draw()` function, the output would be:

```
Drawing a generic shape.
Drawing a generic shape.
Drawing a generic shape.
```

This is because static binding occurs, and the compiler calls the `Shape::draw()` version based on the pointer type, ignoring the actual object type.

### 11.2.6 Why Use Virtual Functions?

- **Polymorphism**: Virtual functions enable **runtime polymorphism**, allowing your program to decide which method to call based on the object's actual type, even when accessed through base class pointers or references.
- **Extensibility**: Derived classes can modify or extend behavior without altering the base class interface.
- **Maintainability**: Code can work generically with base class types, yet behave correctly for all derived types.

### 11.2.7 Virtual Destructors

If a class has any virtual functions, it's a good practice to declare the destructor as virtual too:

```
class Shape {
public:
    virtual ~Shape() {
        std::cout << "Shape destructor called." << std::endl;
    }
};
```

This ensures that when a derived object is deleted through a base class pointer, the derived class destructor is called correctly, preventing resource leaks.

### 11.2.8 Summary

- **Method overriding** allows derived classes to provide specialized implementations of base class functions.
- **Virtual functions** enable **dynamic dispatch**, where the function to call is determined at runtime based on the actual object type.
- Without `virtual`, C++ uses **static binding**, which calls functions based on pointer or reference type, not the actual object.
- Declare base class methods as `virtual` to support polymorphism.
- Use the `override` keyword in derived classes to improve code safety and readability.
- Always consider declaring destructors `virtual` when your class has virtual functions.
- Virtual functions enable flexible and extensible code, fundamental to many advanced C++ designs.

Understanding method overriding and virtual functions sets the stage for exploring **abstract classes**, **pure virtual functions**, and the rich world of **polymorphism**, which you will learn about in the next sections.

## 11.3 Abstract Classes and Pure Virtual Functions

In C++, **abstract classes** and **pure virtual functions** provide powerful mechanisms to define interfaces and enforce certain behaviors in derived classes. They are fundamental to designing extensible and flexible software architectures based on inheritance and polymorphism.

In this section, you will learn what abstract classes are, how to declare pure virtual functions, why abstract classes cannot be instantiated, and how derived classes implement these interfaces. Practical examples will help you understand how to use abstract classes effectively.

### 11.3.1 What Are Abstract Classes?

An **abstract class** is a class that **cannot be instantiated directly**. You cannot create objects of an abstract class type because it is incomplete by design. Instead, abstract classes serve as **blueprints** or **interfaces** for derived classes, defining a common set of functions that all subclasses must implement.

Abstract classes typically contain at least one **pure virtual function**, which is a function declared in the base class but left without an implementation. This forces any non-abstract derived class to provide its own implementation of that function.

### 11.3.2 Pure Virtual Functions

A **pure virtual function** is declared by assigning `= 0` in the function declaration inside a class. For example:

```
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
};
```

Here, `draw()` is a pure virtual function, making `Shape` an abstract class. This means:

- `Shape` cannot be instantiated directly: `Shape s;` will cause a compilation error.
- Any concrete class deriving from `Shape` **must** implement `draw()` to become instantiable.
- The pure virtual function serves as a **contract** that derived classes promise to fulfill.

### 11.3.3 Why Use Abstract Classes?

Abstract classes and pure virtual functions are useful when:

- You want to define a common interface for a family of classes.
- You want to enforce that certain functions must be implemented by all subclasses.
- You want to allow polymorphic behavior through base class pointers or references.

They promote **design by contract**, ensuring that derived classes provide specific functionality.

### 11.3.4 Example: Abstract Class with Pure Virtual Function

Let's revisit the example of a `Shape` class hierarchy, this time making `Shape` abstract:

Full runnable code:

```cpp
#include <iostream>

class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function makes Shape abstract

    // Virtual destructor for proper cleanup
    virtual ~Shape() {}
};

class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a rectangle." << std::endl;
    }
};

int main() {
    // Shape s;  // Error: Cannot instantiate abstract class

    Shape* shapes[2];
    shapes[0] = new Circle();
    shapes[1] = new Rectangle();

    for (int i = 0; i < 2; ++i) {
        shapes[i]->draw();  // Calls appropriate overridden method
    }

    for (int i = 0; i < 2; ++i) {
        delete shapes[i];
    }

    return 0;
}
```

Output:

```
Drawing a circle.
Drawing a rectangle.
```

### 11.3.5 Explanation

- The `Shape` class cannot be instantiated because it has the pure virtual function `draw()`.
- `Circle` and `Rectangle` provide their own implementation of `draw()`.
- Polymorphism allows the program to call the appropriate `draw()` method at runtime.
- This design guarantees that all shapes know how to `draw()` themselves, satisfying the

contract defined by the abstract base class.

### 11.3.6 Implementing Multiple Pure Virtual Functions

Abstract classes can have multiple pure virtual functions, requiring derived classes to implement them all. For example:

```
class Animal {
public:
    virtual void speak() = 0;
    virtual void move() = 0;

    virtual ~Animal() {}
};

class Dog : public Animal {
public:
    void speak() override {
        std::cout << "Woof!" << std::endl;
    }
    void move() override {
        std::cout << "Dog runs." << std::endl;
    }
};
```

Any concrete subclass must implement **all** pure virtual functions, or it will also be abstract.

### 11.3.7 Abstract Classes Without Data Members

Often, abstract classes are designed as **interfaces**, containing only pure virtual functions and no data members. This usage mimics the concept of interfaces in other languages, defining a strict API without implementation details.

For example:

```
class Printable {
public:
    virtual void print() const = 0;
    virtual ~Printable() {}
};
```

Any class that inherits `Printable` must implement `print()`, allowing objects of different types to be printed uniformly.

### 11.3.8 Can Abstract Classes Have Implementations?

Yes! Abstract classes can provide implementations for some or all member functions, except pure virtual functions must be overridden. Even pure virtual functions can have a body if needed, but derived classes still must override them to be instantiable.

Example:

```cpp
class Base {
public:
    virtual void f() = 0;  // Pure virtual

    void g() {
        std::cout << "Implemented in base class." << std::endl;
    }
};
```

Derived classes must implement `f()`, but they can inherit or override `g()`.

### 11.3.9 Summary

- **Abstract classes** cannot be instantiated; they are designed to be base classes.
- Declaring a function as **pure virtual** (`= 0`) makes the class abstract.
- Pure virtual functions enforce that derived classes implement specific behavior.
- Abstract classes are essential for defining interfaces and enabling polymorphism.
- Derived classes must implement **all** pure virtual functions to become concrete.
- Abstract classes can contain implementations for non-pure virtual or regular functions.
- Using abstract classes promotes a clean and extensible class hierarchy.

Understanding abstract classes and pure virtual functions equips you to design flexible, reusable, and maintainable code using the full power of C++'s object-oriented features. The next section will build upon this knowledge by exploring **polymorphism** and **dynamic binding** in greater detail.

## 11.4 Polymorphism and Dynamic Binding

Polymorphism is one of the core principles of object-oriented programming (OOP) and plays a crucial role in writing flexible, maintainable, and reusable code. In C++, **polymorphism** allows objects of different derived classes to be treated uniformly through pointers or references to a common base class, while still invoking the correct overridden methods specific to their actual types.

This section explains how polymorphism works in C++, focusing on **dynamic binding** enabled by virtual functions and virtual tables (vtables). We will explore the concept step-by-step, with practical examples illustrating polymorphic behavior.

### 11.4.1 What is Polymorphism?

The word *polymorphism* literally means “many forms.” In programming, it refers to the ability of different types (usually related by inheritance) to be accessed through a common interface, where the correct implementation is selected dynamically at runtime.

In C++, polymorphism primarily occurs when you use **base class pointers or references** to refer to objects of **derived classes**. Thanks to **virtual functions**, the program calls the appropriate overridden function corresponding to the actual object type, not just the type of the pointer or reference.

### 11.4.2 Why is Polymorphism Important?

Polymorphism lets you:

- Write code that works with base class pointers or references without needing to know the exact derived type.
- Extend functionality by adding new derived classes without modifying existing code.
- Implement generic algorithms or data structures (like containers) that operate on a variety of object types uniformly.
- Reduce code duplication by sharing interfaces and leveraging inheritance.

### 11.4.3 Static Binding vs. Dynamic Binding

Before diving deeper, it’s important to understand the difference between **static (compile-time) binding** and **dynamic (run-time) binding**.

- **Static binding**: The compiler determines which function to call based on the **type of the pointer or object at compile time**. This is the default for all non-virtual functions.

- **Dynamic binding**: The function to call is determined **at runtime** based on the **actual type of the object** being pointed to. This happens only with **virtual functions**.

### 11.4.4 Example: Static Binding

Full runnable code:

```cpp
#include <iostream>
#include <string>
class Base {
```

```cpp
public:
    void greet() {
        std::cout << "Hello from Base" << std::endl;
    }
};

class Derived : public Base {
public:
    void greet() {
        std::cout << "Hello from Derived" << std::endl;
    }
};

int main() {
    Base* ptr = new Derived();
    ptr->greet();  // Calls Base::greet(), not Derived::greet()
    delete ptr;
    return 0;
}
```

**Output:**

```
Hello from Base
```

Since `greet()` is not virtual, the function call is bound at compile time to `Base::greet()`, even though `ptr` points to a `Derived` object.

### 11.4.5 Example: Dynamic Binding with Virtual Functions

Full runnable code:

```cpp
#include <iostream>
#include <string>
class Base {
public:
    virtual void greet() {
        std::cout << "Hello from Base" << std::endl;
    }
};

class Derived : public Base {
public:
    void greet() override {
        std::cout << "Hello from Derived" << std::endl;
    }
};

int main() {
    Base* ptr = new Derived();
    ptr->greet();  // Calls Derived::greet() due to dynamic binding
    delete ptr;
    return 0;
}
```

**Output:**

```
Hello from Derived
```

The `virtual` keyword tells the compiler to perform **dynamic binding** using a mechanism called the **virtual table (vtable)**, so the call is dispatched to the correct derived class method at runtime.

### 11.4.6 How Dynamic Binding Works: Virtual Tables (Vtables)

Behind the scenes, when a class declares virtual functions, the compiler generates a **virtual table (vtable)** — a lookup table that stores pointers to the virtual functions of the class.

Each object of a class with virtual functions contains a hidden pointer (called the **vptr**) to its class's vtable. When a virtual function is called through a base class pointer or reference, the program uses the vptr to look up the appropriate function address and calls the overridden function in the derived class.

This mechanism enables polymorphic behavior without requiring the compiler to know the exact type of the object at compile time.

### 11.4.7 Polymorphic Collections Example

Consider a collection of shapes where we want to treat all shapes uniformly but allow each to implement their own `draw()` behavior.

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

class Shape {
public:
    virtual void draw() const = 0;  // Pure virtual, making Shape abstract
    virtual ~Shape() {}
};

class Circle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a Circle" << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() const override {
        std::cout << "Drawing a Rectangle" << std::endl;
    }
};
```

```cpp
int main() {
    std::vector<Shape*> shapes;
    shapes.push_back(new Circle());
    shapes.push_back(new Rectangle());

    for (const auto& shape : shapes) {
        shape->draw();  // Dynamic binding calls correct draw()
    }

    for (auto& shape : shapes) {
        delete shape;  // Clean up memory
    }

    return 0;
}
```

**Output:**

```
Drawing a Circle
Drawing a Rectangle
```

### 11.4.8 Explanation

- The vector holds pointers to `Shape`, but the actual objects are `Circle` and `Rectangle`.
- When `draw()` is called, dynamic binding ensures the correct overridden version is executed.
- This design allows you to add new shapes without changing the code that manages or draws shapes.

### 11.4.9 Polymorphism in Function Calls

Polymorphism can also simplify functions that operate on various derived objects via base class references or pointers.

```cpp
void renderShape(const Shape& shape) {
    shape.draw();  // Calls the correct overridden draw()
}

int main() {
    Circle c;
    Rectangle r;

    renderShape(c);  // Draws Circle
    renderShape(r);  // Draws Rectangle

    return 0;
}
```

This function `renderShape` can handle any `Shape`-derived object, promoting code reuse and extensibility.

### 11.4.10 Summary of Key Points

- **Polymorphism** allows treating objects of different derived classes uniformly through base class pointers or references.
- The magic behind polymorphism is **dynamic binding** enabled by **virtual functions**.
- The **vtable mechanism** helps the program call the correct overridden function at runtime.
- Without virtual functions, C++ uses static binding, and base class function implementations are called regardless of the actual object type.
- Polymorphism simplifies design by enabling **extensible and maintainable** code structures.
- Polymorphic collections and function calls are common practical use cases.
- Always declare destructors as virtual in base classes to ensure proper cleanup of derived objects through base pointers.

### 11.4.11 Best Practices

- Mark base class methods as `virtual` if you expect them to be overridden.
- Use `override` in derived classes for clarity and to catch mistakes.
- Make base class destructors `virtual` if you use polymorphism.
- Avoid slicing by always using pointers or references when working with polymorphic types.

By mastering polymorphism and dynamic binding, you unlock the full potential of C++'s object-oriented features, enabling elegant solutions to complex problems while writing flexible, reusable code. The next chapters will build on these concepts to explore more advanced topics in C++ programming.

# Chapter 12.

# Templates and Generic Programming

1. Function Templates
2. Class Templates
3. Template Specialization
4. Using Standard Template Library (STL) Basics

# 12 Templates and Generic Programming

## 12.1 Function Templates

In C++, **function templates** offer a powerful way to write generic, type-independent functions that can work with any data type. Instead of writing multiple versions of a function for different types, you write a single template, and the compiler generates the appropriate function code based on the types used in each function call. This technique promotes code reuse, reduces redundancy, and makes your programs more flexible.

### 12.1.1 What is a Function Template?

A **function template** is essentially a blueprint for creating functions. You define it using the keyword `template` followed by a template parameter list enclosed in angle brackets (`<>`). The compiler uses this template to generate concrete functions when you call the function with specific types.

### 12.1.2 Syntax of a Function Template

```
template <typename T>
T functionName(T param1, T param2) {
    // function body
}
```

Here, `T` is a **template parameter** representing a placeholder for a data type. When you call the function with particular argument types, the compiler substitutes `T` with the actual type(s) and creates the function.

### 12.1.3 Example 1: Generic Swap Function

One of the simplest and most common examples of a function template is a generic swap function that exchanges the values of two variables.

```
template <typename T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

**Explanation:**

- `template <typename T>` declares that `T` is a placeholder type.
- The function `swapValues` takes two references of type `T`.
- It swaps the values using a temporary variable of type `T`.

### 12.1.4 Using the Function Template

```
int main() {
    int x = 10, y = 20;
    swapValues(x, y);
    std::cout << "x = " << x << ", y = " << y << std::endl;

    double a = 1.5, b = 2.5;
    swapValues(a, b);
    std::cout << "a = " << a << ", b = " << b << std::endl;

    return 0;
}
```

**Output:**

```
x = 20, y = 10
a = 2.5, b = 1.5
```

The same `swapValues` function works for both `int` and `double` without writing separate functions.

### 12.1.5 Template Argument Deduction

The compiler automatically deduces the template parameter types by inspecting the types of the function arguments you pass. This means you usually do not have to specify the template type explicitly.

```
swapValues(5, 10);  // Compiler deduces T as int
swapValues(3.14, 2.71);  // T deduced as double
```

However, if needed, you can explicitly specify the template arguments:

```
swapValues<int>(5, 10);
```

### 12.1.6 Example 2: Generic Max Function

Let's create a function template to return the maximum of two values:

Full runnable code:

```cpp
#include <iostream>
template <typename T>
T maxValue(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    std::cout << maxValue(10, 20) << std::endl;         // Outputs 20
    std::cout << maxValue(3.14, 2.71) << std::endl;     // Outputs 3.14
    std::cout << maxValue('a', 'z') << std::endl;       // Outputs 'z'
    return 0;
}
```

### 12.1.7 Example 3: Generic Sorting Function (Simplified)

Function templates can also be used to write generic algorithms. Here is a simplified example of a bubble sort function template that sorts an array of any type:

Full runnable code:

```cpp
#include <iostream>

template <typename T>
void bubbleSort(T arr[], int size) {
    for (int i = 0; i < size - 1; ++i) {
        for (int j = 0; j < size - i -1; ++j) {
            if (arr[j] > arr[j + 1]) {
                T temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int numbers[] = {5, 3, 8, 6, 2};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    bubbleSort(numbers, size);

    for (int i = 0; i < size; ++i) {
        std::cout << numbers[i] << " ";
    }

    return 0;
}
```

**Output:**

```
2 3 5 6 8
```

This sorting function works with any type `T` that supports the `>` operator.

### 12.1.8 Limitations of Function Templates

While function templates are versatile, there are some limitations and considerations:

- **Type Requirements:** The operations inside the template function must be valid for the template type. For example, in `maxValue`, the `>` operator must be defined for type `T`.
- **Code Bloat:** Each instantiation of the template with a different type generates a separate copy of the function, which can increase the final executable size.
- **Special Behavior Needed:** Sometimes, you want to implement a function template for most types but customize behavior for specific types.

### 12.1.9 Template Specialization

To handle special cases, C++ allows **template specialization** — providing a different implementation of a function template for a particular type.

### 12.1.10 Example: Specialized Swap for `const char*`

The generic swap template swaps objects using assignment. But swapping `const char*` pointers swaps addresses, not the strings they point to. You may want to specialize the swap function for `const char*` to swap the strings instead.

```
template <>
void swapValues<const char*>(const char*& a, const char*& b) {
    char temp[100];
    strcpy(temp, a);
    strcpy((char*)a, b);
    strcpy((char*)b, temp);
}
```

**Note:** This example is simplified and uses `strcpy` for illustration. In practice, `std::string` is preferred for safe string handling.

### 12.1.11 Summary

- **Function templates** enable writing generic functions that work with any data type.
- The syntax uses `template <typename T>` before the function definition.
- Template parameters are usually **deduced automatically** by the compiler based on function arguments.
- Common examples include generic `swap`, `max`, and sorting functions.
- Templates require that the operations used inside are valid for the types substituted.

- **Template specialization** allows you to provide customized implementations for specific types.
- Using function templates can greatly reduce code duplication and increase flexibility.

By mastering function templates, you gain a fundamental tool for generic programming in C++, setting the stage for exploring class templates, STL containers, and more advanced topics covered in this chapter.

## 12.2 Class Templates

In C++, **class templates** extend the power of templates to user-defined types, allowing you to create **generic classes** that can work with any data type. This means you can design a class once and reuse it with different types without rewriting code for each type.

Class templates are fundamental for building flexible, reusable components like containers, smart pointers, or any data structures that need to work with a variety of types. Understanding class templates is a crucial step toward mastering generic programming in C++.

### 12.2.1 What is a Class Template?

A **class template** is a blueprint for creating classes parameterized by one or more types. Instead of specifying a fixed type for members, methods, or parameters, you use **template parameters** as placeholders.

### 12.2.2 Syntax of a Class Template

```cpp
template <typename T>
class ClassName {
    // Members using T
};
```

- The `template <typename T>` line declares a template with a type parameter `T`.
- The class definition uses `T` as a type placeholder.
- When you instantiate the template, you provide the actual type to replace `T`.

### 12.2.3 Example 1: Generic Stack Class

Let's create a simple generic stack class that can store elements of any type:

```cpp
#include <iostream>

template <typename T>
class Stack {
private:
    static const int maxSize = 100;
    T data[maxSize];
    int top;

public:
    Stack() : top(-1) {}

    bool push(const T& item) {
        if (top >= maxSize - 1) {
            std::cout << "Stack overflow\n";
            return false;
        }
        data[++top] = item;
        return true;
    }

    bool pop() {
        if (top < 0) {
            std::cout << "Stack underflow\n";
            return false;
        }
        --top;
        return true;
    }

    T peek() const {
        if (top < 0) {
            throw std::out_of_range("Stack is empty");
        }
        return data[top];
    }

    bool isEmpty() const {
        return top == -1;
    }
};
```

### 12.2.4 Explanation:

- `template <typename T>` declares the class template with type parameter `T`.
- The stack uses a fixed-size array of type `T` to store elements.
- Methods `push`, `pop`, `peek`, and `isEmpty` operate on elements of type `T`.
- The stack can be instantiated with any data type (e.g., `int`, `double`, or user-defined types).

### 12.2.5 Using the Generic Stack

```
int main() {
    Stack<int> intStack;
    intStack.push(10);
    intStack.push(20);
    std::cout << "Top element: " << intStack.peek() << std::endl;

    Stack<std::string> stringStack;
    stringStack.push("Hello");
    stringStack.push("World");
    std::cout << "Top element: " << stringStack.peek() << std::endl;

    return 0;
}
```

**Output:**

```
Top element: 20
Top element: World
```

Here, the same `Stack` class works with both `int` and `std::string` without any changes to the class code.

### 12.2.6 Template Parameters and Multiple Types

Class templates can have multiple type parameters:

```
template <typename T, typename U>
class Pair {
public:
    T first;
    U second;

    Pair(const T& a, const U& b) : first(a), second(b) {}
};
```

Usage:

```
Pair<int, std::string> p(1, "apple");
std::cout << p.first << ", " << p.second << std::endl;
```

### 12.2.7 Template Member Functions

Member functions of a class template can be defined inside or outside the class.

### 12.2.8 Inside the class (inline):

```
template <typename T>
class Box {
    T value;
public:
    void setValue(const T& val) { value = val; }
    T getValue() const { return value; }
};
```

### 12.2.9 Outside the class:

When defining member functions outside the class, you must repeat the template declaration and specify the class as a template:

```
template <typename T>
void Box<T>::setValue(const T& val) {
    value = val;
}

template <typename T>
T Box<T>::getValue() const {
    return value;
}
```

### 12.2.10 Instantiating Class Templates

You create instances (objects) of a class template by specifying the actual type(s) in angle brackets:

```
Stack<int> s1;            // Stack of integers
Stack<double> s2;         // Stack of doubles
Pair<std::string, int> p("Age", 30);  // Pair of string and int
```

The compiler generates the appropriate class definition and member functions for each type used.

### 12.2.11 Requirements for Template Parameters

Types used as template arguments must support operations used in the class:

- For example, if your class uses the assignment operator, the type must be assignable.
- If your class compares elements using `<` or `==`, the type must support these operators.
- This is often called the "concept" or "type requirements" of a template parameter.

If a type doesn't meet the requirements, you'll get compiler errors.

### 12.2.12 Example 2: Generic Array Class

Here's a generic dynamic array class with minimal functionality:

Full runnable code:

```cpp
#include <iostream>

template <typename T>
class Array {
private:
    T* elements;
    int size;

public:
    Array(int s) : size(s) {
        elements = new T[size];
    }

    ~Array() {
        delete[] elements;
    }

    void set(int index, const T& value) {
        if (index >= 0 && index < size) {
            elements[index] = value;
        }
    }

    T get(int index) const {
        if (index >= 0 && index < size) {
            return elements[index];
        }
        throw std::out_of_range("Index out of range");
    }

    int getSize() const {
        return size;
    }
};

int main() {
    Array<int> arr(5);
    for (int i = 0; i < arr.getSize(); ++i) {
        arr.set(i, i * 10);
    }
    for (int i = 0; i < arr.getSize(); ++i) {
        std::cout << arr.get(i) << " ";
    }
    return 0;
}
```

Output:

```
0 10 20 30 40
```

This class can be instantiated with any type that supports assignment and default construction.

### 12.2.13 Summary

- **Class templates** define generic classes parameterized by types using `template <typename T>`.
- They enable writing reusable, type-independent data structures and classes.
- Template parameters can be one or multiple types.
- Member functions can be defined inside or outside the class template.
- When instantiating class templates, you specify the actual type(s) in angle brackets.
- Template arguments must satisfy type requirements based on the operations used inside the class.
- Examples like generic `Stack` and `Array` classes show practical use of class templates for storing and manipulating data of various types.

Class templates are the foundation of generic programming in C++. Mastering them unlocks the power of the Standard Template Library (STL) and helps you write clean, maintainable, and efficient code that adapts to many types.

## 12.3 Template Specialization

Templates in C++ provide a powerful mechanism to write generic, reusable code. However, sometimes you want to **customize** the behavior of a template for specific types or arguments. This is where **template specialization** comes in. Template specialization allows you to define alternate implementations of a template for particular cases, enhancing flexibility, correctness, and efficiency.

### 12.3.1 What is Template Specialization?

Template specialization means providing a **customized version** of a template when certain template arguments meet specific criteria. Instead of using the generic template definition, the compiler selects the specialized version for those arguments.

There are two main types of specialization:

- **Full specialization:** Specialize the template for a specific set of template arguments.

- **Partial specialization:** Specialize the template for a subset or pattern of template arguments.

### 12.3.2 Full Template Specialization

**Full specialization** occurs when you define a completely separate implementation of a template for a specific type or set of types.

### 12.3.3 Syntax

Suppose you have a generic class template:

```
template <typename T>
class Calculator {
public:
    static void info() {
        std::cout << "Generic Calculator\n";
    }
};
```

You can fully specialize it for `int` like this:

```
template <>
class Calculator<int> {
public:
    static void info() {
        std::cout << "Integer Calculator\n";
    }
};
```

Notice the empty angle brackets `<>` after `template`, which indicates a specialization, and the specific type `int` inside the angle brackets after the class name.

### 12.3.4 Example Usage

```
int main() {
    Calculator<double>::info();  // Output: Generic Calculator
    Calculator<int>::info();     // Output: Integer Calculator
    return 0;
}
```

Here, when `Calculator<int>` is instantiated, the specialized version is used, while for `Calculator<double>`, the generic version applies.

### 12.3.5 Partial Template Specialization

Partial specialization lets you specialize templates for a **subset of template arguments** or according to a pattern, rather than fully specifying all arguments.

Partial specialization is only allowed for **class templates** (not function templates).

### 12.3.6 Syntax Example

Suppose you have a template class with two parameters:

```
template <typename T1, typename T2>
class Pair {
public:
    void display() {
        std::cout << "Generic Pair\n";
    }
};
```

You can partially specialize for the case when both template parameters are the **same type**:

```
template <typename T>
class Pair<T, T> {
public:
    void display() {
        std::cout << "Pair with two identical types\n";
    }
};
```

### 12.3.7 Example Usage

```
int main() {
    Pair<int, double> p1;
    p1.display();   // Output: Generic Pair

    Pair<int, int> p2;
    p2.display();   // Output: Pair with two identical types

    return 0;
}
```

In this example, the compiler uses the partial specialization for pairs where both types are the same, while using the generic template otherwise.

### 12.3.8 Specialization for Pointers

Partial specialization can be very useful for pointer types:

```
template <typename T>
class Wrapper {
public:
    void identify() {
        std::cout << "Generic type\n";
    }
};

// Partial specialization for pointer types
template <typename T>
class Wrapper<T*> {
public:
    void identify() {
        std::cout << "Pointer type\n";
    }
};
```

### 12.3.9 Usage

```
Wrapper<int> w1;
w1.identify();    // Output: Generic type

Wrapper<int*> w2;
w2.identify();    // Output: Pointer type
```

Here, `Wrapper<int*>` uses the specialized implementation for pointer types.

### 12.3.10 When to Use Template Specialization?

Template specialization is valuable in many practical scenarios:

1. **Type-specific behavior:** For example, a generic container might need a special implementation for `bool` because storing bits can be optimized.
2. **Optimizations:** Some types may allow faster or more efficient algorithms, so specialized versions can boost performance.
3. **Correctness:** Certain types require unique handling (e.g., deep copy semantics for pointers or resources).
4. **Interfacing with legacy code:** You might specialize templates to work properly with third-party or system types.

### 12.3.11 Example: Specialized `print` Function Template

Consider a generic function template that prints values:

```cpp
template <typename T>
void print(const T& value) {
    std::cout << "Value: " << value << std::endl;
}
```

This works fine for many types, but suppose you want a different format when printing `bool` values:

```cpp
// Full specialization for bool
template <>
void print<bool>(const bool& value) {
    std::cout << "Boolean: " << (value ? "true" : "false") << std::endl;
}
```

Usage:

```cpp
int main() {
    print(42);        // Output: Value: 42
    print(true);      // Output: Boolean: true
    return 0;
}
```

This shows how you can specialize a function template for `bool` without affecting other types.

### 12.3.12 Summary

- **Template specialization** customizes template behavior for particular types or template arguments.
- **Full specialization** completely overrides the template for specific types.
- **Partial specialization** customizes templates for a subset or pattern of arguments, only for class templates.
- Specialization is useful for optimization, correctness, and type-specific behavior.
- Common uses include handling pointer types, integral types, or special cases like `bool`.
- Understanding specialization helps you write flexible and efficient generic code.

Mastering template specialization opens the door to advanced C++ programming, allowing you to build versatile libraries and applications that adapt smoothly to diverse data types.

## 12.4 Using Standard Template Library (STL) Basics

The **Standard Template Library (STL)** is one of C++'s most powerful features, offering a rich collection of generic classes and functions designed to simplify common programming tasks. Built upon templates, the STL provides reusable, efficient, and flexible components that save you from writing boilerplate code, enabling you to focus on solving your problem rather than reinventing data structures and algorithms.

In this section, we will introduce the core parts of the STL: **containers**, **iterators**, and **algorithms**, and show practical examples of how templates empower the STL to work seamlessly with different data types.

### 12.4.1 What is the STL?

The STL is a library that implements **generic programming** concepts. It includes:

- **Containers:** Data structures such as arrays, lists, queues, stacks, maps, and more, implemented as template classes.
- **Iterators:** Objects that provide a standardized way to access elements within containers, abstracting pointer-like behavior.
- **Algorithms:** Template functions that perform operations like searching, sorting, counting, and manipulating container elements.

Because all STL components are template-based, they work uniformly with any data type, whether built-in or user-defined, without rewriting code for each type.

### 12.4.2 Common STL Containers

**`std::vector`**

A **dynamic array** that can resize automatically. It stores elements contiguously in memory, providing fast random access.

Full runnable code:

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

int main() {
    std::vector<int> numbers;  // Vector of integers

    // Adding elements
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Access elements
    for (size_t i = 0; i < numbers.size(); ++i) {
        std::cout << numbers[i] << " ";
    }
    // Output: 10 20 30
    return 0;
}
```

### `std::list`

A **doubly linked list** allowing efficient insertions and deletions at any position but slower random access compared to `vector`.

Full runnable code:

```cpp
#include <list>
#include <iostream>

int main() {
    std::list<std::string> names = {"Alice", "Bob", "Charlie"};

    // Insert element at beginning
    names.push_front("Zara");

    for (const auto& name : names) {
        std::cout << name << " ";
    }
    // Output: Zara Alice Bob Charlie
    return 0;
}
```

### `std::map`

An **associative container** that stores key-value pairs in sorted order, allowing fast lookups by key.

Full runnable code:

```cpp
#include <map>
#include <iostream>

int main() {
    std::map<int, std::string> idToName;

    idToName[101] = "John";
    idToName[102] = "Emily";

    // Accessing by key
    std::cout << idToName[101] << std::endl;  // Output: John

    return 0;
}
```

### Iterators: Accessing Container Elements

Iterators provide a **uniform interface** to traverse elements in any container, similar to pointers.

### Example: Using Iterators with `std::vector`

Full runnable code:

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

int main() {
    std::vector<int> data = {1, 2, 3, 4, 5};

    // Using iterator to traverse vector
    for (std::vector<int>::iterator it = data.begin(); it != data.end(); ++it) {
        std::cout << *it << " ";  // Dereference iterator to get element
    }
    // Output: 1 2 3 4 5
    return 0;
}
```

Modern C++ allows simpler iteration with range-based for loops, which internally use iterators:

```cpp
for (int value : data) {
    std::cout << value << " ";
}
```

### 12.4.3 STL Algorithms: Generic Operations

STL provides many **template functions** for common operations, applicable to any container supporting the required iterator category.

### 12.4.4 Example: Sorting a Vector

Full runnable code:

```cpp
#include <vector>
#include <algorithm>  // For std::sort
#include <iostream>

int main() {
    std::vector<int> nums = {5, 2, 8, 1, 4};

    std::sort(nums.begin(), nums.end());

    for (int n : nums) {
        std::cout << n << " ";
    }
    // Output: 1 2 4 5 8
    return 0;
}
```

### 12.4.5 Example: Finding an Element

Full runnable code:

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

int main() {
    std::vector<std::string> fruits = {"apple", "banana", "cherry"};

    auto it = std::find(fruits.begin(), fruits.end(), "banana");

    if (it != fruits.end()) {
        std::cout << "Found: " << *it << std::endl;
    } else {
        std::cout << "Not found" << std::endl;
    }

    return 0;
}
```

### 12.4.6 How Templates Simplify STL

Thanks to templates, the STL containers and algorithms can:

- Work with **any data type**, including user-defined classes.
- Be **efficient** because templates allow compile-time type resolution, avoiding overhead.
- Promote **code reuse** by providing a single implementation that handles various data types.
- Allow **custom behavior** by letting you define comparison operators or provide function objects for algorithms.

### 12.4.7 Practical Example: Using `std::vector` and Algorithms Together

Let's write a simple program to store student grades, sort them, and display the top grades.

Full runnable code:

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

int main() {
    std::vector<int> grades = {85, 92, 78, 90, 88};

    // Sort grades in descending order using a lambda function
    std::sort(grades.begin(), grades.end(), [](int a, int b) {
        return a > b;
    });

    std::cout << "Top grades: ";
    for (int grade : grades) {
        std::cout << grade << " ";
    }
    // Output: Top grades: 92 90 88 85 78
    return 0;
}
```

This example demonstrates how containers, iterators, and algorithms work together to accomplish a common task with minimal code.

### 12.4.8 Summary

- The STL is a powerful **generic programming** library built with templates.
- It provides **containers** like `vector`, `list`, and `map` to store data flexibly.
- **Iterators** allow uniform traversal of container elements, abstracting pointer-like operations.
- **Algorithms** are template functions performing common tasks like sorting, searching, and modifying collections.
- Templates enable the STL to be **type-independent**, efficient, and easy to use.
- Mastering STL basics helps you write concise, robust C++ code with less effort.

As you continue learning, exploring the full STL will significantly boost your productivity and deepen your understanding of C++'s powerful capabilities.

# Chapter 13.

# Exception Handling

1. Basics of Exceptions (`try`, `catch`, `throw`)
2. Creating Custom Exception Classes
3. Exception Safety and Best Practices

# 13 Exception Handling

## 13.1 Basics of Exceptions (`try`, `catch`, `throw`)

In programming, errors and unexpected situations can occur at runtime — for example, trying to open a file that doesn't exist, dividing by zero, or running out of memory. **Exception handling** in C++ is a mechanism designed to detect and respond to such errors gracefully, preventing your program from crashing abruptly and enabling you to maintain control flow even in the face of problems.

This section introduces the core concepts of exception handling in C++, including the keywords `try`, `catch`, and `throw`, how exceptions propagate, and how to catch different types of exceptions. We'll provide simple examples to illustrate these concepts and help you write safer, more robust C++ programs.

### 13.1.1 What is Exception Handling?

Exception handling allows you to **detect errors**, **throw exceptions** when something goes wrong, and **catch those exceptions** to take appropriate action. Instead of letting errors silently cause unpredictable behavior or program termination, exceptions give you a structured way to signal and handle problems.

### 13.1.2 The Basic Syntax

C++ uses three main keywords for exceptions:

- **`try`**: Defines a block of code to monitor for exceptions.
- **`throw`**: Used to raise (throw) an exception when an error occurs.
- **`catch`**: Defines a block of code that handles exceptions of specific types.

### 13.1.3 The `try` Block

The `try` block contains the code that might generate an exception. If everything runs smoothly inside this block, the program continues as usual. But if an exception is thrown, the rest of the `try` block is skipped, and the control jumps to the matching `catch` block.

```
try {
    // Code that might throw an exception
}
```

### 13.1.4 The `throw` Statement

When your program detects an error or an exceptional condition, it **throws** an exception object using the `throw` keyword, often followed by a value or an object describing the error.

```
throw 42;             // Throwing an int
throw std::runtime_error("Error occurred");  // Throwing an object
```

Throwing an exception immediately transfers control out of the current scope to the nearest matching `catch` block.

### 13.1.5 The `catch` Block

The `catch` block handles exceptions thrown in the preceding `try` block. You can write multiple `catch` blocks to handle different exception types.

```
catch (int e) {
    // Handle exceptions of type int
}
catch (const std::exception& e) {
    // Handle exceptions derived from std::exception
}
catch (...) {
    // Catch all exceptions (catch-all handler)
}
```

The ellipsis `...` in `catch(...)` means catching any exception regardless of type.

### 13.1.6 Exception Flow and Propagation

When an exception is thrown, C++ looks for the nearest enclosing `try` block and jumps to the matching `catch` block. If no matching `catch` is found in the current function, the exception propagates up the call stack to the caller. If it reaches the top of the stack without being caught, the program terminates.

### 13.1.7 Simple Example: Throwing and Catching an Exception

Let's look at a simple example that demonstrates throwing and catching an exception.

Full runnable code:

```
#include <iostream>

int divide(int numerator, int denominator) {
    if (denominator == 0) {
        throw "Division by zero error";  // Throw a string literal
    }
    return numerator / denominator;
}

int main() {
    try {
        int result = divide(10, 0);  // This will throw an exception
        std::cout << "Result: " << result << std::endl;
    }
    catch (const char* msg) {  // Catch the string literal exception
        std::cerr << "Caught exception: " << msg << std::endl;
    }

    std::cout << "Program continues after exception handling." << std::endl;

    return 0;
}
```

**Explanation:**

- The function `divide` throws a string literal exception if the denominator is zero.
- The `try` block in `main` calls `divide`. When the exception is thrown, control jumps to the matching `catch`.
- The `catch` block prints an error message.
- The program continues running after handling the exception.

### 13.1.8 Catching Different Types of Exceptions

You can throw and catch different types of exceptions, such as integers, strings, or user-defined objects.

Full runnable code:

```
#include <iostream>
#include <stdexcept>  // For std::runtime_error

void testFunction(int x) {
    if (x < 0) {
        throw std::runtime_error("Negative value error");
    }
    else if (x == 0) {
        throw 0;  // Throw int
    }
    else {
        throw "Unknown error";  // Throw string literal
    }
}

int main() {
    try {
        testFunction(-1);
    }
    catch (const std::runtime_error& e) {
        std::cerr << "Runtime error: " << e.what() << std::endl;
    }
    catch (int e) {
        std::cerr << "Integer exception: " << e << std::endl;
    }
    catch (const char* msg) {
        std::cerr << "String exception: " << msg << std::endl;
    }

    return 0;
}
```

**Output:**

```
Runtime error: Negative value error
```

Here, the exception thrown is caught by the most specific matching `catch` block.

### 13.1.9 Using Standard Exception Classes

The C++ Standard Library provides a hierarchy of exception classes derived from `std::exception`. It's good practice to throw exceptions derived from `std::exception` for better consistency and richer information.

Example:

Full runnable code:

```
#include <iostream>
#include <stdexcept>

int divide(int a, int b) {
    if (b == 0) {
        throw std::invalid_argument("Division by zero");
    }
    return a / b;
}

int main() {
    try {
        int res = divide(10, 0);
    }
    catch (const std::invalid_argument& e) {
        std::cerr << "Invalid argument exception: " << e.what() << std::endl;
    }
    return 0;
}
```

The `what()` method provides an error message describing the exception.

### 13.1.10 Summary and Best Practices

- Use `try` blocks to wrap code that might throw exceptions.
- Use `throw` to signal an error by throwing an exception object.
- Use one or more `catch` blocks to handle exceptions of specific types.
- Prefer throwing exceptions derived from `std::exception` for better compatibility.
- Catch exceptions by reference (usually `const&`) to avoid unnecessary copying and slicing.
- Always design your program so it can recover or exit cleanly after an exception.
- Avoid using exceptions for regular control flow; reserve them for truly exceptional conditions.

### 13.1.11 Final Thoughts

Exception handling in C++ provides a structured way to deal with errors and exceptional situations, helping your program maintain stability and correctness. Understanding how to throw, catch, and propagate exceptions is essential for writing robust applications that can gracefully handle runtime issues.

In the next sections, we will explore how to create custom exception classes and write exception-safe code that adheres to best practices.

## 13.2 Creating Custom Exception Classes

In C++, the standard library provides several exception classes like `std::runtime_error`, `std::invalid_argument`, and more, which cover many common error situations. However, in real-world applications, you often encounter specific error conditions unique to your program's logic or domain. To represent such errors clearly and make your code more expressive and maintainable, **creating custom exception classes** is highly recommended.

This section explains why and how to define custom exception classes in C++, focusing on inheriting from `std::exception`, overriding the `what()` method, and providing practical examples to make your error handling precise and meaningful.

### 13.2.1 Why Create Custom Exception Classes?

Here are some important reasons to create your own exception classes:

- **Clearer Error Identification**: Instead of catching generic exceptions, you can catch specific exception types to handle different error scenarios differently.

- **Better Debugging**: Custom exceptions allow you to provide detailed and descriptive error messages or additional information about the error.
- **Improved Code Organization**: Defining your exceptions groups related error handling code and separates concerns between normal logic and error management.
- **Extensibility**: You can add extra data members or methods in your custom exception class to convey more error context.

### 13.2.2 Inheriting from `std::exception`

C++ encourages you to inherit from the base class `std::exception` when creating custom exceptions. This practice ensures compatibility with standard exception handling, and you get access to the `what()` method, which returns a C-style string describing the error.

### 13.2.3 Syntax for a Basic Custom Exception

```cpp
#include <exception>
#include <string>

class MyException : public std::exception {
private:
    std::string message;

public:
    // Constructor accepting error message
    explicit MyException(const std::string& msg) : message(msg) {}

    // Override what() method to return error message
    const char* what() const noexcept override {
        return message.c_str();
    }
};
```

### 13.2.4 Key Points:

- The `what()` method must return a `const char*` and be marked as `noexcept` (meaning it does not throw exceptions).
- The exception class usually stores the error message internally (here, in a `std::string`).
- The constructor takes the error message as an argument and initializes the internal string.

### 13.2.5 Using Custom Exception Classes

Here's a practical example demonstrating how to throw and catch a custom exception:

Full runnable code:

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

class FileOpenException : public std::exception {
private:
    std::string message;

public:
    explicit FileOpenException(const std::string& filename)
        : message("Failed to open file: " + filename) {}

    const char* what() const noexcept override {
        return message.c_str();
    }
};

void openFile(const std::string& filename) {
    bool fileOpened = false; // Simulate failure
    if (!fileOpened) {
        throw FileOpenException(filename);
    }
    // File processing logic...
}

int main() {
    try {
        openFile("data.txt");
    }
    catch (const FileOpenException& e) {
        std::cerr << "Custom exception caught: " << e.what() << std::endl;
    }
    catch (const std::exception& e) {
        std::cerr << "Standard exception caught: " << e.what() << std::endl;
    }
    return 0;
}
```

**Explanation:**

- `FileOpenException` is a custom exception class that holds the filename that failed to open.
- When `openFile` fails to open a file, it throws a `FileOpenException`.
- The `main` function catches this specific exception and prints a meaningful error message.
- If another standard exception occurs, it's caught by the generic `std::exception` handler.

### 13.2.6 Extending Custom Exceptions with Additional Data

Sometimes, you might want your exception to carry more than just a string message, such as an error code, the function name, or other context.

Example:

```cpp
class CalculationException : public std::exception {
private:
    int errorCode;
    std::string message;

public:
    CalculationException(int code, const std::string& msg)
        : errorCode(code), message(msg) {}

    const char* what() const noexcept override {
        return message.c_str();
    }

    int code() const noexcept {
        return errorCode;
    }
};
```

You can then use it as:

```cpp
try {
    throw CalculationException(404, "Invalid calculation input");
}
catch (const CalculationException& e) {
    std::cerr << "Calculation error " << e.code() << ": " << e.what() << std::endl;
}
```

This approach allows you to provide richer information for error handling and debugging.

### 13.2.7 Best Practices for Custom Exceptions

- **Inherit from `std::exception`**: This ensures compatibility and lets users catch all standard exceptions uniformly.
- **Override `what()`**: Provide a meaningful description that explains the error.
- **Make exception classes lightweight**: Avoid adding complex logic or dependencies inside exception classes.
- **Use `noexcept` for `what()`**: Ensures that exception handling itself does not throw exceptions.
- **Use descriptive names**: Choose clear, specific names for your exceptions to reflect the error context.
- **Throw by value, catch by reference**: Throw exceptions as objects (by value) and catch them by const reference to avoid slicing and unnecessary copying.

### 13.2.8 When to Create Custom Exceptions?

You don't need to create a custom exception for every error. Use them when:

- The standard exceptions don't adequately describe the error.
- You want to enable calling code to distinguish between error types and handle them differently.
- You want to attach additional context or data with the error.
- Your application domain has specific failure conditions requiring unique error types.

### 13.2.9 Summary

Custom exception classes empower you to design expressive, maintainable error handling tailored to your application's needs. By inheriting from `std::exception` and overriding the `what()` method, you create exceptions that integrate seamlessly with C++'s exception handling system while giving clear and precise error messages.

Using custom exceptions improves debugging, helps separate error-handling logic from normal code flow, and ultimately leads to more robust software.

In the next section, we'll cover **Exception Safety and Best Practices** — essential techniques to ensure your programs remain stable and leak-free even when exceptions occur.

## 13.3 Exception Safety and Best Practices

Exception handling is a powerful tool in C++ that helps your programs gracefully recover from unexpected errors. However, just adding `try` and `catch` blocks isn't enough. Writing **exception-safe code** ensures that your program behaves correctly even when exceptions occur, avoiding resource leaks, inconsistent states, or crashes.

In this section, we introduce the important **levels of exception safety guarantees**, discuss how to manage resources safely with the **RAII (Resource Acquisition Is Initialization)** pattern, and share practical best practices to write robust and maintainable code.

### 13.3.1 Exception Safety Levels

Exception safety is about ensuring your program remains in a well-defined state, no matter where an exception occurs. There are three commonly accepted levels of exception safety guarantees:

### 13.3.2 Basic Guarantee

- **What it means:** If an exception is thrown, the program remains in a valid state, with no resource leaks or corrupted data.
- **What it does not guarantee:** The operation may have only partially completed; some changes might be visible, but the overall data structures stay consistent.
- **Example:** If a function modifies a data structure but throws midway, the structure won't be corrupted but may reflect partial changes.

### 13.3.3 Strong Guarantee

- **What it means:** Either the operation completes fully, or nothing changes—like a transaction that is all-or-nothing.
- **Rollback Behavior:** If an exception is thrown, the program's state is rolled back to before the operation started.
- **Example:** In a container insertion, if the insertion fails due to memory issues, the container remains unchanged.

### 13.3.4 No-Throw Guarantee (Nothrow)

- **What it means:** The function is guaranteed never to throw an exception.
- **Use cases:** Destructors, swap functions, and critical operations where throwing exceptions would be dangerous.
- **Declared with:** `noexcept` specifier in C++11 and later.

### 13.3.5 Managing Resources Safely: RAII (Resource Acquisition Is Initialization)

One of the biggest challenges with exceptions is ensuring that resources—memory, file handles, network sockets, locks—are properly released if an exception interrupts the normal flow.

**RAII** is a core C++ idiom that solves this elegantly:

- **Concept:** Wrap each resource in an object whose constructor acquires the resource, and whose destructor releases it.
- Because destructors are always called when an object goes out of scope (even if an exception is thrown), resources are safely cleaned up automatically.

### 13.3.6 Example: Managing a File Handle with RAII

```cpp
#include <fstream>
#include <iostream>

void writeFile(const std::string& filename) {
    std::ofstream file(filename); // File opened in constructor

    if (!file.is_open()) {
        throw std::runtime_error("Failed to open file");
    }

    file << "Hello, RAII!"; // Writing data

    // No need to explicitly close; file closes in ofstream destructor,
    // even if an exception occurs later
}
```

Here, `std::ofstream` manages the file resource and closes it safely, no matter what happens.

### 13.3.7 Best Practices for Writing Exception-Safe Code

#### Use RAII Everywhere Possible

Wrap **every resource** in a class that manages acquisition and release:

- Memory → use smart pointers (`std::unique_ptr`, `std::shared_ptr`)
- Files → use file stream objects
- Locks → use `std::lock_guard` or `std::unique_lock`

This eliminates manual cleanup and avoids resource leaks.

#### Write Exception-Safe Functions

- **Separate resource allocation from processing.**
- Use **copy-and-swap idiom** for assignment operators to ensure strong exception safety.
- Avoid raw pointers or manual resource management inside critical code.

Example of copy-and-swap for strong exception safety:

```cpp
class MyClass {
    int* data;
public:
    MyClass(int size) : data(new int[size]) {}

    // Copy constructor
    MyClass(const MyClass& other) : data(new int[/* size */]) {
        // Copy data...
    }

    // Assignment operator using copy-and-swap
    MyClass& operator=(MyClass other) {
        std::swap(data, other.data);
        return *this;
    }

    ~MyClass() {
        delete[] data;
    }
};
```

This approach ensures the object is either fully updated or unchanged if exceptions occur during copying.

### Avoid Throwing Exceptions from Destructors

Destructors should never throw exceptions because if an exception is already active, throwing another leads to program termination.

- Use `noexcept` to explicitly declare non-throwing destructors.
- Handle cleanup carefully and avoid operations that may throw.

### Use `noexcept` Where Appropriate

Mark functions `noexcept` if they are guaranteed not to throw exceptions (e.g., move constructors, destructors).

This allows optimizations and better exception handling by the compiler.

### Catch Exceptions at Appropriate Boundaries

- Catch exceptions at the boundaries of modules or layers where it makes sense to recover or clean up.
- Avoid catching exceptions too early (which can hide errors) or too late (which can cause crashes).

## 13.3.8 Practical Example: Exception Safety in Action

Consider a function that modifies two related resources—an array and a file. To provide **strong exception safety**, either both modifications succeed, or none do:

```
#include <iostream>
#include <fstream>
#include <vector>

void updateData(std::vector<int>& data, const std::string& filename, int newValue) {
    std::vector<int> backup = data;  // Backup copy for rollback

    try {
        // Modify data
        data.push_back(newValue);

        // Open file
        std::ofstream file(filename);
        if (!file) throw std::runtime_error("Cannot open file");

        file << newValue << std::endl;  // Write data

        // File closed automatically by RAII

    } catch (...) {
        data = backup;  // Rollback data
        throw;          // Rethrow exception
    }
}
```

This example ensures:

- If file opening or writing fails, the vector is restored to its original state.
- No partial modifications remain.
- RAII manages the file resource.

### 13.3.9 Summary

- **Exception safety** is crucial to writing robust C++ programs.
- Understand and apply **basic**, **strong**, and **no-throw guarantees**.
- Use **RAII** to automatically manage resources and avoid leaks.
- Write functions that either complete successfully or leave the program state unchanged.
- Mark destructors and critical functions `noexcept` to prevent unexpected terminations.
- Catch exceptions thoughtfully at well-defined program boundaries.
- Avoid throwing exceptions from destructors.
- Use modern C++ tools (smart pointers, standard library classes) to help manage exceptions safely.

Mastering exception safety transforms your code from fragile to resilient, enabling your programs to handle errors gracefully without compromising stability or leaking resources.

In the next chapter, we'll explore **File Handling and Streams**, building on your knowledge of exceptions and resource management.

# Chapter 14.

# Standard Template Library (STL)

1. Containers: `vector`, `list`, `deque`, `map`, `set`
2. Iterators and Algorithms
3. Functors and Lambda Expressions
4. String and Stream Classes

# 14 Standard Template Library (STL)

## 14.1 Containers: `vector`, `list`, `deque`, `map`, `set`

The Standard Template Library (STL) is a cornerstone of modern C++ programming, offering powerful, generic data structures called **containers** that allow you to store and organize collections of data efficiently. Understanding the core STL containers, their characteristics, and use cases is essential for writing clean, efficient, and maintainable code.

This section introduces the most commonly used STL containers:

- Sequence Containers: `vector`, `list`, `deque`
- Associative Containers: `map`, `set`

We'll explain their differences and provide examples illustrating how to declare, insert into, access, and traverse these containers.

### 14.1.1 Sequence Containers

Sequence containers store elements in a linear order. The main sequence containers are:

- **`vector`**
- **`list`**
- **`deque`**

Each offers different performance characteristics for inserting, deleting, and accessing elements.

### 14.1.2 `vector`

- **Description:** A dynamic array that stores elements contiguously in memory.
- **Use case:** Ideal for fast random access and situations where elements are primarily added or removed at the end.
- **Performance:**
  - Random access: O(1)
  - Insertion/removal at end: Amortized O(1)
  - Insertion/removal in middle or front: O(n) because elements must be shifted

**Example:**

Full runnable code:

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

int main() {
    std::vector<int> numbers;       // Declare an empty vector of integers

    // Insert elements
    numbers.push_back(10);
    numbers.push_back(20);
    numbers.push_back(30);

    // Access elements by index
    std::cout << "First element: " << numbers[0] << "\n";

    // Traverse using range-based for loop
    std::cout << "All elements: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    return 0;
}
```

### 14.1.3 `list`

- **Description:** A doubly-linked list storing elements non-contiguously.
- **Use case:** When frequent insertions and removals occur anywhere in the sequence.
- **Performance:**
  - Insertion/removal anywhere: O(1) if iterator is known
  - Random access: O(n) (no direct indexing)

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <list>

int main() {
    std::list<std::string> names = {"Alice", "Bob", "Charlie"};

    // Insert at front
    names.push_front("Zara");

    // Insert at back
    names.push_back("David");

    // Traverse using iterator
    std::cout << "Names: ";
    for (auto it = names.begin(); it != names.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << "\n";

    return 0;
}
```

### 14.1.4 `deque` (Double-Ended Queue)

- **Description:** A sequence container that supports fast insertion/removal at both front and back.
- **Use case:** When you need dynamic arrays but also efficient insertions/removals at both ends.
- **Performance:**
  - Random access: O(1)
  - Insertion/removal at front or back: O(1)
  - Insertion/removal in middle: O(n)

**Example:**

Full runnable code:

```
#include <iostream>
#include <deque>

int main() {
    std::deque<int> dq;

    dq.push_back(100);
    dq.push_front(50);

    std::cout << "Deque elements: ";
    for (int val : dq) {
        std::cout << val << " ";
    }
    std::cout << "\n";

    return 0;
}
```

### 14.1.5 Associative Containers

Associative containers organize data using keys for efficient retrieval based on those keys. The most common are:

- `map`
- `set`

These containers use balanced binary search trees internally, providing efficient lookup, insertion, and removal.

### 14.1.6 `map`

- **Description:** A collection of key-value pairs (associative array), where each key is unique.
- **Use case:** When you need to associate data with unique keys and look up values efficiently.
- **Performance:**
  - Lookup, insertion, removal: O(log n)

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> ageMap;

    // Insert key-value pairs
    ageMap["Alice"] = 30;
    ageMap["Bob"] = 25;
    ageMap["Charlie"] = 35;

    // Access value by key
    std::cout << "Alice's age: " << ageMap["Alice"] << "\n";

    // Traverse map
    std::cout << "All entries:\n";
    for (const auto& pair : ageMap) {
        std::cout << pair.first << ": " << pair.second << "\n";
    }

    return 0;
}
```

### 14.1.7 set

- **Description:** A collection of unique keys, sorted by default.
- **Use case:** When you need to store unique items and quickly check for membership.
- **Performance:**
  - Insertion, removal, lookup: O(log n)

**Example:**

Full runnable code:

```cpp
#include <iostream>
#include <set>

int main() {
    std::set<int> numbers = {3, 1, 4, 1, 5};

    // Insert element
    numbers.insert(2);

    // Attempt to insert duplicate
    numbers.insert(3);  // Ignored because 3 is already present

    // Traverse set (sorted order)
    std::cout << "Numbers: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    // Check membership
    if (numbers.find(4) != numbers.end()) {
        std::cout << "4 is in the set\n";
    }

    return 0;
}
```

### 14.1.8 Key Differences Between Containers

| Container | Memory Layout | Access Time | Insert/Remove Efficiency | Use Case |
|---|---|---|---|---|
| `vector` | Contiguous | O(1) random access | Fast insert/remove at end, slow at front/middle | Dynamic arrays, fast access |

| Container | Memory Layout | Access Time | Insert/Remove Efficiency | Use Case |
|---|---|---|---|---|
| `list` | Non-contiguous (linked list) | O(n) random access | Fast insert/remove anywhere (if iterator known) | Frequent insertion/removal |
| `deque` | Multiple contiguous blocks | O(1) random access | Fast insert/remove at front and back | Double-ended queue operations |
| `map` | Balanced binary tree | O(log n) | O(log n) insert/remove | Key-value storage with sorted keys |
| `set` | Balanced binary tree | O(log n) | O(log n) insert/remove | Unique elements, sorted collection |

### 14.1.9 Summary

- **Sequence containers** store data in linear order with different trade-offs between access speed and insertion/removal efficiency.
- **Associative containers** store unique keys and provide fast lookup, insertion, and removal using tree-based data structures.
- Choosing the right container depends on your application's requirements for access speed, insertion/removal frequency, and data organization.
- STL containers come with a rich set of member functions that make common operations intuitive and efficient.

Mastering these containers will help you write clear, efficient, and scalable C++ programs by leveraging the power of the STL. In the next section, we'll explore **Iterators and Algorithms**, the tools that allow you to traverse and manipulate containers in a generic and powerful way.

## 14.2 Iterators and Algorithms

The Standard Template Library (STL) is built on two fundamental concepts that work hand-in-hand to provide powerful, reusable, and efficient code: **iterators** and **algorithms**. Together, they allow you to traverse, access, and manipulate data stored in STL containers in a generic way.

This section introduces **iterators** as generalized pointers used to navigate container elements, explains different iterator categories, and showcases how common STL **algorithms** leverage iterators to perform operations like searching, sorting, and applying functions.

### 14.2.1 What Are Iterators?

Think of **iterators** as generalized pointers that point to elements inside containers like `vector`, `list`, or `map`. They provide a standardized way to move through the elements regardless of the container type.

Unlike raw pointers, iterators work with all STL containers uniformly, allowing algorithms to operate generically.

### 14.2.2 Why Use Iterators?

- **Abstraction:** Iterators abstract away container details.
- **Generality:** Algorithms work on any container supporting the required iterator type.
- **Safety:** Iterator interfaces prevent invalid memory access and enable safe traversal.

### 14.2.3 Iterator Categories

Not all iterators are created equal. STL defines five main iterator categories that describe their capabilities:

| Iterator Category | Capabilities | Typical Containers |
|---|---|---|
| **Input Iterator** | Read elements sequentially | Streams, single-pass input |
| **Output Iterator** | Write elements sequentially | Streams, output iterators |
| **Forward Iterator** | Read/write, multi-pass, increment only | `forward_list`, `unordered_map` |
| **Bidirectional Iterator** | Forward + backward movement | `list`, `set`, `map` |
| **Random Access Iterator** | All above + direct access by offset | `vector`, `deque`, native pointers |

### 14.2.4 Summary of Key Operations per Category

| Operation | Input | Output | Forward | Bidirectional | Random Access |
|---|---|---|---|---|---|
| Read `*it` | Yes | No | Yes | Yes | Yes |
| Write `*it` | No | Yes | Yes | Yes | Yes |
| Increment `++it` | Yes | Yes | Yes | Yes | Yes |

| Operation | Input | Out-put | Forward | Bidirectional | Random Access |
|---|---|---|---|---|---|
| Decrement `--it` | No | No | No | Yes | Yes |
| Random access (`it + n`) | No | No | No | No | Yes |

### 14.2.5 Using Iterators: Basic Syntax

Iterators behave much like pointers:

```cpp
std::vector<int> vec = {10, 20, 30};
auto it = vec.begin();  // Get iterator to first element

std::cout << *it << "\n"; // Dereference to access element (outputs 10)

++it;                     // Move iterator to next element
std::cout << *it << "\n"; // Outputs 20
```

You can get iterators using:

- `.begin()` — points to the first element
- `.end()` — points just past the last element (not dereferenceable)

When looping over containers, the standard pattern is:

```cpp
for (auto it = container.begin(); it != container.end(); ++it) {
    // Access *it
}
```

### 14.2.6 STL Algorithms: Operating on Iterator Ranges

The true power of iterators is unlocked by STL **algorithms**. These are template functions that operate on ranges defined by pairs of iterators: `[first, last)`.

For example, many algorithms expect two iterators specifying the start and end positions, allowing them to work generically on any container or sequence that supports those iterators.

Let's explore some of the most common algorithms.

### 14.2.7 `std::find`

Searches for a value in a range and returns an iterator to it or the end iterator if not found.

Full runnable code:

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::find

int main() {
    std::vector<int> numbers = {1, 3, 5, 7, 9};
    int target = 5;

    auto it = std::find(numbers.begin(), numbers.end(), target);

    if (it != numbers.end()) {
        std::cout << "Found " << *it << " at index " << (it - numbers.begin()) << "\n";
    } else {
        std::cout << target << " not found\n";
    }

    return 0;
}
```

### 14.2.8 `std::sort`

Sorts elements in a range. Requires **random access iterators** like those from `vector` or `deque`.

Full runnable code:

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::sort

int main() {
    std::vector<int> nums = {5, 2, 9, 1, 7};

    std::sort(nums.begin(), nums.end());

    std::cout << "Sorted numbers: ";
    for (int n : nums) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    return 0;
}
```

### 14.2.9 `std::for_each`

Applies a function to every element in a range.

Full runnable code:

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::for_each

void print(int n) {
    std::cout << n << " ";
}

int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5};

    std::for_each(nums.begin(), nums.end(), print);
    std::cout << "\n";

    return 0;
}
```

You can also use lambdas with `std::for_each`:

```cpp
std::for_each(nums.begin(), nums.end(), [](int n) {
    std::cout << n * 2 << " ";
});
```

### 14.2.10 `std::copy`

Copies elements from one range to another, useful for containers and raw arrays.

Full runnable code:

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // for std::copy

int main() {
    std::vector<int> source = {10, 20, 30};
    std::vector<int> dest(source.size());

    std::copy(source.begin(), source.end(), dest.begin());

    for (int n : dest) {
        std::cout << n << " ";
    }
    std::cout << "\n";

    return 0;
}
```

### 14.2.11 Practical Example: Combining Iterators and Algorithms

Suppose we want to find all even numbers in a list and print them.

Full runnable code:

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

int main() {
    std::vector<int> data = {3, 4, 7, 8, 10, 13};

    // Use std::copy_if and ostream_iterator to print even numbers
    std::cout << "Even numbers: ";
    std::copy_if(data.begin(), data.end(),
                 std::ostream_iterator<int>(std::cout, " "),
                 [](int x) { return x % 2 == 0; });

    std::cout << "\n";

    return 0;
}
```

This example combines:

- Iterators (`data.begin()`, `data.end()`)
- Algorithms (`copy_if`)
- Lambda expression (predicate to check if number is even)
- Output iterator (`ostream_iterator`)

### 14.2.12 Summary and Best Practices

- **Iterators** are the bridge between containers and algorithms, providing a uniform way to access container elements.
- Different **iterator categories** provide different capabilities; understanding them helps choose appropriate algorithms and containers.
- STL **algorithms** like `sort`, `find`, and `for_each` operate on iterator ranges, enabling code reuse and abstraction.
- Use **range-based for loops** for simple iteration, but leverage algorithms for more complex tasks.
- Combining iterators and algorithms with **lambda expressions** leads to concise, expressive code.

Mastering iterators and algorithms will greatly increase your productivity and code quality in C++. The next chapter section will explore **Functors and Lambda Expressions**, which help customize algorithm behavior further and enhance functional programming styles in C++.

## 14.3 Functors and Lambda Expressions

In C++, the Standard Template Library (STL) offers a rich collection of algorithms that work with containers via iterators. To customize the behavior of these algorithms, such as specifying sorting criteria or filtering conditions, you need to supply callable objects. Two powerful tools for this are **functors** (function objects) and **lambda expressions**.

This section introduces functors and lambda expressions, explains their syntax and uses, and demonstrates how they simplify writing expressive, reusable code with STL algorithms.

### 14.3.1 What Are Functors?

A **functor**, or function object, is any object that can be called like a function. This is achieved by defining the function call operator `operator()` in a class or struct. Functors behave like functions but can also hold state (data) and be passed around like objects.

### 14.3.2 Why Use Functors?

- **Stateful behavior:** Unlike plain functions, functors can keep internal state.
- **Reusability:** You can configure functors once and reuse them.
- **Performance:** Functors are often inlined by the compiler, resulting in efficient code.

### 14.3.3 Example: A Simple Functor

Full runnable code:

```
#include <iostream>

struct MultiplyBy {
    int factor;

    MultiplyBy(int f) : factor(f) {}

    int operator()(int x) const {
        return x * factor;
    }
};

int main() {
    MultiplyBy timesThree(3);
    std::cout << timesThree(5);  // Outputs 15
}
```

Here, `MultiplyBy` is a functor that multiplies its argument by a fixed factor.

### 14.3.4 Functors in STL Algorithms

STL algorithms accept functors as parameters to customize their behavior. For example, `std::sort` lets you provide a comparison functor:

Full runnable code:

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

struct Descending {
    bool operator()(int a, int b) const {
        return a > b;  // Reverse order
    }
};

int main() {
    std::vector<int> nums = {5, 2, 9, 1, 7};
    std::sort(nums.begin(), nums.end(), Descending());

    for (int n : nums) {
        std::cout << n << " ";  // Outputs: 9 7 5 2 1
    }
}
```

### 14.3.5 Introducing Lambda Expressions

While functors are flexible, they require boilerplate code to define a class or struct. **Lambda expressions**, introduced in C++11, offer a more concise way to write inline, anonymous function objects without explicit classes.

### 14.3.6 Lambda Syntax

Basic syntax of a lambda:

```cpp
[capture](parameters) -> return_type {
    // function body
};
```

- **Capture:** Specifies variables from the surrounding scope that the lambda can access.
- **Parameters:** Arguments the lambda accepts.

- **Return type:** Optional; can often be omitted if the compiler can infer it.
- **Function body:** The code executed when the lambda is called.

### 14.3.7 Example: Simple Lambda

```
auto add = [](int a, int b) {
    return a + b;
};

std::cout << add(3, 4);  // Outputs 7
```

### 14.3.8 Lambdas with STL Algorithms

Lambdas integrate smoothly with STL algorithms, making code compact and readable.

### 14.3.9 Example 1: Sorting with Lambda

Sort numbers in descending order:

```
std::vector<int> nums = {5, 2, 9, 1, 7};

std::sort(nums.begin(), nums.end(), [](int a, int b) {
    return a > b;  // Descending order
});

for (int n : nums) std::cout << n << " ";  // 9 7 5 2 1
```

### 14.3.10 Example 2: Filtering with `std::copy_if`

Copy only even numbers to another container:

Full runnable code:

```
#include <algorithm>
#include <vector>
#include <iostream>

int main() {
    std::vector<int> data = {1, 2, 3, 4, 5, 6};
    std::vector<int> evens;
    std::copy_if(data.begin(), data.end(), std::back_inserter(evens), [](int x) {
        return x % 2 == 0;
    });

    for (int n : evens) std::cout << n << " ";  // 2 4 6
}
```

### 14.3.11 Example 3: Transforming with `std::transform`

Multiply each element by 10:

```
std::vector<int> numbers = {1, 2, 3, 4};
std::vector<int> result(numbers.size());

std::transform(numbers.begin(), numbers.end(), result.begin(), [](int x) {
    return x * 10;
});

for (int n : result) std::cout << n << " ";  // 10 20 30 40
```

### 14.3.12 Capturing Variables in Lambdas

Lambda expressions can access variables from their surrounding scope by **capture**:

- `[ ]` — no captures.
- `[=]` — capture all variables by value.
- `[&]` — capture all variables by reference.
- `[x, &y]` — capture `x` by value and `y` by reference explicitly.

Example showing captures:

```
int factor = 3;
auto multiply = [factor](int x) {
    return x * factor;  // factor captured by value
};

std::cout << multiply(5);  // Outputs 15
```

### 14.3.13 When to Use Functors vs. Lambdas?

| Feature | Functors | Lambdas |
|---|---|---|
| Definition | Named class or struct with `operator()` | Anonymous inline function object |

| Feature | Functors | Lambdas |
|---|---|---|
| State (data members) | Can store data | Can capture variables from scope |
| Reusability | Reusable and can have complex logic | Usually short and used inline |
| Syntax | Verbose (class/struct definition) | Concise and expressive |
| Performance | Efficient, often inlined | Same as functors, equally efficient |

### 14.3.14 Summary

- **Functors** are function objects with customizable behavior and state, ideal for reusable logic in STL algorithms.
- **Lambda expressions** are concise, inline anonymous functions that simplify passing custom behavior, especially for short or one-off uses.
- Both functors and lambdas are fundamental tools for customizing and extending STL algorithms like `sort`, `copy_if`, and `transform`.
- Understanding when and how to use each leads to cleaner, more expressive, and maintainable C++ code.

Mastering functors and lambda expressions will empower you to write elegant and efficient code that leverages the full power of the STL.

## 14.4 String and Stream Classes

C++ provides powerful classes for handling text and formatted input/output through its Standard Template Library (STL). In this section, we explore the versatile `std::string` class and stream classes such as `stringstream`, `istringstream`, and `ostringstream`. These classes make working with text data easier, safer, and more efficient than using traditional C-style strings and manual parsing.

### 14.4.1 The `std::string` Class: Flexible and Safe Strings

`std::string` is part of the C++ Standard Library and represents a dynamic, mutable string of characters. Unlike C-style strings (which are arrays of characters terminated by a null `'\0'`), `std::string` manages memory automatically, offers convenient operations, and reduces errors.

### 14.4.2 Key Features of `std::string`

- **Dynamic size:** The string grows or shrinks as needed.
- **Safe memory management:** No manual buffer handling.
- **Rich member functions:** For searching, modifying, and extracting substrings.
- **Interoperability:** Easily convert to/from C-style strings if needed.

### 14.4.3 Basic Usage Example

Full runnable code:

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

int main() {
    std::string greeting = "Hello";
    greeting += ", world!";  // Concatenate strings

    std::cout << greeting << std::endl;  // Output: Hello, world!

    std::cout << "Length: " << greeting.length() << std::endl;  // Length: 13

    std::string sub = greeting.substr(7, 5);  // Extract substring "world"
    std::cout << sub << std::endl;
}
```

### 14.4.4 Stream Classes: Flexible Text Parsing and Formatting

C++ provides several stream classes for reading from and writing to different sources. Beyond `std::cin` and `std::cout`, the **string stream classes** enable reading and writing data directly to and from strings.

### 14.4.5 Types of String Streams

- **`std::stringstream`** — a bidirectional stream allowing both input and output.
- **`std::istringstream`** — input-only stream from a string.
- **`std::ostringstream`** — output-only stream writing into a string.

These classes are defined in the `<sstream>` header and behave similarly to file or console streams but operate on strings.

### 14.4.6 Practical Uses of String Streams

- Parsing complex strings into multiple data types.
- Formatting data into strings.
- Converting between strings and numbers.

### 14.4.7 Example 1: Parsing a String Using `istringstream`

Suppose you have a string with multiple values separated by spaces and want to extract them:

Full runnable code:

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

int main() {
    std::string input = "2025 6 26 13 45";
    std::istringstream iss(input);

    int year, month, day, hour, minute;
    iss >> year >> month >> day >> hour >> minute;

    std::cout << "Parsed date/time: "
              << year << "-" << month << "-" << day << " "
              << hour << ":" << minute << std::endl;
}
```

This outputs:

```
Parsed date/time: 2025-6-26 13:45
```

The stream extracts individual integers from the string as if reading from user input.

### 14.4.8 Example 2: Formatting Data with `ostringstream`

You can build formatted strings by inserting values into an `ostringstream`:

Full runnable code:

```
#include <iostream>
#include <sstream>

int main() {
    int score = 95;
    std::string player = "Alice";
    std::ostringstream oss;
    oss << "Player " << player << " scored " << score << " points.";

    std::string result = oss.str();
    std::cout << result << std::endl;
}
```

Output:

```
Player Alice scored 95 points.
```

Using `ostringstream` helps avoid complex string concatenations and type conversions.

### 14.4.9 Example 3: Converting Strings to Numbers and Vice Versa

Conversion between strings and numeric types is a common task. While C++11 introduced `std::stoi`, `std::stod`, etc., string streams remain useful for more complex or formatted conversions.

### 14.4.10 String to Number

Full runnable code:

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

int main() {
    std::string str = "12345";
    int number;

    std::istringstream iss(str);
    iss >> number;

    std::cout << "Number is: " << number << std::endl;
}
```

### 14.4.11 Number to String

Full runnable code:

```
#include <iostream>
#include <sstream>
```

```
int main() {
    double pi = 3.14159;

    std::ostringstream oss;
    oss << pi;

    std::string pi_str = oss.str();
    std::cout << "Pi as string: " << pi_str << std::endl;
}
```

### 14.4.12 Combining String and Stream Classes

String streams make it easier to handle structured data stored as text. For example, consider parsing CSV (comma-separated values) or formatting log messages with timestamps.

### 14.4.13 Example: Simple CSV Parsing

Full runnable code:

```
#include <iostream>
#include <sstream>
#include <vector>
#include <string>

int main() {
    std::string csv = "John,25,3.75";
    std::istringstream ss(csv);

    std::string name;
    int age;
    double gpa;

    std::getline(ss, name, ',');  // Extract until comma
    std::string age_str, gpa_str;

    std::getline(ss, age_str, ',');
    std::getline(ss, gpa_str, ',');

    std::istringstream(age_str) >> age;
    std::istringstream(gpa_str) >> gpa;

    std::cout << "Name: " << name << "\nAge: " << age << "\nGPA: " << gpa << std::endl;
}
```

Output:

```
Name: John
Age: 25
```

```
GPA: 3.75
```

### 14.4.14 Benefits of Using `std::string` and Stream Classes

- **Safety:** Avoids buffer overflows and manual memory management errors common with C-style strings.
- **Flexibility:** Easy to concatenate, compare, and manipulate strings.
- **Convenience:** Stream classes let you parse and format text cleanly without tedious code.
- **Integration:** Work seamlessly with other STL components and algorithms.

### 14.4.15 Summary

- `std::string` is the modern C++ string class providing dynamic, easy-to-use string manipulation.
- Stream classes such as `stringstream`, `istringstream`, and `ostringstream` allow parsing and formatting strings conveniently.
- Together, these tools simplify tasks like input parsing, data conversion, and formatted output.
- Using these classes leads to cleaner, safer, and more maintainable code than manual C-style string operations.

Mastering these classes will empower you to handle text and data efficiently in your C++ programs, a crucial skill in software development.

# Chapter 15.

# File Input and Output

1. Reading from and Writing to Files
2. File Streams (`ifstream`, `ofstream`, `fstream`)
3. Binary File Handling
4. Practical File Processing Examples

# 15 File Input and Output

## 15.1 Reading from and Writing to Files

File input/output (I/O) is a fundamental part of many programs that need to save data persistently or process information stored on disk. In C++, file I/O is done using **file stream** objects, which provide an interface similar to console input/output (`std::cin` and `std::cout`) but operate on files instead.

In this section, we will explore how to open files, read data from them, write data to them, and properly close them to ensure data integrity. We will also cover basic error checking and file modes to control how files are accessed.

### 15.1.1 Basic Concepts of File I/O

- **Opening a file**: You connect a file stream object to a file on disk.
- **Reading from a file**: You extract data from the file, typically line-by-line or word-by-word.
- **Writing to a file**: You send data to the file, creating or overwriting its contents.
- **Closing a file**: You disconnect the file stream to flush buffers and release system resources.

### 15.1.2 File Stream Classes in C

- **`std::ifstream`**: Input file stream for reading files.
- **`std::ofstream`**: Output file stream for writing files.
- **`std::fstream`**: File stream for both reading and writing.

All are part of the `<fstream>` header and inherit from `std::istream` or `std::ostream`.

### 15.1.3 Opening Files

Before reading or writing, you must open a file by associating a stream object with a file name.

Full runnable code:

```
#include <fstream>
#include <iostream>
```

```
int main() {
    std::ifstream inputFile("data.txt");  // Open file for reading

    if (!inputFile) {
        std::cerr << "Error: Could not open file for reading.\n";
        return 1;
    }

    // Use inputFile...

    inputFile.close();  // Close the file
}
```

- The file `"data.txt"` is opened for reading.
- Always check if the file opened successfully by testing the stream object.
- If the file does not exist or cannot be opened, the stream will be in a **fail** state.

### 15.1.4 Reading from Files

#### Reading Line-by-Line

The most common way to read text files is line-by-line using `std::getline`.

Full runnable code:

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

int main() {
    std::ifstream inputFile("example.txt");
    if (!inputFile) {
        std::cerr << "Cannot open file for reading.\n";
        return 1;
    }

    std::string line;
    while (std::getline(inputFile, line)) {
        std::cout << line << std::endl;  // Process each line
    }

    inputFile.close();
}
```

- `std::getline` reads until it encounters a newline character.
- This method is safe for any length of lines.
- Use `while (std::getline(...))` to read until the end of the file.

#### Reading Word-by-Word or Character-by-Character

You can also extract words or characters using the extraction operator (`>>`) or `get()`:

```
std::string word;
while (inputFile >> word) {
    std::cout << word << std::endl;
}
```

### Writing to Files

Use `std::ofstream` to write data to files. By default, opening a file with `ofstream` truncates (clears) the file contents or creates a new file if it does not exist.

Full runnable code:

```
#include <fstream>
#include <iostream>

int main() {
    std::ofstream outputFile("output.txt");
    if (!outputFile) {
        std::cerr << "Failed to open file for writing.\n";
        return 1;
    }

    outputFile << "Hello, file!" << std::endl;
    outputFile << "Writing multiple lines to a file." << std::endl;

    outputFile.close();
}
```

- Writing is done just like using `std::cout`.
- Remember to close the file to flush data to disk.

## 15.1.5 File Modes and Opening Options

When opening files, you can specify **file modes** to control the behavior of the file stream using `std::ios` flags:

| Mode | Description |
|---|---|
| `std::ios::in` | Open for reading (default for `ifstream`) |
| `std::ios::out` | Open for writing (default for `ofstream`) |
| `std::ios::app` | Append to the end of the file |
| `std::ios::ate` | Open and move to the end immediately |
| `std::ios::trunc` | Truncate file (default for `ofstream`) |
| `std::ios::binary` | Open in binary mode (no text translation) |

### 15.1.6 Example: Open file for appending text

```
std::ofstream logFile("log.txt", std::ios::app);
if (logFile) {
    logFile << "New log entry" << std::endl;
}
```

This adds new content without overwriting existing data.

### 15.1.7 Combining Input and Output: `fstream`

`std::fstream` can be used to open a file for both reading and writing.

Full runnable code:

```
#include <fstream>
#include <iostream>

int main() {
    std::fstream file("data.txt", std::ios::in | std::ios::out);
    if (!file) {
        std::cerr << "Unable to open file for reading and writing.\n";
        return 1;
    }

    // Use file for both reading and writing

    file.close();
}
```

Be careful managing file pointers when mixing reads and writes.

### 15.1.8 Error Checking and File States

Always check if the file operations succeed to avoid unexpected behavior:

- Use `file.is_open()` to check if the file is open.
- Test the stream in a boolean context (`if (file)`) to check if it's good.
- Use `file.fail()` or `file.eof()` to detect errors or end-of-file.

### 15.1.9 Example: Reading and Writing Files Together

This example copies the content of one text file to another:

Full runnable code:

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

int main() {
    std::ifstream inputFile("input.txt");
    std::ofstream outputFile("output.txt");

    if (!inputFile) {
        std::cerr << "Error opening input file.\n";
        return 1;
    }
    if (!outputFile) {
        std::cerr << "Error opening output file.\n";
        return 1;
    }

    std::string line;
    while (std::getline(inputFile, line)) {
        outputFile << line << std::endl;
    }

    inputFile.close();
    outputFile.close();

    std::cout << "File copied successfully.\n";
}
```

### 15.1.10 Summary

- File I/O in C++ is done using stream classes: `ifstream` for reading, `ofstream` for writing, and `fstream` for both.
- Always open files using the appropriate modes and check for successful opening.
- Use `std::getline` to read files line-by-line safely.
- Write to files with `<<` operator as you do with `std::cout`.
- Remember to close files to flush buffers and release resources.
- Use file mode flags to control how files are opened (append, truncate, binary, etc.).
- Proper error checking prevents unexpected program crashes or data loss.

Mastering file I/O lets you build programs that persist data, log information, or communicate with other applications effectively. Next, we will explore **File Streams** in detail, covering advanced techniques and best practices for robust file handling.

## 15.2 File Streams (`ifstream`, `ofstream`, `fstream`)

In C++, file input and output operations are handled through specialized classes known as **file streams**. These classes provide an interface similar to the standard console streams (`std::cin`, `std::cout`), but operate on files stored on disk.

The three main file stream classes are:

- `std::ifstream`: for **input file streams** (reading from files)
- `std::ofstream`: for **output file streams** (writing to files)
- `std::fstream`: for **file streams** that support both reading and writing

All of these classes are part of the `<fstream>` header and inherit from the standard stream classes (`std::istream`, `std::ostream`, or both).

### 15.2.1 `std::ifstream` Input File Stream

The `ifstream` class is designed for reading data from files. It inherits from `std::istream` and provides all the familiar input operations (like `operator>>` and `getline`) but reads from files instead of the keyboard.

### 15.2.2 Creating an `ifstream` Object

You can create an `ifstream` object in two ways:

1. **Default constructor and then open:**

```cpp
#include <fstream>

std::ifstream inputFile;
inputFile.open("data.txt");
if (!inputFile) {
    // Handle error
}
```

2. **Constructor that opens a file immediately:**

```cpp
std::ifstream inputFile("data.txt");
if (!inputFile) {
    // Handle error
}
```

### 15.2.3 Common Member Functions of `ifstream`

- `open(filename)`: Opens the specified file.
- `close()`: Closes the file.
- `is_open()`: Returns `true` if the file is open.
- `fail()`: Returns `true` if the last I/O operation failed.
- `eof()`: Returns `true` if the end of file has been reached.

### 15.2.4 Reading Example

Full runnable code:

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

int main() {
    std::ifstream inputFile("example.txt");
    if (!inputFile) {
        std::cerr << "Failed to open file for reading.\n";
        return 1;
    }

    std::string line;
    while (std::getline(inputFile, line)) {
        std::cout << line << std::endl;
    }

    inputFile.close();
    return 0;
}
```

This program reads the file line-by-line and prints each line to the console.

### 15.2.5 `std::ofstream` Output File Stream

The `ofstream` class is used for writing data to files. It inherits from `std::ostream` and supports output operations (`operator<<`, `write`, etc.) that write to files instead of the console.

### 15.2.6 Creating an `ofstream` Object

Similar to `ifstream`, `ofstream` can be constructed empty and opened later, or opened directly upon construction:

```
std::ofstream outputFile("output.txt");
if (!outputFile) {
    // Handle error
}
```

### 15.2.7 File Opening Modes with `ofstream`

- By default, opening a file with `ofstream` **truncates** (clears) the file.
- You can specify additional modes like `std::ios::app` to **append** instead of overwrite.

Example of appending to a file:

```
std::ofstream logFile("log.txt", std::ios::app);
```

### 15.2.8 Writing Example

Full runnable code:

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream outputFile("output.txt");
    if (!outputFile) {
        std::cerr << "Failed to open file for writing.\n";
        return 1;
    }

    outputFile << "First line of text.\n";
    outputFile << "Second line of text.\n";

    outputFile.close();
    return 0;
}
```

This program writes two lines to `output.txt`. If the file already exists, it will be overwritten unless opened in append mode.

### 15.2.9 `std::fstream` Input/Output File Stream

The `fstream` class supports both reading and writing operations. It inherits from both `std::istream` and `std::ostream`, allowing you to open a file and perform input and output on the same stream.

### 15.2.10 Creating an `fstream` Object

You must specify the file open mode explicitly when you want both input and output:

```cpp
std::fstream file("data.txt", std::ios::in | std::ios::out);
if (!file) {
    // Handle error
}
```

If the file does not exist and you want to create it, include `std::ios::trunc` or `std::ios::out` in the mode.

### 15.2.11 Common Use Case

`fstream` is useful when you need to modify an existing file without deleting its contents or to perform complex read/write operations within the same program.

### 15.2.12 Example of Reading and Writing Using `fstream`

Full runnable code:

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

int main() {
    std::fstream file("data.txt", std::ios::in | std::ios::out);
    if (!file) {
        std::cerr << "Cannot open file for reading and writing.\n";
        return 1;
    }

    // Read first line
    std::string line;
    if (std::getline(file, line)) {
        std::cout << "First line: " << line << std::endl;
    }
    // Move to end to write
    file.clear();           // Clear EOF flag if reached
    file.seekp(0, std::ios::end);
    file << "\nAdditional line added.";

    file.close();
    return 0;
}
```

This example reads the first line of the file, then appends a new line at the end.

### 15.2.13 Summary of Key Points

| Class | Purpose | Base Classes | Typical Usage |
|---|---|---|---|
| `ifstream` | Input from file | `std::istream` | Reading from text files |
| `ofstream` | Output to file | `std::ostream` | Writing/creating files |
| `fstream` | Both input and output | `std::istream` and `std::ostream` | Modifying files with read/write access |

### 15.2.14 Important Member Functions (Common to all)

- `open(filename, mode)`: Open a file with specific mode.
- `close()`: Close the file stream.
- `is_open()`: Check if the file is open.
- `fail()`: Check if an error occurred.
- Stream extraction (`>>`) and insertion (`<<`) operators for reading and writing respectively.

### 15.2.15 Practical Tips

- Always **check if the file stream is open** before performing operations to avoid runtime errors.
- Remember to **close files** explicitly or let the destructor do it when the stream object goes out of scope.
- Use **file open modes** to control behavior (append, binary mode, etc.).
- When mixing reading and writing on the same `fstream`, use `clear()` and `seekg`/`seekp` carefully to manage file pointers.

### 15.2.16 Conclusion

Understanding and using `ifstream`, `ofstream`, and `fstream` gives you powerful tools to read from and write to files in C++. These classes mirror the familiar syntax of console I/O, making file operations intuitive and flexible. Whether you need to read a configuration, log data, or modify files, mastering file streams is essential for real-world C++ programming.

In the next section, we will explore **Binary File Handling**, diving into how to read and write binary data efficiently with file streams.

## 15.3 Binary File Handling

In C++, file input and output usually involve working with **text files**, which store data as human-readable characters. However, some applications require handling **binary files**—files that contain raw bytes representing data in a compact and precise form. Examples include image files, audio files, serialized objects, and other non-text data formats.

This section introduces binary file operations in C++, explains their differences from text file handling, and shows how to use the `read()` and `write()` member functions of file streams for working with raw data.

### 15.3.1 Why Use Binary Files?

Text files are easy to read and edit but store data inefficiently. For example, storing an integer `1234` in a text file writes the characters `'1'`, `'2'`, `'3'`, `'4'`, which take four bytes, plus possibly a newline. In contrast, a binary file stores the integer in its native binary format, usually 4 bytes (on most systems).

### 15.3.2 When to Use Binary Files?

- **Performance and space efficiency:** Binary files are more compact and faster to read/write because they don't involve formatting conversions.
- **Exact representation:** Data like floating-point numbers, structs, or raw buffers must be saved exactly as they are.
- **Interoperability with other systems:** Many file formats (images, audio, video) require binary access.
- **Serialization:** Saving program data in binary allows precise reconstruction later.

### 15.3.3 Opening Files in Binary Mode

To work with binary files, you open files with the `std::ios::binary` flag in addition to the usual input/output modes.

Example:

```
std::ofstream outFile("data.bin", std::ios::binary);
std::ifstream inFile("data.bin", std::ios::binary);
```

Without the `binary` flag, the file is treated as text, and the system might perform newline translations or other conversions, corrupting binary data.

### 15.3.4 Reading and Writing Binary Data with `read()` and `write()`

The main difference between text and binary file I/O is in how data is transferred:

- **Text I/O** uses formatted input/output operators like `<<` and `>>`.
- **Binary I/O** uses the unformatted member functions `write()` and `read()`, which operate on raw memory buffers.

### 15.3.5 Syntax

```
ostream.write(const char* buffer, std::streamsize size);
istream.read(char* buffer, std::streamsize size);
```

- `buffer`: pointer to the memory area to write/read.
- `size`: number of bytes to write/read.

### 15.3.6 Example: Writing and Reading an Integer in Binary

Full runnable code:

```
#include <iostream>
#include <fstream>

int main() {
    int number = 123456789;

    // Writing integer to binary file
    std::ofstream outFile("number.bin", std::ios::binary);
    if (!outFile) {
        std::cerr << "Failed to open file for writing.\n";
        return 1;
    }
    outFile.write(reinterpret_cast<const char*>(&number), sizeof(number));
    outFile.close();

    // Reading integer from binary file
    int readNumber = 0;
    std::ifstream inFile("number.bin", std::ios::binary);
    if (!inFile) {
        std::cerr << "Failed to open file for reading.\n";
        return 1;
    }
    inFile.read(reinterpret_cast<char*>(&readNumber), sizeof(readNumber));
    inFile.close();

    std::cout << "Read number: " << readNumber << std::endl;
    return 0;
}
```

- We use `reinterpret_cast` to convert the pointer to a `char*` because `write()` and `read()` expect a pointer to a character buffer.
- `sizeof(number)` ensures the exact number of bytes for the data type is written/read.

### 15.3.7 Writing and Reading Arrays and Structures

Binary files can store more complex data like arrays or structs, provided the memory layout is consistent.

### 15.3.8 Example: Writing and Reading an Array

```
int arr[5] = {1, 2, 3, 4, 5};

std::ofstream outFile("array.bin", std::ios::binary);
outFile.write(reinterpret_cast<const char*>(arr), sizeof(arr));
outFile.close();

int arrRead[5];
std::ifstream inFile("array.bin", std::ios::binary);
inFile.read(reinterpret_cast<char*>(arrRead), sizeof(arrRead));
inFile.close();

for (int i = 0; i < 5; ++i) {
    std::cout << arrRead[i] << " ";
}
```

### 15.3.9 Example: Writing and Reading a Struct

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

Point p1 = {10, 20};

std::ofstream outFile("point.bin", std::ios::binary);
outFile.write(reinterpret_cast<const char*>(&p1), sizeof(p1));
outFile.close();

Point p2;
std::ifstream inFile("point.bin", std::ios::binary);
inFile.read(reinterpret_cast<char*>(&p2), sizeof(p2));
inFile.close();

std::cout << "Point read: (" << p2.x << ", " << p2.y << ")\n";
```

### 15.3.10 Important Considerations

#### Data Alignment and Padding

Structures in C++ might have **padding bytes** inserted for alignment, meaning the memory layout can include unused bytes. This affects how data is written and read:

- Reading binary data written on one system may fail on another if the compiler applies different padding or alignment rules.
- Use compiler-specific **packing directives** or manually serialize fields for portable binary formats.

#### Endianness (Byte Order)

Different computer architectures store multi-byte values in different byte orders (little-endian or big-endian):

- If a binary file is shared between systems with different endianness, values may be misinterpreted.
- Consider converting data to a standard byte order (like network byte order) before writing.

#### Portability and Compatibility

Binary files are **not inherently portable** across different platforms, compilers, or architectures without care. For cross-platform data exchange, consider:

- Standardized serialization libraries (e.g., Protocol Buffers, Boost.Serialization).
- Text-based formats like JSON or XML (though less efficient).

### 15.3.11 Error Checking with Binary I/O

Always check the success of `read()` and `write()`:

```cpp
if (!outFile.write(reinterpret_cast<const char*>(&number), sizeof(number))) {
    std::cerr << "Write failed.\n";
}

if (!inFile.read(reinterpret_cast<char*>(&readNumber), sizeof(readNumber))) {
    std::cerr << "Read failed.\n";
}
```

### 15.3.12 Summary

| Aspect | Text File I/O | Binary File I/O |
|---|---|---|
| Data stored as | Characters readable by humans | Raw bytes (compact, exact representation) |
| File opening mode | Default | Use `std::ios::binary` flag |
| I/O functions | Formatted (<<, >>, `getline`) | Unformatted (`read()`, `write()`) |
| Use cases | Config files, logs, simple data | Images, audio, serialized objects, performance-critical data |
| Portability | High | Can be low; depends on platform and data layout |

### 15.3.13 Conclusion

Binary file handling in C++ gives you precise control over how data is stored and read from files, which is critical in many real-world applications such as image processing, network protocols, and serialization of complex objects. By mastering `read()` and `write()`, and understanding the nuances of data alignment and portability, you can implement efficient and reliable binary I/O in your programs.

In the next section, we will explore **Practical File Processing Examples**, tying together concepts from both text and binary file handling into real-world scenarios.

## 15.4 Practical File Processing Examples

In this section, we will explore realistic examples that combine the core file input/output concepts you've learned. These hands-on scenarios include reading and parsing CSV files, logging application data, and copying files. Along the way, we'll emphasize best practices for error handling, resource management, and writing clean, efficient code.

### 15.4.1 Parsing a CSV File

CSV (Comma-Separated Values) is a popular format for tabular data where each line represents a row and each field is separated by commas. Parsing CSV files is a common task for processing data such as user records, financial data, or configuration parameters.

### 15.4.2 Example: Reading a CSV File and Summarizing Data

Suppose we have a file named `sales.csv` with the following content:

```
Product,Quantity,Price
Apple,10,0.5
Banana,5,0.3
Orange,8,0.7
```

We want to read this file, parse each line, and calculate the total sales for each product.

Full runnable code:

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

int main() {
    std::ifstream file("sales.csv");
    if (!file) {
        std::cerr << "Error: Cannot open sales.csv\n";
        return 1;
    }

    std::string line;
    // Read the header line and ignore
    std::getline(file, line);

    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string product;
        int quantity;
        double price;
        // Parse the CSV line
        std::getline(ss, product, ',');
        std::string quantityStr, priceStr;
        std::getline(ss, quantityStr, ',');
        std::getline(ss, priceStr, ',');

        try {
            quantity = std::stoi(quantityStr);
            price = std::stod(priceStr);
        } catch (const std::exception& e) {
            std::cerr << "Invalid number format in line: " << line << '\n';
            continue; // Skip to next line
        }

        double totalSale = quantity * price;
        std::cout << "Product: " << product
                  << ", Total Sale: $" << totalSale << '\n';
    }

    file.close();
    return 0;
}
```

### 15.4.3 Key points:

- **Use `std::getline()`** to read lines from the file.
- **Use `std::stringstream`** to parse each line by commas.
- **Error handling**: Check if the file opens successfully and catch exceptions during conversion.
- **Resource management**: Explicitly close the file (though RAII would close it automatically when `ifstream` goes out of scope).

### 15.4.4 Logging Application Data to a File

Logging is essential for monitoring program behavior, debugging, and auditing. The example below shows how to write log messages with timestamps to a log file.

### 15.4.5 Example: Simple Logger Writing to a File

Full runnable code:

```
#include <iostream>
#include <fstream>
```

```
#include <ctime>
#include <string>

void logMessage(const std::string& message, std::ofstream& logFile) {
    if (!logFile) {
        std::cerr << "Log file not open\n";
        return;
    }

    // Get current time
    std::time_t now = std::time(nullptr);
    char timeStr[20];
    std::strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H:%M:%S", std::localtime(&now));

    logFile << "[" << timeStr << "] " << message << std::endl;
}

int main() {
    std::ofstream logFile("app.log", std::ios::app); // Append mode
    if (!logFile) {
        std::cerr << "Failed to open log file\n";
        return 1;
    }

    logMessage("Application started", logFile);
    logMessage("Performing some task", logFile);
    logMessage("Application finished", logFile);

    logFile.close();
    return 0;
}
```

### 15.4.6 Key points:

- Open the log file in **append mode (`std::ios::app`)** to avoid overwriting.
- Use **timestamps** to record when events occur.
- Check if the file is successfully opened before writing.
- Writing to the log file uses the insertion operator (`<<`), just like standard output.

### 15.4.7 Copying a File Efficiently

Copying a file is a common task. This example shows how to read a source file and write its contents to a destination file efficiently using binary mode, which works well for all file types including text, images, and executables.

### 15.4.8 Example: File Copy Utility

Full runnable code:

```
#include <iostream>
#include <fstream>

int main() {
    std::ifstream source("source.txt", std::ios::binary);
    if (!source) {
        std::cerr << "Error: Cannot open source file\n";
        return 1;
    }

    std::ofstream dest("destination.txt", std::ios::binary);
    if (!dest) {
        std::cerr << "Error: Cannot open destination file\n";
        return 1;
    }

    // Buffer for copying
    const size_t bufferSize = 4096;
    char buffer[bufferSize];

    while (source.read(buffer, bufferSize)) {
        dest.write(buffer, source.gcount());
    }
    // Write any remaining bytes if the last read was partial
    dest.write(buffer, source.gcount());

    source.close();
    dest.close();

    std::cout << "File copied successfully.\n";
    return 0;
}
```

### 15.4.9 Key points:

- Open files in **binary mode** to copy raw bytes correctly.
- Use a **buffer** to read and write chunks rather than byte-by-byte for performance.
- Use `gcount()` to know how many bytes were actually read on the last operation.
- Check for file open errors before proceeding.

### 15.4.10 Best Practices for File Processing

#### Error Handling

Always check if files are successfully opened before reading or writing to avoid crashes or undefined behavior.

```
if (!file) {
    std::cerr << "Failed to open file.\n";
    // Handle error or exit
}
```

Also, validate that I/O operations succeed:

```
if (!file.read(buffer, size)) {
    std::cerr << "Error during read operation.\n";
}
```

#### Resource Management

- Use RAII: Prefer **automatic variables** for streams so files close automatically when they go out of scope.
- Avoid manual calls to `close()` unless you want to close and reopen during the program.

#### Efficient Data Processing

- Use buffered reads/writes instead of character-by-character to improve performance.
- Use standard library utilities (`std::getline`, `std::stringstream`) for parsing text files.
- Use binary mode when dealing with raw data.

#### Clean and Modular Code

- Separate file I/O logic from data processing logic.
- Wrap file operations in functions to improve readability and reuse.

### 15.4.11 Summary

This section provided practical examples demonstrating:

- Reading and parsing structured text files like CSV.
- Logging runtime data to files with timestamps.
- Copying files efficiently in binary mode.
- Essential error checking and resource management techniques.

These examples are foundational for many real-world applications, and mastering these will give you the confidence to handle file operations in more complex programs.

In the next chapter, we will explore **Advanced Topics** including memory management, smart pointers, and best practices for modern C++ programming.

# Chapter 16.

# Advanced Memory Management

1. Smart Pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`)
2. Move Semantics and Rvalue References
3. RAII (Resource Acquisition Is Initialization)
4. Memory Leaks and Tools for Detection

# 16 Advanced Memory Management

## 16.1 Smart Pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`)

Managing dynamic memory manually with raw pointers in C++ can be error-prone. Common problems such as memory leaks, dangling pointers, or double deletions often arise when `new` and `delete` calls are not carefully balanced. To address these issues, **smart pointers** were introduced in C++ to provide safer, automatic memory management. They encapsulate raw pointers and control the lifetime of dynamically allocated objects, reducing the risk of bugs and making code easier to maintain.

In this section, we explore the three primary smart pointer types provided by the C++ Standard Library: `unique_ptr`, `shared_ptr`, and `weak_ptr`. Each serves a specific purpose for managing ownership and object lifetimes.

### 16.1.1 Why Use Smart Pointers?

Before diving into each type, let's briefly outline why smart pointers are important:

- **Automatic deletion:** They automatically delete the managed object when it is no longer needed.
- **Exception safety:** They prevent resource leaks even if exceptions occur.
- **Clear ownership semantics:** They express who "owns" an object, making code easier to understand.
- **Avoid dangling pointers:** They reduce the risk of pointers referring to deleted objects.

### 16.1.2 `unique_ptr`: Exclusive Ownership

`std::unique_ptr` represents *exclusive ownership* of a dynamically allocated object. At any time, only one `unique_ptr` can own the object. When the `unique_ptr` goes out of scope or is reset, it automatically deletes the owned object.

### 16.1.3 Creating and Using `unique_ptr`

Full runnable code:

```
#include <iostream>
#include <memory>
```

```
int main() {
    std::unique_ptr<int> ptr1(new int(42)); // owns an int with value 42

    std::cout << "Value: " << *ptr1 << '\n'; // Access via dereference

    // Transferring ownership using std::move
    std::unique_ptr<int> ptr2 = std::move(ptr1);
    if (!ptr1) {
        std::cout << "ptr1 is now null after move\n";
    }
    std::cout << "ptr2 owns: " << *ptr2 << '\n';

    // Automatic deletion when ptr2 goes out of scope
    return 0;
}
```

### 16.1.4 Important points about `unique_ptr`:

- **No copy allowed:** Copying is disallowed to enforce exclusive ownership. You must use `std::move()` to transfer ownership.
- **Automatic deletion:** When the last `unique_ptr` owning the object is destroyed or reset, it deletes the object.
- **Custom deleters:** You can provide a custom deleter if needed (e.g., to delete arrays or close files).

```
std::unique_ptr<int[]> arr(new int[10]); // managing dynamic array
```

### 16.1.5 Pitfalls to avoid

- Don't manually delete the managed object.
- Avoid creating multiple `unique_ptr`s managing the same raw pointer; it leads to double deletion.
- Always use `std::move` to transfer ownership, not copy.

### 16.1.6 `shared_ptr`: Shared Ownership with Reference Counting

Sometimes, multiple parts of a program need to share ownership of an object. `std::shared_ptr` allows multiple pointers to *share ownership* of the same resource. The object is destroyed only when the *last* `shared_ptr` owning it is destroyed or reset.

### 16.1.7 Creating and Using `shared_ptr`

Full runnable code:

```
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> sp1 = std::make_shared<int>(100);
    std::cout << "Value: " << *sp1 << '\n';
    std::cout << "Use count: " << sp1.use_count() << '\n'; // 1

    {
        std::shared_ptr<int> sp2 = sp1; // shared ownership
        std::cout << "Use count after copy: " << sp1.use_count() << '\n'; // 2
        std::cout << "sp2 points to: " << *sp2 << '\n';
    } // sp2 goes out of scope here

    std::cout << "Use count after sp2 destroyed: " << sp1.use_count() << '\n'; // 1

    return 0;
}
```

### 16.1.8 Key features of `shared_ptr`:

- Maintains an internal *reference count* to track how many `shared_ptr`s own the object.
- When the last `shared_ptr` is destroyed or reset, the managed object is deleted.
- You can query the number of owners with `use_count()`.
- You can safely copy `shared_ptr`s, increasing the reference count.

### 16.1.9 Potential issues

- **Circular references:** If two `shared_ptr`s reference each other, the reference count never reaches zero, causing a memory leak. This is where `weak_ptr` helps (explained next).
- Slightly more overhead than `unique_ptr` due to reference counting.

### 16.1.10 `weak_ptr`: Breaking Cyclic Dependencies

`std::weak_ptr` is a companion to `shared_ptr`. It holds a *non-owning* ("weak") reference to an object managed by `shared_ptr`. It does **not** increase the reference count and thus does not affect the lifetime of the object.

### 16.1.11 Why `weak_ptr`?

When two or more objects hold `shared_ptr`s to each other, they form a **reference cycle**, preventing their memory from ever being freed. Using `weak_ptr` breaks this cycle by providing a way to reference an object without extending its lifetime.

### 16.1.12 Using `weak_ptr`

Full runnable code:

```cpp
#include <iostream>
#include <memory>

struct Node {
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev; // weak_ptr to break cycle
    int value;

    Node(int v) : value(v) {}
};

int main() {
    auto first = std::make_shared<Node>(1);
    auto second = std::make_shared<Node>(2);

    first->next = second;
    second->prev = first; // weak_ptr doesn't increase ref count

    std::cout << "First use count: " << first.use_count() << '\n';  // 1
    std::cout << "Second use count: " << second.use_count() << '\n'; // 1

    // Access prev node safely
    if (auto prevPtr = second->prev.lock()) {
        std::cout << "Second's prev value: " << prevPtr->value << '\n';
    }

    return 0;
}
```

### 16.1.13 Important aspects of `weak_ptr`:

- You **cannot directly dereference** a `weak_ptr`. Instead, call `lock()` which returns a `shared_ptr` if the object is still alive.
- If the object has been deleted, `lock()` returns an empty `shared_ptr`.
- Useful for caching, observer patterns, or any situation where you want to observe an object without preventing its deletion.

### 16.1.14 Summary of Smart Pointer Usage

| Smart Pointer | Ownership Type | Copyable? | Use Case |
|---|---|---|---|
| `unique_ptr` | Exclusive | No (move only) | When single ownership is needed |
| `shared_ptr` | Shared (reference counting) | Yes | When multiple owners share the object |
| `weak_ptr` | Non-owning observer | Yes | To avoid circular references or observe object |

### 16.1.15 Additional Tips and Best Practices

- Prefer `std::make_unique<T>()` and `std::make_shared<T>()` to create smart pointers. They are safer and more efficient.

```cpp
auto up = std::make_unique<int>(10);
auto sp = std::make_shared<int>(20);
```

- Avoid using raw `new` and `delete` directly; smart pointers handle memory safely.
- Avoid mixing raw pointers and smart pointers for the same resource.
- Remember that `unique_ptr` is lighter and faster but less flexible than `shared_ptr`.
- Always use `weak_ptr` to break reference cycles when designing complex object graphs.

### 16.1.16 Conclusion

Smart pointers are a fundamental part of modern C++ programming, providing powerful tools to manage dynamic memory safely and efficiently. `unique_ptr` guarantees exclusive ownership and zero-overhead automatic cleanup, `shared_ptr` enables shared ownership with automatic reference counting, and `weak_ptr` prevents cyclic dependencies and dangling references.

By incorporating smart pointers into your programs, you reduce the chances of memory leaks and dangling pointers, making your code more robust, readable, and maintainable.

## 16.2 Move Semantics and Rvalue References

In traditional C++, copying objects—especially those managing dynamic resources like memory or file handles—can be expensive. Copy constructors and copy assignment operators duplicate the underlying data, which can be slow and resource-intensive, particularly for large objects or containers.

**Move semantics**, introduced in C++11, provide a way to transfer resources from one object to another **without copying**, optimizing performance. This powerful feature uses **rvalue references** and enables the compiler and programmer to "move" resources rather than duplicate them.

In this section, we will explore the concepts behind move semantics, explain rvalue references and their distinction from lvalues, and show how to implement move constructors and move assignment operators. We will also discuss how `std::move` helps explicitly indicate moves and examine the practical benefits through examples.

### 16.2.1 Understanding Lvalues and Rvalues

To understand move semantics, it is crucial to know the difference between **lvalues** and **rvalues**:

- **Lvalue (locator value):** An expression that refers to a specific memory location and has an identifiable address. Variables you can take the address of are lvalues. For example:

  ```
  int x = 5;      // x is an lvalue
  int* p = &x;    // legal: taking address of x
  ```

- **Rvalue (read value):** A temporary object or value that does not have a persistent memory address. Examples include literals, temporary objects returned from functions, or expressions like `x + y`. For example:

  ```
  int y = 10;
  int z = x + y;  // x + y is an rvalue
  ```

Before C++11, you could only bind lvalues to references (e.g., `int& ref = x;`). Rvalues could not bind to non-const references, limiting optimization opportunities.

### 16.2.2 Rvalue References (`&&`)

C++11 introduced **rvalue references**, declared with `&&`, allowing functions and constructors to accept rvalues explicitly.

```
int&& rref = 10;  // binds to temporary (rvalue)
```

This enables functions to differentiate between lvalues and rvalues, so they can:

- Copy resources when passed an lvalue (to keep the original intact).
- Move resources (i.e., steal ownership) when passed an rvalue, avoiding expensive copies.

Rvalue references are the foundation of move semantics.

### 16.2.3 What Is Move Semantics?

Move semantics allow the **resources owned by a temporary object (rvalue)** to be transferred ("moved") to another object rather than copied. After the move, the source object is typically left in a valid but unspecified (often "empty") state.

This transfer avoids the costly operations involved in copying large resources like dynamically allocated arrays, file handles, or sockets.

### 16.2.4 Using `std::move`

The function `std::move` is a utility that **casts its argument to an rvalue reference**, enabling move semantics for objects that are otherwise lvalues.

```cpp
#include <utility>  // for std::move

std::string s1 = "Hello";
std::string s2 = std::move(s1);  // Move s1's content to s2
```

After this, `s1` is in a valid but unspecified state (often empty), and `s2` now owns the original string content.

### 16.2.5 Move Constructor and Move Assignment Operator

Classes that manage resources can define:

- A **move constructor**: constructs an object by taking ownership of the resources from an rvalue.
- A **move assignment operator**: transfers resources from an rvalue to an existing object.

These are similar in purpose to copy constructor and copy assignment but optimized for moving resources.

### 16.2.6 Move Constructor Syntax

```
ClassName(ClassName&& other) noexcept;
```

- Takes an rvalue reference to another object.
- Transfers ownership of resources.
- Leaves `other` in a safe-to-destruct state.

### 16.2.7 Move Assignment Operator Syntax

```
ClassName& operator=(ClassName&& other) noexcept;
```

- Transfers ownership from `other` to the current object.
- Properly frees any existing resources before the move.
- Leaves `other` in a valid state.

### 16.2.8 Example: Move Semantics in a Simple String Wrapper

Let's illustrate move semantics with a simplified string wrapper class:

Full runnable code:

```
#include <iostream>
#include <cstring>

class MyString {
private:
    char* data;
public:
    // Constructor
    MyString(const char* s) {
        if (s) {
            data = new char[strlen(s) + 1];
            strcpy(data, s);
        } else {
            data = nullptr;
        }
    }

    // Destructor
    ~MyString() {
        delete[] data;
    }

    // Copy Constructor
    MyString(const MyString& other) {
        std::cout << "Copy constructor called\n";
        if (other.data) {
            data = new char[strlen(other.data) + 1];
            strcpy(data, other.data);
        } else {
            data = nullptr;
        }
    }

    // Move Constructor
    MyString(MyString&& other) noexcept {
        std::cout << "Move constructor called\n";
        data = other.data;    // Steal resource
        other.data = nullptr; // Leave other in safe state
    }

    // Copy Assignment
    MyString& operator=(const MyString& other) {
        std::cout << "Copy assignment called\n";
        if (this != &other) {
            delete[] data;
            if (other.data) {
                data = new char[strlen(other.data) + 1];
                strcpy(data, other.data);
            } else {
                data = nullptr;
            }
        }
        return *this;
    }

    // Move Assignment
    MyString& operator=(MyString&& other) noexcept {
        std::cout << "Move assignment called\n";
        if (this != &other) {
            delete[] data;
            data = other.data;    // Steal resource
            other.data = nullptr; // Safe state for other
        }
        return *this;
    }

    void print() const {
        if (data) std::cout << data << '\n';
        else std::cout << "(null)\n";
    }
};

int main() {
    MyString s1("Hello");
    MyString s2 = std::move(s1);  // Calls move constructor

    s2.print();
    s1.print(); // Should be null

    MyString s3("World");
    s3 = std::move(s2);  // Calls move assignment

    s3.print();
    s2.print(); // Should be null

    return 0;
}
```

### 16.2.9 Output

```
Move constructor called
Hello
(null)
Move assignment called
Hello
(null)
```

### 16.2.10 Explanation:

- Move constructor steals the internal pointer instead of allocating and copying.
- After the move, the source (`s1` or `s2`) is left with `nullptr` to prevent double deletion.
- Move assignment frees any current resource before stealing the new one.

### 16.2.11 Performance Benefits

Move semantics significantly improve performance by:

- Eliminating deep copies of large resources.
- Enabling efficient transfer of ownership in STL containers like `std::vector`, `std::string`, `std::unique_ptr`.
- Reducing unnecessary memory allocations and deallocations.

For example, when a vector resizes, it moves elements to the new storage using move constructors if available, making resizing faster than copying.

### 16.2.12 Using Move Semantics with STL Containers

The Standard Library containers support move semantics. Consider this example with a vector of strings:

Full runnable code:

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

int main() {
    std::vector<std::string> v;
    v.push_back("One");
    v.push_back("Two");

    std::string s = "Three";
    v.push_back(std::move(s));  // Moves 's' into vector

    std::cout << "Vector contents:\n";
    for (const auto& str : v) {
        std::cout << str << '\n';
    }
    std::cout << "Original string after move: " << s << '\n'; // Usually empty

    return 0;
}
```

### 16.2.13 Summary and Best Practices

- Implement move constructors and move assignment operators for classes managing resources.
- Use `std::move` to enable moves for lvalues when appropriate.
- Mark move operations `noexcept` if possible to enable optimizations.
- Prefer `std::make_unique` and `std::make_shared` to create smart pointers efficiently using move semantics.
- Understand that after moving, the source object is left in a valid but unspecified state.
- Use move semantics to optimize performance, especially for large objects or containers.

### 16.2.14 Conclusion

Move semantics and rvalue references provide powerful tools for efficient resource management in C++. By enabling the transfer of resources instead of expensive copies, move semantics improve performance and express intent clearly. With practice, move semantics become an essential technique for writing modern, optimized C++ code.

In the next section, we will explore **RAII (Resource Acquisition Is Initialization)**—a core idiom for managing resources safely and effectively in C++.

## 16.3 RAII (Resource Acquisition Is Initialization)

In C++, managing resources such as memory, file handles, or locks can be tricky. Forgetting to release these resources leads to issues like memory leaks, resource exhaustion, or deadlocks. To solve these problems in a clean, robust way, C++ programmers rely on a fundamental idiom called **RAII**, which stands for **Resource Acquisition Is Initialization**.

RAII ties the lifetime of a resource directly to the lifetime of an object, ensuring that resources are acquired and released in a controlled, exception-safe manner.

### 16.3.1 What Is RAII?

RAII is a design pattern where **resource management is bound to the lifespan of an object**. When an object is created (initialized), it acquires the resource. When the object goes out of scope or is destroyed, its destructor automatically releases the resource.

This automatic acquisition and release help eliminate many common bugs related to manual resource management, such as:

- Memory leaks (forgetting to `delete` allocated memory)
- File handles left open
- Mutexes never unlocked

In other words, resource management becomes predictable and safe, even in the presence of exceptions.

### 16.3.2 How RAII Works

The core idea is simple:

- **Constructor acquires the resource** (e.g., opens a file, allocates memory, locks a mutex).
- **Destructor releases the resource** (e.g., closes the file, frees memory, unlocks the mutex).

Because destructors run automatically when an object goes out of scope (stack unwinding), resources are always properly cleaned up, even if an exception is thrown.

### 16.3.3 RAII Example: Managing a File Handle

Consider managing a file with manual `fopen` and `fclose` calls:

```
#include <cstdio>

void writeToFile(const char* filename) {
    FILE* file = fopen(filename, "w");
    if (!file) {
        throw std::runtime_error("Failed to open file");
    }

    fprintf(file, "Hello RAII!\n");

    // Must remember to close file manually
    fclose(file);
}
```

If an exception is thrown before `fclose` is called, the file remains open, potentially causing resource leaks.

Using RAII, we wrap the file in a class that handles opening and closing automatically:

```
#include <cstdio>
#include <stdexcept>

class FileRAII {
private:
    FILE* file;
public:
    FileRAII(const char* filename, const char* mode) {
        file = fopen(filename, mode);
        if (!file) {
            throw std::runtime_error("Failed to open file");
        }
    }

    ~FileRAII() {
        if (file) {
            fclose(file);
        }
    }

    FILE* get() const { return file; }

    // Disable copying to avoid double fclose
    FileRAII(const FileRAII&) = delete;
    FileRAII& operator=(const FileRAII&) = delete;
};

void writeToFile(const char* filename) {
    FileRAII file(filename, "w");
    fprintf(file.get(), "Hello RAII!\n");

    // No explicit fclose needed; handled automatically
}
```

Now, regardless of how the function exits (normal return or exception), the file is closed properly.

### 16.3.4 RAII with Mutex Locks

RAII is especially important for managing **locks** in concurrent programming to prevent deadlocks.

Manually locking and unlocking a mutex can be error-prone:

```cpp
#include <mutex>

std::mutex mtx;

void doWork() {
    mtx.lock();
    // ... critical section ...
    mtx.unlock();
}
```

If an exception occurs inside the critical section, `mtx.unlock()` may never be called, causing deadlocks.

Using RAII, the Standard Library provides `std::lock_guard` to manage mutexes:

```cpp
#include <mutex>

std::mutex mtx;

void doWork() {
    std::lock_guard<std::mutex> lock(mtx);  // locks on construction
    // ... critical section ...

    // automatically unlocks when 'lock' goes out of scope
}
```

`std::lock_guard` locks the mutex in its constructor and unlocks it in its destructor, guaranteeing the mutex is always released.

### 16.3.5 RAII and Smart Pointers

Smart pointers, such as `std::unique_ptr` and `std::shared_ptr`, are classic RAII examples that manage dynamic memory:

```cpp
#include <memory>

void example() {
    std::unique_ptr<int> ptr(new int(42));  // acquires memory

    // use ptr like a normal pointer
    std::cout << *ptr << std::endl;

    // no explicit delete needed; memory freed when ptr is destroyed
}
```

When the smart pointer goes out of scope, it automatically deletes the memory it manages, preventing leaks and dangling pointers.

### 16.3.6 Benefits of RAII

- **Exception Safety:** Resources are released even when exceptions occur, preventing leaks.
- **Simplified Code:** No need for explicit resource release calls scattered throughout code.
- **Improved Maintainability:** The resource management logic is localized within RAII classes.
- **Predictable Lifetime:** The resource lifetime exactly matches the object lifetime.

### 16.3.7 Guidelines for Implementing RAII

- Encapsulate resource acquisition in constructors.
- Encapsulate resource release in destructors.
- Disable copying if copying a resource doesn't make sense or implement deep copies.
- Prefer using existing RAII wrappers (like smart pointers, lock guards) instead of reinventing the wheel.
- Make RAII classes small and focused, managing a single resource.

### 16.3.8 Summary

RAII is a cornerstone idiom in C++ programming that leverages constructors and destructors to manage resources automatically. It ties resource lifetime to object lifetime, ensuring resources like memory, file handles, and locks are safely acquired and released.

By embracing RAII, your code becomes safer, cleaner, and easier to maintain — especially when dealing with exceptions or complex control flows.

## 16.4 Memory Leaks and Tools for Detection

Memory leaks are a common and critical problem in C++ programming, especially in applications that manage dynamic memory manually. They occur when a program allocates memory on the heap but fails to release it, causing the application's memory usage to grow unnecessarily over time. This can lead to degraded performance, system instability, or even crashes due to exhausted memory.

### 16.4.1 Common Causes of Memory Leaks in C

Manual Memory Management Mistakes

In C++, dynamic memory is often allocated using `new` and deallocated with `delete`. Forgetting to call `delete` after a `new` allocation causes memory leaks:

```
void leakExample() {
    int* ptr = new int(42);
    // Forgot to delete ptr
}
```

### 16.4.2 Lost Pointers

Another common cause is losing the pointer to allocated memory before freeing it:

```
void leakExample() {
    int* ptr = new int[10];
    ptr = nullptr;  // lost the reference to allocated memory
    // No delete[] called, memory leaked
}
```

### 16.4.3 Cyclic References in Smart Pointers

Even with smart pointers, memory leaks can occur if there are **cyclic references**, especially with `std::shared_ptr`. For example, two objects holding shared pointers to each other prevent both from being destroyed.

### 16.4.4 Impact of Memory Leaks

- **Increased Memory Usage:** Applications consume more RAM over time.
- **Performance Degradation:** More memory usage can slow down systems.
- **Crashes:** Out-of-memory conditions can cause program failure.
- **Hard-to-Debug Issues:** Leaks may not manifest immediately, making them challenging to identify.

### 16.4.5 Tools and Techniques for Detecting Memory Leaks

C++ developers use specialized tools to detect and analyze memory leaks. These tools monitor your program's memory allocations and deallocations, helping you identify leaks and their locations.

### 16.4.6 Valgrind (Linux/macOS)

Valgrind is a popular open-source tool for detecting memory leaks and errors:

- **How it works:** Runs your program in a virtual environment, tracking memory operations.
- **Usage:** Run your executable with `valgrind ./your_program`.
- **Output:** Provides detailed reports of leaked blocks, including allocation stack traces.

**Example Valgrind command:**

```
valgrind --leak-check=full ./your_program
```

Valgrind will list all leaks and indicate where memory was allocated but never freed.

### 16.4.7 AddressSanitizer (ASan)

AddressSanitizer is a fast memory error detector integrated with compilers like GCC and Clang:

- **How it works:** Adds instrumentation during compilation to check memory operations at runtime.
- **Usage:** Compile with `-fsanitize=address -g` flags.
- **Output:** Detects leaks, buffer overflows, use-after-free, and more.

**Example compile command:**

```
g++ -fsanitize=address -g main.cpp -o main
./main
```

ASan outputs detailed diagnostics with line numbers and stack traces.

### 16.4.8 IDE-Integrated Profilers and Debuggers

Many IDEs like Visual Studio, CLion, and Xcode provide built-in tools or plugins for memory profiling:

- **Visual Studio:** Offers a Diagnostic Tools window with memory usage reports.
- **CLion:** Supports Valgrind integration and its own profiler.
- **Xcode:** Instruments tool can track leaks and allocations.

These tools offer GUI interfaces for easier analysis.

### 16.4.9 Best Practices for Writing Leak-Free Code

#### Prefer RAII and Smart Pointers

Use RAII techniques and smart pointers (`std::unique_ptr`, `std::shared_ptr`) to automate memory management and avoid manual `new`/`delete` mistakes.

#### Avoid Raw `new` and `delete` in High-Level Code

Minimize direct use of `new` and `delete`. If manual management is necessary, encapsulate allocations within classes that follow RAII principles.

#### Watch Out for Cyclic References

Use `std::weak_ptr` to break cycles in shared ownership scenarios:

```
struct Node {
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;  // prevents cyclic references
};
```

#### Always Match `new[]` with `delete[]`

When allocating arrays, use `new[]` and pair it with `delete[]` to avoid undefined behavior and leaks.

#### Initialize Pointers and Reset Them After Deletion

Set pointers to `nullptr` after deleting to avoid dangling pointers and accidental reuse.

### 16.4.10 Interpreting Tool Output

#### Valgrind Example Output

```
==1234== 40 bytes in 1 blocks are definitely lost in loss record 1 of 2
==1234==    at 0x4C2BBAF: operator new(unsigned long) (vg_replace_malloc.c:334)
==1234==    by 0x4006F4: leakExample() (example.cpp:5)
==1234==    by 0x400720: main (example.cpp:12)
```

- Shows the number of bytes leaked.
- Shows the allocation site (file and line number).
- Use this info to locate and fix the leak.

### 16.4.11 AddressSanitizer Output

```
=================================================================
==5678==ERROR: LeakSanitizer: detected memory leaks
```

```
Direct leak of 40 byte(s) in 1 object(s) allocated at:
    #0 0x4b9d70 in operator new(unsigned long)
    #1 0x4011a4 in leakExample() example.cpp:5
    #2 0x401240 in main example.cpp:12
```

Similar info to Valgrind, with clear stack traces for debugging.

### 16.4.12 Summary

Memory leaks remain a serious issue in C++ development, but modern tools and best practices significantly reduce their risk. By:

- Understanding common leak causes,
- Using smart pointers and RAII,
- Employing powerful detection tools like Valgrind and AddressSanitizer,
- Carefully interpreting diagnostic reports,

you can write efficient, leak-free C++ applications that maintain stability and performance over time.

# Chapter 17.

# Multithreading and Concurrency Basics

# 17 Multithreading and Concurrency Basics

## 17.1 Introduction to Threads in C++

In modern software development, programs that perform multiple tasks simultaneously are becoming increasingly common. This capability is known as **concurrency**, and one of its fundamental building blocks is the **thread**. In this section, we will introduce the concept of threads, explain how C++ supports multithreading through the C++11 `<thread>` library, discuss the advantages and challenges of multithreading, and provide simple examples to get you started.

### 17.1.1 What Are Threads and Why Use Them?

A **thread** is the smallest unit of execution within a program. Traditionally, a program runs as a single thread, executing one instruction after another sequentially. However, by using multiple threads, a program can perform multiple operations at the same time.

For example, consider a program that needs to:

- Download files from the internet,
- Process user input,
- Write data to disk.

If these operations are performed one after another, the program may be slow or unresponsive. Using multiple threads allows these tasks to run concurrently, improving performance and responsiveness.

### 17.1.2 Concurrency vs. Parallelism

- **Concurrency** means multiple tasks making progress at the same time. This might happen by quickly switching between tasks on a single CPU core (time slicing).
- **Parallelism** means multiple tasks actually executing simultaneously, often on multiple CPU cores.

Multithreading enables both concurrency and parallelism depending on the system capabilities.

### 17.1.3 Introducing the C11 `thread` Library

Before C++11, thread support was platform-specific and inconsistent. The C++11 standard introduced the `<thread>` library, providing a portable and standardized way to create and manage threads.

### 17.1.4 Creating a Thread

You can create a new thread by constructing an object of the `std::thread` class and passing it a function or callable to run:

Full runnable code:

```cpp
#include <iostream>
#include <thread>

void hello() {
    std::cout << "Hello from thread!\n";
}

int main() {
    std::thread t(hello);  // Start thread executing hello()
    t.join();              // Wait for thread t to finish
    return 0;
}
```

In this example:

- A thread `t` is launched to execute the function `hello()`.
- `t.join()` blocks the main thread until `t` finishes.

### 17.1.5 Thread Lifecycle Basics

A thread typically goes through these stages:

1. **Created**: The `std::thread` object is constructed, starting the thread.
2. **Running**: The thread executes the provided function.
3. **Joined or Detached**: The thread is either joined or detached to manage its lifetime.
4. **Finished**: The thread completes execution.

### 17.1.6 Joining Threads

- The **`join()`** function blocks the calling thread until the target thread completes.
- You must call `join()` or `detach()` on every `std::thread` object before it is destroyed; otherwise, the program will terminate.

### 17.1.7 Detaching Threads

- The **detach()** function lets the thread run independently (in the background).
- Detached threads continue executing after the main thread ends, but you lose control over them.

Example of detaching a thread:

```
std::thread t([] {
    std::cout << "Detached thread running\n";
});
t.detach();
```

### 17.1.8 Advantages of Multithreading

- **Improved Performance:** Utilize multiple CPU cores to run tasks simultaneously.
- **Better Responsiveness:** User interfaces remain active while background tasks run.
- **Resource Sharing:** Threads within the same process share memory, enabling efficient communication.
- **Scalability:** Programs can handle more workload by adding threads.

### 17.1.9 Challenges and Pitfalls of Multithreading

While multithreading offers many benefits, it also introduces challenges:

### 17.1.10 Race Conditions

Occurs when two or more threads access shared data simultaneously and at least one thread modifies it, leading to inconsistent or incorrect results.

Example:

```
int counter = 0;

void increment() {
    for (int i = 0; i < 1000; ++i) {
        ++counter;  // Not thread-safe
    }
}
```

If multiple threads run `increment()` concurrently without synchronization, the final value of `counter` may be incorrect.

### 17.1.11 Deadlocks

Happens when two or more threads wait indefinitely for each other to release resources.

Example scenario:

- Thread A holds lock 1 and waits for lock 2.
- Thread B holds lock 2 and waits for lock 1.

Both wait forever, causing a deadlock.

### 17.1.12 Simple Example: Launching Multiple Threads

Let's create a program that launches multiple threads to print messages:

Full runnable code:

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

void printMessage(int id) {
    std::cout << "Thread " << id << " is running.\n";
}

int main() {
    const int numThreads = 5;
    std::vector<std::thread> threads;

    // Launch threads
    for (int i = 0; i < numThreads; ++i) {
        threads.emplace_back(printMessage, i);
    }

    // Join threads
    for (auto& t : threads) {
        t.join();
    }

    std::cout << "All threads finished.\n";
    return 0;
}
```

Output (order may vary):

```
Thread 0 is running.
Thread 1 is running.
Thread 2 is running.
Thread 3 is running.
Thread 4 is running.
All threads finished.
```

### 17.1.13 Summary

- A **thread** is a single flow of execution within a program.
- C++11 introduced the standardized `<thread>` library for multithreading.
- Threads allow concurrent and parallel execution, improving performance and responsiveness.
- You create a thread by constructing `std::thread` with a callable.
- Threads must be either **joined** or **detached** to manage their lifecycle.
- Multithreading introduces challenges like **race conditions** and **deadlocks** that require careful synchronization.
- Starting with simple thread creation and management prepares you for more complex concurrency programming.

## 17.2 Thread Management and Synchronization

In the previous section, we introduced the concept of threads in C++ and how to create and launch them using the C++11 `<thread>` library. As you work with multithreading, managing threads properly and ensuring safe access to shared resources become critical. This section explores thread management operations like joining and detaching threads, explains the importance of synchronization, introduces synchronization primitives such as mutexes and condition variables, and demonstrates examples to illustrate thread-safe programming and coordination between threads.

### 17.2.1 Thread Management: Joining and Detaching

Once a thread is started using `std::thread`, managing its lifetime correctly is essential to avoid program errors and undefined behavior.

### 17.2.2 Joining Threads

Calling the `join()` method on a thread makes the calling thread (usually the main thread) wait until the target thread completes execution. This ensures that resources associated with the thread are cleaned up properly and the program does not exit prematurely.

```cpp
std::thread t([] {
    std::cout << "Thread running\n";
});
t.join();  // Wait for t to finish
```

If you forget to join a joinable thread before it goes out of scope, the program will terminate with a runtime error.

### 17.2.3 Detaching Threads

The `detach()` method lets a thread run independently in the background. Once detached, you cannot join the thread anymore or directly check if it has finished.

```
std::thread t([] {
    std::cout << "Detached thread running\n";
});
t.detach();
```

Detaching is useful for fire-and-forget tasks, but it requires caution because the thread may continue running even after the main program finishes, which can lead to resource leaks or crashes if not handled properly.

### 17.2.4 Why Synchronization Is Essential

When multiple threads access **shared resources** such as variables, files, or hardware devices, **race conditions** can occur if their access is not controlled. A race condition happens when the outcome depends on the unpredictable timing of thread execution, leading to inconsistent or incorrect results.

For example, consider two threads incrementing the same integer:

```
int counter = 0;

void increment() {
    for (int i = 0; i < 10000; ++i) {
        ++counter;  // Not thread-safe
    }
}
```

If two threads call `increment()` concurrently without synchronization, some increments may be lost because `++counter` involves multiple CPU instructions (read-modify-write) that can be interrupted by another thread.

**Synchronization** mechanisms prevent such data races by controlling how threads access shared data, ensuring only one thread modifies or reads data at a time or coordinating their actions.

### 17.2.5 Mutexes: Mutual Exclusion Locks

A **mutex** (short for mutual exclusion) is a synchronization primitive that ensures only one thread can own the mutex and access the protected resource at a time.

C++ provides `std::mutex` in the `<mutex>` header:

```cpp
#include <iostream>
#include <thread>
#include <mutex>

int counter = 0;
std::mutex mtx;  // Mutex to protect counter

void safe_increment() {
    for (int i = 0; i < 10000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);  // Lock mutex
        ++counter;                              // Critical section
    }
}
```

Here:

- `std::lock_guard` automatically locks the mutex when constructed and releases it when destructed (e.g., at the end of the scope). This ensures exception-safe locking.
- Only one thread can execute the critical section modifying `counter` at a time.

### 17.2.6 Using the mutex directly:

```cpp
mtx.lock();
counter++;
mtx.unlock();
```

However, using `lock()` and `unlock()` manually is error-prone; you risk deadlocks or unlocking errors, so prefer `std::lock_guard` or `std::unique_lock`.

### 17.2.7 Condition Variables: Thread Coordination

While mutexes protect shared data, sometimes threads need to **wait** for certain conditions or events before continuing. **Condition variables** provide a way to block a thread until notified by another thread.

C++ provides `std::condition_variable` for this purpose:

Full runnable code:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void worker() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, []{ return ready; });  // Wait until ready == true
    std::cout << "Worker thread proceeding after signal\n";
}

int main() {
    std::thread t(worker);

    {
        std::lock_guard<std::mutex> lock(mtx);
        ready = true;   // Set condition
    }
    cv.notify_one();    // Notify worker thread

    t.join();
    return 0;
}
```

How it works:

- The worker thread locks the mutex and waits on the condition variable `cv`.
- The `wait()` function releases the mutex and blocks the thread until notified and the condition predicate (`ready == true`) becomes true.
- The main thread sets `ready` to true under mutex protection and calls `notify_one()` to wake the waiting thread.

### 17.2.8 Example: Thread-Safe Counter Using Mutex

Here's a complete example demonstrating multiple threads incrementing a shared counter safely:

Full runnable code:

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>

int counter = 0;
std::mutex mtx;

void increment(int id) {
    for (int i = 0; i < 1000; ++i) {
        std::lock_guard<std::mutex> lock(mtx);
        ++counter;
        // Optionally print:
        // std::cout << "Thread " << id << " incremented counter to " << counter << "\n";
    }
}

int main() {
    const int numThreads = 10;
    std::vector<std::thread> threads;

    for (int i = 0; i < numThreads; ++i) {
        threads.emplace_back(increment, i);
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Final counter value: " << counter << "\n";  // Should be 10000
    return 0;
}
```

### 17.2.9 Summary

- Thread management involves **joining** to wait for thread completion or **detaching** for background execution.
- Synchronization is vital to protect shared resources and avoid race conditions.
- **Mutexes** (`std::mutex`) provide mutual exclusion, allowing only one thread to access a critical section.
- **Lock guards** like `std::lock_guard` and `std::unique_lock` simplify mutex management and prevent errors.
- **Condition variables** (`std::condition_variable`) allow threads to wait for specific conditions and coordinate execution.
- Combining these primitives leads to robust, thread-safe programs.

Mastering thread management and synchronization is the foundation for writing reliable multithreaded C++ programs. In the next section, we will explore **mutexes, locks, and condition variables** in more detail, including common patterns and best practices.

## 17.3 Mutexes, Locks, and Condition Variables

In multithreaded programming, coordinating access to shared resources is crucial to avoid data races, inconsistencies, and undefined behavior. In the previous section, we introduced basic thread management and synchronization concepts. Now, we dive deeper into three essential synchronization primitives in C++: **mutexes**, **locks**, and **condition variables**.

These tools help you manage concurrent access to shared data safely and enable threads to communicate with one another by waiting and signaling specific conditions.

### 17.3.1 Mutexes: Ensuring Mutual Exclusion

A **mutex** (short for *mutual exclusion*) is a synchronization primitive that allows only one thread at a time to own the lock and access a protected resource. This prevents race conditions when multiple threads try to read or modify shared data simultaneously.

### 17.3.2 Using `std::mutex`

C++ provides `std::mutex` in the `<mutex>` header. Here's how to use it:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx;
int shared_data = 0;

void increment() {
    mtx.lock();         // Acquire the mutex lock
    ++shared_data;      // Critical section: safe to access shared data
    mtx.unlock();       // Release the lock
}
```

However, manually locking and unlocking the mutex is error-prone. Forgetting to unlock or exceptions thrown before unlocking can cause deadlocks.

### 17.3.3 Locks: Managing Mutexes Safely and Conveniently

To avoid manual management, C++ provides **lock classes** that automatically acquire and release mutex locks, making your code safer and cleaner.

### 17.3.4 `std::lock_guard`

`std::lock_guard` is a lightweight wrapper that locks a mutex when created and unlocks it when destroyed (typically at the end of a scope).

Example:

```
void safe_increment() {
    std::lock_guard<std::mutex> lock(mtx);  // Locks mutex on creation
    ++shared_data;                           // Safe critical section
}   // Automatically unlocks when lock goes out of scope
```

Advantages:

- Exception-safe: the lock is released even if an exception is thrown.
- Simple syntax, minimal overhead.

### 17.3.5 `std::unique_lock`

`std::unique_lock` is a more flexible lock that allows:

- Locking and unlocking multiple times.
- Deferred locking.
- Moving ownership between locks.

Example:

```
void flexible_increment() {
    std::unique_lock<std::mutex> lock(mtx, std::defer_lock);  // Do not lock immediately
    // ... do some work ...
    lock.lock();    // Lock when ready
    ++shared_data;
    lock.unlock();  // Unlock manually if needed

    // Later, can lock again
    lock.lock();
    // ... another critical section ...
}
```

Use `std::unique_lock` when you need more control over locking/unlocking or want to use condition variables (which require `std::unique_lock`).

### 17.3.6 Condition Variables: Waiting and Signaling Between Threads

Mutexes protect shared data, but sometimes threads need to **wait for a specific condition** before proceeding. For example, a thread might wait until a buffer has data before processing it, or wait until a flag becomes true.

**Condition variables** provide this capability: they allow threads to **block and wait** until notified by another thread that a condition has changed.

### 17.3.7 Using `std::condition_variable`

The condition variable class in C++ is `std::condition_variable`, defined in `<condition_variable>`.

Key methods:

- `wait(std::unique_lock<std::mutex>& lock)`: Blocks the current thread until notified.
- `notify_one()`: Wakes up one waiting thread.
- `notify_all()`: Wakes up all waiting threads.

### 17.3.8 Basic Workflow

- A thread locks a mutex and calls `wait()` on the condition variable, which atomically unlocks the mutex and suspends the thread.
- Another thread changes the condition and calls `notify_one()` or `notify_all()`.
- The waiting thread wakes up, reacquires the mutex, and continues.

### 17.3.9 Example: Producer-Consumer Problem

The producer-consumer problem is a classic example where condition variables shine. One thread produces data and places it in a buffer, and another thread consumes it.

Full runnable code:

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>

std::queue<int> buffer;
const unsigned int maxBufferSize = 10;

std::mutex mtx;
std::condition_variable cv_producer;
std::condition_variable cv_consumer;

bool finished = false;
```

```
void producer() {
    for (int i = 0; i < 20; ++i) {
        std::unique_lock<std::mutex> lock(mtx);
        cv_producer.wait(lock, []{ return buffer.size() < maxBufferSize; });  // Wait if buffer is full

        buffer.push(i);
        std::cout << "Produced: " << i << std::endl;

        lock.unlock();
        cv_consumer.notify_one();  // Notify consumer
    }

    // Notify consumer no more data will be produced
    std::unique_lock<std::mutex> lock(mtx);
    finished = true;
    lock.unlock();
    cv_consumer.notify_all();
}

void consumer() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        cv_consumer.wait(lock, []{ return !buffer.empty() || finished; });  // Wait if buffer empty

        if (!buffer.empty()) {
            int data = buffer.front();
            buffer.pop();
            std::cout << "Consumed: " << data << std::endl;

            lock.unlock();
            cv_producer.notify_one();  // Notify producer space is available
        } else if (finished) {
            break;  // No more data to consume
        }
    }
}

int main() {
    std::thread p(producer);
    std::thread c(consumer);

    p.join();
    c.join();

    return 0;
}
```

### 17.3.10 Explanation

- **Producer** waits if the buffer is full (`buffer.size() >= maxBufferSize`) before pushing data.
- **Consumer** waits if the buffer is empty before popping data.

- Condition variables synchronize the threads so that producers wait for space and consumers wait for data.
- `finished` flag signals the consumer when no more items will be produced.

### 17.3.11 Summary and Best Practices

- Use **`std::mutex`** to protect shared resources and avoid race conditions.
- Prefer **`std::lock_guard`** for simple, exception-safe locking.
- Use **`std::unique_lock`** when you need more control or plan to use condition variables.
- **Condition variables** allow threads to wait efficiently for events or state changes, enabling safe thread coordination.
- Always protect shared data and condition checks with mutexes to avoid race conditions.
- Be cautious about **spurious wakeups**: condition variables may wake even if the condition is not true, so always use a predicate in `wait()` to re-check the condition.
- Avoid deadlocks by locking mutexes in a consistent order and keeping critical sections short.

### 17.3.12 Next Steps

Understanding these synchronization primitives sets the stage for writing correct and efficient multithreaded C++ programs. The next section will provide practical **simple parallel programming examples**, applying what we have learned so far to real-world tasks.

## 17.4 Simple Parallel Programming Examples

Multithreading can significantly improve performance in applications that perform computationally intensive or I/O-bound tasks by running parts of the code concurrently. In this section, we will explore practical examples demonstrating how to use threads in C++ to parallelize simple tasks such as numerical computation and file processing. Along the way, we'll highlight proper synchronization, safe data sharing, and common pitfalls.

### 17.4.1 Example 1: Parallel Summation of an Array

Suppose we want to compute the sum of a large array of integers. We can divide the array into chunks and sum each chunk in a separate thread, then combine the results.

Full runnable code:

```
#include <iostream>
#include <vector>
#include <thread>
#include <numeric> // std::accumulate
#include <mutex>

std::mutex result_mutex;

void partial_sum(const std::vector<int>& data, int start, int end, int& result) {
    int sum = std::accumulate(data.begin() + start, data.begin() + end, 0);

    std::lock_guard<std::mutex> lock(result_mutex);
    result += sum; // Safely update shared result
}

int main() {
    const int dataSize = 10000;
    std::vector<int> data(dataSize, 1); // Fill with 1s
    int result = 0;

    int mid = dataSize / 2;
    std::thread t1(partial_sum, std::cref(data), 0, mid, std::ref(result));
    std::thread t2(partial_sum, std::cref(data), mid, dataSize, std::ref(result));

    t1.join();
    t2.join();

    std::cout << "Total Sum = " << result << std::endl;
    return 0;
}
```

**Explanation**:

- The array is divided into two halves.
- Each thread computes the sum of a half.
- `std::mutex` ensures that updates to the shared `result` variable are thread-safe.

### 17.4.2 Example 2: Concurrent File Line Counting

Let's assume we have multiple files and want to count the number of lines in each concurrently. This is I/O-bound and benefits from multithreading.

Full runnable code:

```
#include <iostream>
#include <fstream>
#include <thread>
#include <vector>
#include <string>
```

```cpp
void count_lines(const std::string& filename) {
    std::ifstream file(filename);
    if (!file) {
        std::cerr << "Cannot open file: " << filename << std::endl;
        return;
    }

    int lines = 0;
    std::string line;
    while (std::getline(file, line)) {
        ++lines;
    }

    std::cout << filename << ": " << lines << " lines\n";
}

int main() {
    std::vector<std::string> files = {"file1.txt", "file2.txt", "file3.txt"};
    std::vector<std::thread> threads;

    for (const auto& file : files) {
        threads.emplace_back(count_lines, file);
    }

    for (auto& t : threads) {
        t.join();
    }

    return 0;
}
```

**Key Takeaways**:

- Each file is processed in its own thread.
- There is no need for synchronization here because threads do not share data.

### 17.4.3 Example 3: Thread-Safe Logging

When multiple threads log messages to **std::cout**, outputs can become interleaved and unreadable. Using a mutex ensures clean output.

Full runnable code:

```cpp
#include <iostream>
#include <thread>
#include <mutex>

std::mutex cout_mutex;

void log_message(int thread_id) {
    for (int i = 0; i < 5; ++i) {
        std::lock_guard<std::mutex> lock(cout_mutex);
        std::cout << "Thread " << thread_id << " - Message " << i << std::endl;
    }
}

int main() {
    std::thread t1(log_message, 1);
    std::thread t2(log_message, 2);

    t1.join();
    t2.join();

    return 0;
}
```

This guarantees that each line is printed atomically by one thread, avoiding jumbled output.

### 17.4.4 Performance Considerations

While multithreading can enhance performance, it's not always a free win:

- **Thread creation overhead**: Creating too many threads for small tasks can actually slow down the program.
- **False sharing**: When threads modify adjacent memory, cache performance may degrade.
- **Context switching**: Excessive thread switching can reduce CPU efficiency.
- **Load balancing**: Distribute work evenly across threads.

Use **thread pools** or libraries like **OpenMP** or **std::async** for more scalable solutions in larger programs.

### 17.4.5 Common Pitfalls

1. **Race Conditions**: Occur when multiple threads access and modify shared data unsafely. Always use synchronization primitives like `std::mutex`.

2. **Deadlocks**: Happen when two or more threads wait for each other's resources. Avoid nested locking or use `std::lock()` to lock multiple mutexes safely.

3. **Resource leaks**: Forgetting to `join()` or `detach()` threads can lead to resource leaks or undefined behavior.

4. **Non-determinism**: Thread scheduling is non-deterministic, so bugs may only appear occasionally. Always test thoroughly.

### 17.4.6 Conclusion

These simple examples illustrate how C++ threads can be used to speed up tasks like computation and file processing. As a beginner, focus on:

- Using threads to divide clearly separable tasks.
- Ensuring shared data is protected with mutexes.
- Keeping code simple and readable.

As you grow more comfortable, you'll explore more advanced concurrency tools and design patterns to write scalable and robust multithreaded applications.

In the next chapters, we'll further explore the Standard Library and other advanced C++ programming topics that complement concurrent programming effectively.

# Chapter 18.

# Lambda Functions and Functional Programming Features

# 18 Lambda Functions and Functional Programming Features

## 18.1 Defining and Using Lambdas

Modern C++ (starting with C++11) introduced **lambda expressions**, also known as **lambda functions**, to support functional programming techniques. Lambdas provide a concise way to define **anonymous functions**—functions without names—that can be created inline, passed as arguments, and used wherever function objects or callbacks are needed.

In this section, we'll explore the syntax of lambdas, explain their components, and demonstrate how they simplify many programming tasks such as filtering and sorting.

### 18.1.1 What Is a Lambda?

A **lambda expression** is an unnamed function defined using a special syntax. Unlike regular functions, lambdas are defined in place—right where they're needed—and typically used for short-lived functionality, such as one-time callbacks or in conjunction with Standard Template Library (STL) algorithms.

**Basic Syntax:**

```
[capture](parameter_list) -> return_type {
    // function body
};
```

- `capture`: Specifies which variables from the surrounding scope are accessible inside the lambda.
- `parameter_list`: Specifies the parameters just like a normal function.
- `return_type`: (Optional) Specifies the return type; can usually be inferred.
- `function body`: Contains the code to execute.

### 18.1.2 A Simple Lambda Example

Here's a basic lambda that adds two numbers:

Full runnable code:

```
#include <iostream>

int main() {
    auto add = [](int a, int b) {
        return a + b;
    };
    std::cout << "Sum: " << add(3, 4) << std::endl;  // Output: Sum: 7
    return 0;
}
```

- `auto` is used because the exact type of a lambda is compiler-generated.
- The lambda is assigned to the variable `add` and can be used like a regular function.

### 18.1.3 Lambda with No Parameters

A lambda can also take no arguments:

```
auto sayHello = []() {
    std::cout << "Hello, World!" << std::endl;
};

sayHello();
```

This lambda takes no parameters and prints a greeting.

### 18.1.4 Capturing Variables

Lambdas can capture variables from their surrounding scope. There are several ways to do this:

#### Capture by Value ([])

```
int x = 10;
auto printX = [=]() {
    std::cout << "x = " << x << std::endl;
};
printX();
```

The variable `x` is **captured by value**, meaning the lambda makes a copy. Modifications inside the lambda do not affect the original `x`.

#### Capture by Reference ([&])

```
int x = 10;
auto modifyX = [&]() {
    x += 5;
};
modifyX();
std::cout << "x = " << x << std::endl;  // Output: x = 15
```

Here, `x` is **captured by reference**, so changes inside the lambda affect the original variable.

### Explicit Capture

You can explicitly specify how individual variables are captured:

```
int a = 5, b = 10;
auto mixedCapture = [a, &b]() {
    // a is captured by value, b by reference
};
```

## 18.1.5 Specifying Return Types

Lambdas can return values, and most of the time, the compiler can deduce the return type. However, if needed, you can specify it using `-> return_type` syntax:

```
auto multiply = [](int x, int y) -> int {
    return x * y;
};
```

This is useful when returning different types based on conditions, or when returning `auto` would be ambiguous.

## 18.1.6 Lambdas in Practice: Sorting and Filtering

Lambda functions shine when used with STL algorithms. Let's look at some common use cases.

### Sorting a Vector

Full runnable code:

```
#include <vector>
#include <algorithm>
#include <iostream>

int main() {
    std::vector<int> numbers = {4, 2, 7, 1, 9};

    std::sort(numbers.begin(), numbers.end(), [](int a, int b) {
        return a < b;  // ascending
    });

    for (int n : numbers) {
        std::cout << n << " ";
    }
    return 0;
}
```

Here, a lambda is passed directly to `std::sort` to define a custom comparison.

**Filtering with `std::remove_if`**

```
#include <algorithm>

std::vector<int> data = {1, 2, 3, 4, 5, 6};
data.erase(
    std::remove_if(data.begin(), data.end(), [](int n) {
        return n % 2 == 0;  // remove even numbers
    }),
    data.end()
);
```

The lambda identifies elements to remove—in this case, even numbers.

### 18.1.7 Conclusion

Lambda functions provide a powerful and elegant way to write concise, readable, and expressive code, especially in combination with STL algorithms. Whether you're sorting, filtering, or transforming data, lambdas help eliminate the need for verbose function declarations or custom functor classes.

As you continue learning C++, you'll find lambdas invaluable in writing cleaner and more modern code. In the next section, we'll dive deeper into **captures and mutable lambdas**, expanding your ability to manipulate and control lambda behavior efficiently.

## 18.2 Captures and Mutable Lambdas

In the previous section, we learned the basics of lambda expressions and how they can be used as inline, anonymous functions. One of the most powerful aspects of lambdas in C++ is their ability to **capture variables** from the surrounding scope. This allows a lambda to work with data that exists outside of its own body, making it highly flexible and convenient.

In this section, we'll explore how **captures** work in C++, including capturing by value and by reference, and how the `mutable` keyword enables modification of captured values in lambdas.

### 18.2.1 What Is a Capture?

A **capture** allows a lambda to use variables defined outside its scope. These captured variables are defined in the lambda's **capture list**, which appears in square brackets (`[]`) before the parameter list.

### 18.2.2 Capture by Value (`[]` or `[x]`)

When a variable is captured **by value**, the lambda receives a **copy** of the variable. Any modification to this copy inside the lambda does **not** affect the original variable.

```cpp
int count = 5;

auto showCount = [=]() {
    std::cout << "Count: " << count << std::endl;
};

showCount();  // Output: Count: 5
```

In the example above, `count` is captured by value (`[=]`), meaning the lambda holds a snapshot of `count` at the time the lambda is defined. Even if `count` is modified later in the outer scope, the lambda's internal copy remains unchanged.

### 18.2.3 Capture by Reference (`[&]` or `[&x]`)

Capturing a variable **by reference** allows the lambda to access and **modify** the original variable directly.

```cpp
int count = 5;

auto increment = [&]() {
    count++;
};

increment();
std::cout << "Count: " << count << std::endl;  // Output: Count: 6
```

In this case, the lambda modifies `count` directly because it was captured by reference (`[&]`).

### 18.2.4 Mixed Captures

You can combine captures by value and by reference:

```cpp
int a = 10, b = 20;

auto lambda = [a, &b]() {
    // a is captured by value; b is by reference
    std::cout << "a = " << a << ", b = " << b << std::endl;
    // b can be modified; a cannot
};

lambda();
```

This flexibility allows precise control over how lambdas access external variables.

### 18.2.5 Default Capture Modes

You can specify default capture modes:

- `[=]` — Capture all external variables used in the lambda by **value**
- `[&]` — Capture all external variables used in the lambda by **reference**

You can override the default for specific variables:

```
auto lambda = [=, &x]() { /* x by reference, others by value */ };
auto lambda2 = [&, y]() { /* y by value, others by reference */ };
```

### 18.2.6 The `mutable` Keyword

By default, lambdas that capture variables **by value** cannot modify them. This is because the lambda's `operator()` is implicitly `const`.

To allow modification of a value-captured variable **within the lambda**, you must use the `mutable` keyword:

```
int x = 10;

auto changeX = [=]() mutable {
    x += 5;  // okay because of 'mutable'
    std::cout << "Inside lambda: x = " << x << std::endl;
};

changeX();
std::cout << "Outside lambda: x = " << x << std::endl;  // x remains 10
```

Even though `x` was modified inside the lambda, the change does **not** affect the outer `x` because it was captured **by value**. The `mutable` keyword simply allows the lambda to modify its internal copy.

### 18.2.7 Capturing `this`

In a member function, lambdas can capture the current object by capturing `this`. This allows access to the member variables and functions inside the lambda:

```
class MyClass {
    int value = 42;

public:
    void printValue() {
        auto lambda = [this]() {
            std::cout << "Value = " << value << std::endl;
        };
        lambda();
    }
};
```

Capturing `this` is equivalent to capturing all members of the class by reference.

### 18.2.8 Practical Example: Counter with Mutable Lambda

Full runnable code:

```
#include <iostream>
#include <functional>

std::function<int()> makeCounter() {
    int count = 0;
    return [count]() mutable {
        return ++count;
    };
}

int main() {
    auto counter = makeCounter();

    std::cout << counter() << std::endl;  // Output: 1
    std::cout << counter() << std::endl;  // Output: 2
}
```

This shows a lambda that maintains its own internal state using `mutable`, allowing a captured value to change across invocations.

### 18.2.9 Summary

Lambdas in C++ become much more powerful and expressive when used with **captures**. Capturing by value or reference allows you to control how external variables are accessed, while the `mutable` keyword enables modification of captured values. Understanding these mechanics is essential for writing clean, efficient lambda expressions, especially in combination with the STL.

In the next section, we'll explore how lambdas work seamlessly with **STL algorithms**, making your code more expressive and less error-prone.

## 18.3 Using Lambdas with STL Algorithms

One of the most powerful features introduced in modern C++ (starting from C++11) is the ability to define **lambda expressions**, which are concise, inline, anonymous functions. While we've explored how lambdas work in isolation, their real strength shines when used with the Standard Template Library (STL) algorithms. These algorithms often take function pointers, functors, or lambdas as arguments to define **custom behavior**.

In this section, we'll demonstrate how lambdas simplify working with algorithms such as `std::for_each`, `std::sort`, and `std::transform`, replacing verbose function objects with expressive and maintainable code.

### 18.3.1 Why Use Lambdas with STL?

Before lambdas, customizing behavior in algorithms typically required defining functors (structs with `operator()`) or passing function pointers, both of which add unnecessary verbosity. With lambdas, you can define the behavior directly where it is used, improving **readability**, **locality of logic**, and **maintenance**.

Let's explore practical examples.

### 18.3.2 Example 1: Iterating with `std::for_each`

The `std::for_each` algorithm applies a given function to every element in a range. Here's how you might use it with a lambda:

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5};

    std::for_each(nums.begin(), nums.end(), [](int n) {
        std::cout << n << " ";
    });

    std::cout << std::endl;
    return 0;
}
```

Output:

```
1 2 3 4 5
```

The lambda `[](int n) { std::cout << n << " "; }` replaces a separate function or functor, allowing the behavior to remain inline with the algorithm call.

### 18.3.3 Example 2: Sorting with `std::sort`

The `std::sort` algorithm sorts a range. By default, it sorts in ascending order using `operator<`. With lambdas, you can easily define custom sorting criteria.

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> nums = {5, 2, 8, 1, 3};

    // Sort in descending order using a lambda
    std::sort(nums.begin(), nums.end(), [](int a, int b) {
        return a > b;
    });

    for (int n : nums) {
        std::cout << n << " ";
    }

    std::cout << std::endl;
    return 0;
}
```

Output:

```
8 5 3 2 1
```

With lambdas, you don't need to write a separate comparator function elsewhere in your code, keeping logic localized and intuitive.

### 18.3.4 Example 3: Transforming Data with `std::transform`

`std::transform` applies a transformation to each element in a range and stores the result elsewhere. Lambdas make it effortless to define the transformation inline.

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
int main() {
    std::vector<int> nums = {1, 2, 3};
    std::vector<int> squared(nums.size());

    std::transform(nums.begin(), nums.end(), squared.begin(), [](int x) {
        return x * x;
    });

    for (int n : squared) {
        std::cout << n << " ";
    }

    std::cout << std::endl;
    return 0;
}
```

Output:

```
1 4 9
```

The lambda `[ ](int x) { return x * x; }` defines a square operation that is immediately understandable and doesn't require creating a separate named function.

### 18.3.5 Example 4: Filtering Elements Using `std::copy_if`

You can also use lambdas with algorithms like `std::copy_if` to filter data based on conditions.

Full runnable code:

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> nums = {1, 5, 10, 15, 20};
    std::vector<int> filtered;

    std::copy_if(nums.begin(), nums.end(), std::back_inserter(filtered), [](int x) {
        return x > 10;
    });

    for (int n : filtered) {
        std::cout << n << " ";
    }

    std::cout << std::endl;
    return 0;
}
```

Output:

```
15 20
```

This shows how lambda expressions can make filtering operations compact and clear.

### 18.3.6 Lambda Benefits Recap

Using lambdas with STL algorithms offers several key benefits:

- **Readability**: The logic is close to where it's used, avoiding context-switching.
- **Conciseness**: Replaces multiple lines of boilerplate code.
- **Flexibility**: Lambdas can capture external variables for use in algorithm logic.
- **Performance**: Lambdas can often be inlined by the compiler for efficient execution.

### 18.3.7 Capture Use in Algorithms

Lambdas can capture external variables to adapt behavior based on runtime values.

```
int threshold = 10;

std::copy_if(nums.begin(), nums.end(), std::back_inserter(filtered), [threshold](int x) {
    return x > threshold;
});
```

Here, `threshold` is captured by value. You can also use `[&threshold]` to capture by reference if the variable should be modifiable.

### 18.3.8 Conclusion

Lambdas greatly enhance the power and usability of STL algorithms in modern C++. They allow for **in-place** customization of algorithm behavior, leading to cleaner, shorter, and more maintainable code. As you write more C++ code, especially when working with containers and algorithms, lambdas will become an essential tool in your toolbox.

In the next section, we'll explore how `std::function` and callbacks further extend functional programming capabilities in C++.

## 18.4 `std::function` and Callbacks

Modern C++ programming often requires working with functions as first-class objects—passing them as parameters, storing them, or invoking them dynamically. To support this style, C++ provides `std::function`, a powerful and flexible **type-erased function wrapper**. It can hold anything callable with a specific signature, including **function pointers**, **functors**, and **lambda expressions**.

This section introduces `std::function`, shows how it enables flexible and decoupled program designs through **callbacks**, and provides practical examples of its usage in event handling and asynchronous operations.

### 18.4.1 What is `std::function`?

`std::function` is a class template in the `<functional>` header that can store and invoke **any callable entity** that matches a given signature. This includes:

- Normal functions
- Lambda expressions
- Function pointers
- Functor objects (objects with `operator()`)

The syntax to declare an `std::function` looks like this:

```
#include <functional>

std::function<int(int, int)> myFunc;
```

This line declares `myFunc` as a callable object that takes two `int` arguments and returns an `int`.

### 18.4.2 Basic Example: Storing and Calling a Lambda

Full runnable code:

```
#include <iostream>
#include <functional>

int main() {
    std::function<int(int, int)> add = [](int a, int b) {
        return a + b;
    };

    std::cout << "Sum: " << add(5, 3) << std::endl;
    return 0;
}
```

**Output:**

```
Sum: 8
```

This shows how a lambda is assigned to an `std::function` and invoked just like a regular function.

### 18.4.3 Callback Functions

Callbacks are functions passed to another function to be **called at a later point**, often when an event occurs or an operation completes. `std::function` makes implementing such callbacks simple and clean.

#### Example: Callback Mechanism

Full runnable code:

```cpp
#include <iostream>
#include <functional>

void performOperation(int x, int y, std::function<void(int)> callback) {
    int result = x + y;
    callback(result);
}

int main() {
    auto printResult = [](int value) {
        std::cout << "Operation result: " << value << std::endl;
    };

    performOperation(4, 6, printResult);
    return 0;
}
```

**Output:**

```
Operation result: 10
```

Here, the `performOperation` function accepts a callback that is called with the result. This pattern is common in UI event handling, network programming, and task scheduling.

### 18.4.4 Using Function Pointers with `std::function`

`std::function` can also hold traditional function pointers:

Full runnable code:

```cpp
#include <iostream>
#include <functional>

int multiply(int a, int b) {
    return a * b;
}
```

```cpp
int main() {
    std::function<int(int, int)> operation = multiply;
    std::cout << "Product: " << operation(4, 5) << std::endl;
    return 0;
}
```

**Output:**

```
Product: 20
```

This shows how existing functions can be seamlessly integrated into a modern interface using `std::function`.

### 18.4.5 Capturing State with Functors

Functors are objects that act like functions via the `operator()` member. `std::function` supports them as well.

Full runnable code:

```cpp
#include <iostream>
#include <functional>

struct Divider {
    int divisor;

    Divider(int d) : divisor(d) {}

    int operator()(int x) const {
        return x / divisor;
    }
};

int main() {
    Divider divideBy2(2);
    std::function<int(int)> divide = divideBy2;

    std::cout << "10 / 2 = " << divide(10) << std::endl;
    return 0;
}
```

**Output:**

```
10 / 2 = 5
```

This is especially useful when callbacks require stored state or configuration.

### 18.4.6 Use Case: Event System

Let's implement a simplified event subscription system using `std::function`.

Full runnable code:

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

class Button {
public:
    std::vector<std::function<void()>> listeners;

    void onClick(std::function<void()> callback) {
        listeners.push_back(callback);
    }

    void click() {
        for (auto& func : listeners) {
            func();
        }
    }
};

int main() {
    Button btn;

    btn.onClick([]() {
        std::cout << "Button clicked! (Handler 1)" << std::endl;
    });

    btn.onClick([]() {
        std::cout << "Button clicked! (Handler 2)" << std::endl;
    });

    btn.click();
    return 0;
}
```

**Output:**

```
Button clicked! (Handler 1)
Button clicked! (Handler 2)
```

This flexible mechanism is commonly used in GUI frameworks, gaming engines, and custom event systems.

### 18.4.7 When to Use `std::function`

`std::function` should be used when:

- You need to store or pass around **any kind of callable** with a consistent signature.

- You want to **decouple** implementation from behavior (e.g., plugins, strategy patterns).
- You're building **event systems** or **asynchronous callbacks**.
- You want a uniform interface for different types of callables.

Note: `std::function` introduces a small runtime overhead due to type erasure. If maximum performance is critical (such as in tight loops), consider alternatives like templates or inline lambdas.

### 18.4.8 Conclusion

`std::function` is a cornerstone of modern C++ programming that brings **flexibility**, **type safety**, and **elegance** to function handling. It enables callbacks, decouples APIs, and integrates seamlessly with lambdas, functors, and function pointers. By mastering `std::function`, you unlock powerful patterns for writing clean, extensible, and modular C++ code.

In the next chapter, we'll explore file I/O techniques for persistent data storage in C++ applications.

# Chapter 19.

# Advanced Class Features

1. Copy and Move Constructors and Assignment Operators
2. The Rule of Three/Five/Zero
3. Nested Classes and Friend Functions
4. Namespaces and Using Directives

# 19 Advanced Class Features

## 19.1 Copy and Move Constructors and Assignment Operators

In C++, **copy constructors**, **copy assignment operators**, **move constructors**, and **move assignment operators** are special member functions that control how objects are **copied** or **moved**. Understanding when and how to define them is critical for writing efficient, bug-free C++ programs, especially when dealing with dynamically allocated resources.

This section explains what each of these functions does, how to declare and implement them, and when you should provide custom versions to ensure proper behavior.

### 19.1.1 Copy Constructor and Copy Assignment Operator

#### What They Are

The **copy constructor** creates a new object as a copy of an existing object. The **copy assignment operator** assigns one existing object to another existing object.

C++ automatically generates both by default, performing **shallow copy** (bitwise copy of member variables). However, if your class manages resources (like dynamic memory), you must implement **deep copy logic**.

#### Syntax

```
// Copy constructor
ClassName(const ClassName& other);

// Copy assignment operator
ClassName& operator=(const ClassName& other);
```

#### Example: Deep Copy with Dynamic Memory

Full runnable code:

```
#include <iostream>
#include <cstring>

class String {
private:
    char* data;

public:
    // Constructor
    String(const char* str = "") {
        data = new char[strlen(str) + 1];
        strcpy(data, str);
    }
    // Copy constructor
    String(const String& other) {
        data = new char[strlen(other.data) + 1];
        strcpy(data, other.data);
    }

    // Copy assignment operator
    String& operator=(const String& other) {
        if (this != &other) {
            delete[] data;
            data = new char[strlen(other.data) + 1];
            strcpy(data, other.data);
        }
        return *this;
    }

    // Destructor
    ~String() {
        delete[] data;
    }

    void print() const {
        std::cout << data << std::endl;
    }
};

int main() {
    String s1("Hello");
    String s2 = s1;     // Copy constructor
    String s3;
    s3 = s1;            // Copy assignment
    s2.print();
    s3.print();
    return 0;
}
```

### 19.1.2 Move Constructor and Move Assignment Operator

#### What They Are

C++11 introduced **move semantics** to enable efficient resource transfers. The **move constructor** and **move assignment operator** transfer ownership of resources from one object to another **without duplicating data**.

They're essential when working with **temporary objects** or **large data structures** (like vectors, strings, or files) that would be expensive to copy.

#### Syntax

```
// Move constructor
ClassName(ClassName&& other);
```

```
// Move assignment operator
ClassName& operator=(ClassName&& other);
```

Note the use of **rvalue references** (`&&`), which signify that `other` can be "moved from."

### Example: Move Semantics in Action

Full runnable code:

```
#include <iostream>
#include <cstring>

class String {
private:
    char* data;

public:
    // Constructor
    String(const char* str = "") {
        data = new char[strlen(str) + 1];
        strcpy(data, str);
    }

    // Move constructor
    String(String&& other) noexcept {
        data = other.data;
        other.data = nullptr;  // Prevent deletion
    }

    // Move assignment operator
    String& operator=(String&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            other.data = nullptr;
        }
        return *this;
    }

    // Destructor
    ~String() {
        delete[] data;
    }

    void print() const {
        std::cout << (data ? data : "(null)") << std::endl;
    }
};

int main() {
    String s1("World");
    String s2 = std::move(s1);  // Move constructor
    s2.print();                 // World
    s1.print();                 // (null)
    return 0;
}
```

### 19.1.3 When Are These Functions Called?

| Operation | Function Called |
|---|---|
| `ClassName obj2 = obj1;` | Copy constructor |
| `obj2 = obj1;` | Copy assignment operator |
| `ClassName obj = std::move(obj);` | Move constructor |
| `obj2 = std::move(obj1);` | Move assignment operator |

### 19.1.4 Best Practices

1. **Always clean up**: When writing assignment operators, **release existing resources** before acquiring new ones.
2. **Check self-assignment**: Use `if (this != &other)` to avoid copying an object into itself.
3. **Use `noexcept`** for move operations: This allows STL containers to optimize performance.
4. **Use `std::move()`**: It converts lvalues to rvalues explicitly, enabling move semantics.
5. **Favor move over copy**: When performance matters, and the object owns resources, provide move logic.

### 19.1.5 Rule of Three and Five

If your class manages dynamic resources, follow the **Rule of Three**:

- Define:
  - Destructor
  - Copy constructor
  - Copy assignment operator

With C++11 move semantics, this extends to the **Rule of Five**:

- Also define:
  - Move constructor
  - Move assignment operator

If your class doesn't need custom copy/move behavior (i.e., no raw pointers), let the compiler generate the defaults—this is the **Rule of Zero**.

### 19.1.6 Summary

In this section, you've learned how to implement and use copy and move constructors and assignment operators. Copy operations create deep copies of objects, while move operations enable fast transfers of resources. Understanding when these are called and how to define them properly is key to writing efficient, safe, and modern C++ code.

In the next section, we'll discuss the Rule of Three, Five, and Zero to help you decide which special functions to implement based on your class design.

## 19.2 The Rule of Three/Five/Zero

In C++, classes often manage resources such as memory, file handles, or sockets. When a class handles such resources, it must ensure proper allocation, copying, moving, and deallocation. To do this safely and efficiently, C++ developers follow the **Rule of Three**, the **Rule of Five**, or in modern designs, the **Rule of Zero**.

These "rules" guide you in deciding whether to manually define special member functions such as destructors, copy constructors, copy assignment operators, move constructors, and move assignment operators.

### 19.2.1 The Rule of Three

The **Rule of Three** comes from pre-C++11 days. It states:

> If your class **needs** a user-defined **destructor**, it probably also needs a user-defined **copy constructor** and **copy assignment operator**.

This is because any of these three typically implies the class owns a resource (like dynamic memory), and shallow copies (provided by default) will lead to **double deletion**, **memory leaks**, or **undefined behavior**.

#### Special Member Functions

1. **Destructor** – Cleans up resources (e.g., calls `delete` on a pointer).
2. **Copy Constructor** – Called when creating a new object from another object (e.g., `MyClass b = a;`).
3. **Copy Assignment Operator** – Called when assigning one object to another (e.g., `b = a;`).

#### Example

```
#include <iostream>
#include <cstring>
```

```cpp
class Buffer {
private:
    char* data;

public:
    // Constructor
    Buffer(const char* str = "") {
        data = new char[strlen(str) + 1];
        strcpy(data, str);
    }

    // Destructor
    ~Buffer() {
        delete[] data;
    }

    // Copy Constructor
    Buffer(const Buffer& other) {
        data = new char[strlen(other.data) + 1];
        strcpy(data, other.data);
    }

    // Copy Assignment Operator
    Buffer& operator=(const Buffer& other) {
        if (this != &other) {
            delete[] data;
            data = new char[strlen(other.data) + 1];
            strcpy(data, other.data);
        }
        return *this;
    }

    void print() const {
        std::cout << data << std::endl;
    }
};
```

If you omit the copy constructor or assignment operator here, a **shallow copy** would occur, and both objects would point to the same memory—leading to a double delete on destruction.

### 19.2.2 The Rule of Five

C++11 introduced **move semantics**, which adds two more special member functions:

1. **Move Constructor** – Transfers resources from a temporary or expired object.
2. **Move Assignment Operator** – Similar to move constructor, but for assignment.

So the **Rule of Five** extends the Rule of Three:

> If your class needs one of the five special functions (destructor, copy/move constructor, copy/move assignment), it likely needs **all five**.

**Example**

Let's extend the `Buffer` class to include move operations:

```
class Buffer {
private:
    char* data;

public:
    Buffer(const char* str = "") {
        data = new char[strlen(str) + 1];
        strcpy(data, str);
    }

    ~Buffer() {
        delete[] data;
    }

    Buffer(const Buffer& other) {
        data = new char[strlen(other.data) + 1];
        strcpy(data, other.data);
    }

    Buffer& operator=(const Buffer& other) {
        if (this != &other) {
            delete[] data;
            data = new char[strlen(other.data) + 1];
            strcpy(data, other.data);
        }
        return *this;
    }

    // Move constructor
    Buffer(Buffer&& other) noexcept {
        data = other.data;
        other.data = nullptr;
    }

    // Move assignment operator
    Buffer& operator=(Buffer&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            other.data = nullptr;
        }
        return *this;
    }
};
```

This ensures both **copying** and **moving** work correctly and safely.

### 19.2.3 The Rule of Zero

The **Rule of Zero** is a design principle that encourages writing classes that do not **own resources directly**, thus eliminating the need for user-defined copy/move constructors or destructors.

Instead, resource management is **delegated** to well-designed standard types such as:

- `std::vector`
- `std::string`
- `std::unique_ptr`
- `std::shared_ptr`

If your class only uses these standard types, the compiler-generated special functions are usually correct and sufficient.

**Example: Rule of Zero in Practice**

```
#include <string>

class Person {
private:
    std::string name;
    int age;

public:
    Person(const std::string& n, int a) : name(n), age(a) {}

    void print() const {
        std::cout << name << " is " << age << " years old.\n";
    }
};
```

Here, `std::string` manages its own memory. The compiler provides a destructor, copy/move constructors, and assignment operators that "just work." You don't need to write them manually—making the code safer and easier to maintain.

### 19.2.4 Summary of the Rules

| Rule | When to Use |
|---|---|
| **Rule of Three** | Class manages resources (e.g., dynamic memory) |
| **Rule of Five** | Class manages resources and must support moving |
| **Rule of Zero** | Class delegates all resource management to members like `std::vector`, `std::unique_ptr`, etc. |

### 19.2.5 Best Practices

- Favor **Rule of Zero**: Prefer using standard containers and smart pointers.

- If you must manage resources manually, follow **Rule of Five**.
- Always implement **destructor**, **copy**, and **move** together if needed.
- Use `= delete` to disable copy or move operations explicitly.
- Use `noexcept` in move constructors for better performance in STL containers.

**Example: Deleting Copy**

```
class NonCopyable {
public:
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;
};
```

This is useful when copying is unsafe or logically incorrect (e.g., for file handles or thread objects).

### 19.2.6 Conclusion

The Rule of Three, Five, and Zero help you write safer, cleaner C++ code that handles object lifetimes correctly. If your class owns resources, define the appropriate special functions. But when possible, let standard library types manage those resources and follow the Rule of Zero.

Understanding and applying these rules will prevent memory leaks, double deletions, and subtle bugs—and make your C++ code more modern and maintainable.

## 19.3 Nested Classes and Friend Functions

As C++ programs grow in size and complexity, developers often need to organize code more logically and maintain encapsulation while still allowing controlled interaction between tightly coupled components. Two powerful tools that C++ provides for this purpose are **nested classes** and **friend functions** (or friend classes). This section introduces both concepts, explains their syntax and use cases, and provides practical examples.

### 19.3.1 Nested Classes

A **nested class** is a class defined within another class. It is often used when the inner class is conceptually subordinate to or closely associated with the outer class. Nested classes can access the private and protected members of the outer class if declared as `friend`, but by default, they do not have this access. The outer class does not automatically have access to the private members of the nested class either.

### Syntax

```cpp
class Outer {
public:
    class Inner {
    public:
        void display();
    };
};
```

You can then create an object of `Inner` using:

```cpp
Outer::Inner obj;
```

### Use Case: Logical Grouping

Nested classes are helpful when you want to encapsulate a helper or utility class within a larger class, and that helper has no use outside the context of its containing class.

### Example

Full runnable code:

```cpp
#include <iostream>

class Vehicle {
public:
    class Engine {
    public:
        void start() {
            std::cout << "Engine started.\n";
        }
    };

    void run() {
        Engine engine;
        engine.start();
        std::cout << "Vehicle is running.\n";
    }
};

int main() {
    Vehicle car;
    car.run();

    // You can also create Engine directly
    Vehicle::Engine engine;
    engine.start();

    return 0;
}
```

In this example, the `Engine` class is only meaningful in the context of a `Vehicle`, so nesting it makes logical sense.

### 19.3.2 Friend Functions

A **friend function** is a non-member function that is granted access to the private and protected members of a class. Although not a member of the class, a friend function is declared inside the class definition using the `friend` keyword.

#### Syntax

```
class MyClass {
private:
    int secret;

public:
    MyClass() : secret(42) {}

    friend void reveal(const MyClass& obj);
};

void reveal(const MyClass& obj) {
    std::cout << "Secret: " << obj.secret << std::endl;
}
```

#### Use Case: External Utility Access

Friend functions are useful when a function that isn't a member needs intimate access to the class's internal state. This is common in operator overloading and serialization.

### 19.3.3 Friend Classes

Just like functions, entire classes can also be declared as `friend`, allowing them full access to the private and protected members of the class that grants friendship.

#### Syntax

```
class A;

class B {
public:
    void accessA(const A& a);
};

class A {
private:
    int data = 100;

    friend class B; // B can now access A's private members
};

void B::accessA(const A& a) {
    std::cout << "Accessing A's private data: " << a.data << std::endl;
}
```

### Use Case: Tight Class Cooperation

Friend classes are helpful when two classes are tightly coupled and need direct access to each other's internals without exposing everything to the outside world.

### 19.3.4 Practical Example: Nested Class with Friend Access

Let's combine both concepts to create a secure messaging system:

Full runnable code:

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

class SecureBox {
private:
    std::string password = "secret123";

public:
    class Key {
    public:
        void unlock(const SecureBox& box) {
            std::cout << "Access granted. Password: " << box.password << std::endl;
        }
    };

    friend class Key; // Allow Key to access private members
};

int main() {
    SecureBox::Key masterKey;
    SecureBox safe;

    masterKey.unlock(safe);  // Accesses SecureBox::password

    return 0;
}
```

Here, `Key` is a nested class of `SecureBox` and is declared as a friend. It can therefore access `SecureBox`'s private members, demonstrating both **nesting** and **friendship** in action.

### 19.3.5 Guidelines and Best Practices

- **Use nested classes** for logical grouping when an inner class is only relevant to the outer class.
- **Avoid excessive friendship**. Granting too many friend functions or classes access to private members can weaken encapsulation.
- **Use `friend` deliberately**, especially when operator overloading or serialization logic

benefits from it.

- **Keep coupling low**, unless the relationship between the classes or functions justifies a friend declaration.

### 19.3.6 Summary

Nested classes and friend functions/classes are advanced features that offer powerful mechanisms for structuring and controlling access in C++ programs. Nested classes allow tight logical grouping, while friend functions and classes enable safe exceptions to the usual access rules. When used judiciously, they contribute to clean, maintainable, and expressive code architectures.

## 19.4 Namespaces and Using Directives

In larger C++ projects, it's common to have multiple developers working on different parts of the code. As the codebase grows, the likelihood of name collisions increases—where two functions, classes, or variables have the same name but different meanings. To handle this problem, C++ provides *namespaces*, a feature that allows developers to group logically related code under a unique identifier. This helps organize the code, avoid conflicts, and improve readability.

### 19.4.1 What Is a Namespace?

A **namespace** is a declarative region that provides a scope to the identifiers inside it. These identifiers can include variables, functions, classes, and objects. When you define a namespace, everything inside it is qualified with the namespace's name, avoiding name conflicts in the global scope.

#### Basic Syntax

Here's how you define and use a namespace:

Full runnable code:

```
#include <iostream>

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

```
int main() {
    int result = MathUtils::add(3, 4);
    std::cout << "Result: " << result << std::endl;
    return 0;
}
```

In this example, the `add` function is defined inside the `MathUtils` namespace. To call it, we must qualify it with the namespace name: `MathUtils::add`.

### 19.4.2 Nested Namespaces

C++ also allows *nested namespaces*, which means you can define one namespace inside another. This is useful for further categorization, especially in large-scale applications.

Full runnable code:

```
#include <iostream>

namespace Company {
    namespace Finance {
        void generateReport() {
            std::cout << "Generating financial report..." << std::endl;
        }
    }
}

int main() {
    Company::Finance::generateReport();
    return 0;
}
```

Starting with C++17, you can simplify nested namespace declarations:

```
namespace Company::Finance {
    void generateReport() {
        std::cout << "Generating financial report..." << std::endl;
    }
}
```

### 19.4.3 Anonymous (Unnamed) Namespaces

An **anonymous namespace** is a namespace without a name. Its contents have internal linkage, meaning they are limited to the translation unit (typically, a single `.cpp` file). This can be useful when you want to limit the scope of a helper function or variable to the file.

```
namespace {
    void logInternalEvent() {
        std::cout << "Internal event logged." << std::endl;
    }
}

int main() {
    logInternalEvent(); // No namespace prefix needed
    return 0;
}
```

### 19.4.4 Using Declarations and Directives

To simplify access to identifiers within a namespace, C++ offers two mechanisms: **using declarations** and **using directives**.

#### Using Declaration

A *using declaration* introduces a single name from a namespace into the current scope.

Full runnable code:

```
#include <iostream>

namespace Utility {
    void printMessage() {
        std::cout << "Utility message" << std::endl;
    }
}

int main() {
    using Utility::printMessage;
    printMessage(); // Now accessible without prefix
    return 0;
}
```

This is safe and limits the risk of name collisions because only one name is brought into the current scope.

#### Using Directive

A *using directive* brings **all** the names from a namespace into the current scope.

Full runnable code:

```
#include <iostream>

namespace Utility {
    void printMessage() {
        std::cout << "Utility message" << std::endl;
    }

    void logEvent() {
        std::cout << "Logging event" << std::endl;
    }
}

int main() {
    using namespace Utility;
    printMessage();
    logEvent();
    return 0;
}
```

While convenient, this can be dangerous in larger programs because it may lead to name collisions, especially when multiple namespaces contain functions or types with the same names.

### 19.4.5 Best Practices

Here are some guidelines for using namespaces effectively in real-world C++ projects:

- **Avoid `using namespace` in header files.** This can unintentionally bring names into every file that includes the header, leading to hard-to-debug conflicts.
- **Prefer using declarations over directives.** They are more precise and reduce the risk of name collisions.
- **Use namespaces to group logically related code.** For example, `Math`, `Network`, or `Graphics` namespaces can help indicate the function of the code.
- **Limit the use of `using namespace std;`.** Though common in small programs and tutorials, avoid this in larger applications or headers. Instead, qualify names like `std::cout` and `std::vector`.

### 19.4.6 Practical Example

Let's create a small example that uses multiple namespaces:

Full runnable code:

```
#include <iostream>

namespace Math {
    int square(int x) {
        return x * x;
    }
}

namespace Geometry {
    const double PI = 3.14159;

    double circleArea(double radius) {
        return PI * Math::square(radius);
    }
}

int main() {
    double area = Geometry::circleArea(5.0);
    std::cout << "Circle area: " << area << std::endl;
    return 0;
}
```

In this program, `Math::square` is used inside `Geometry::circleArea`, and both are organized under meaningful namespaces. This improves modularity and avoids cluttering the global namespace with generic function names like `square`.

Namespaces are an essential feature in C++ that help keep large projects manageable and free from naming conflicts. By understanding how to define and use them properly, you'll write cleaner, more organized, and scalable C++ code.

# Chapter 20.

# Modern C Features Overview (C11/14/17/20)

# 20 Modern C Features Overview (C11/14/17/20)

## 20.1 Auto Type Deduction

Modern C++ emphasizes cleaner, more maintainable code without sacrificing type safety or performance. One of the key features introduced in C++11 to help achieve this goal is the `auto` keyword. It allows the compiler to automatically deduce the type of a variable from its initializer. This reduces verbosity, especially when dealing with complex or templated types, and can improve code clarity when used appropriately.

### 20.1.1 What Is `auto`?

The `auto` keyword tells the compiler to *infer the type* of a variable at compile time based on the value used to initialize it. This means you no longer need to explicitly specify a type if the initializer already makes the type clear.

**Basic Syntax**

```
auto x = 10;             // x is deduced as int
auto y = 3.14;           // y is deduced as double
auto message = "Hi";     // message is deduced as const char*
```

In each case, the compiler examines the initializer and determines the correct type. The resulting variable behaves as if you had written the explicit type.

### 20.1.2 Why Use `auto`?

There are several benefits to using `auto` in modern C++:

- **Reduced Typing:** Saves effort and minimizes repetition, especially with long type names.
- **Improved Readability:** Reduces clutter, particularly when the type is obvious from context.
- **Code Maintenance:** If the type of an expression changes, you may not need to update the variable declaration.
- **Simplified Templates and Lambdas:** Makes code more readable when working with templated types or lambda expressions.

### 20.1.3 auto with Complex and Templated Types

When working with containers from the Standard Template Library (STL), type names can become quite verbose. `auto` is especially helpful in these scenarios.

**Without auto:**

```
std::vector<std::pair<int, std::string>>::iterator it = myVector.begin();
```

**With auto:**

```
auto it = myVector.begin();
```

Both lines accomplish the same thing, but the version using `auto` is shorter and easier to read.

### 20.1.4 auto in Range-Based For Loops

C++11 introduced range-based `for` loops, and `auto` complements them perfectly:

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

for (auto num : numbers) {
    std::cout << num << " ";
}
```

If you're modifying the elements or want to avoid unnecessary copying, use references:

```
for (auto& num : numbers) {
    num *= 2; // modify in place
}
```

You can even use `const auto&` for read-only access:

```
for (const auto& num : numbers) {
    std::cout << num << " ";
}
```

### 20.1.5 auto and Functions

While you can't use `auto` to declare a function parameter or a function variable without initialization, you *can* use `auto` as a return type in functions (from C++14 onwards):

**C14 Return Type Deduction**

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

The compiler deduces the return type based on the return expression.

### 20.1.6 `auto` and Pointers

You can use `auto` with pointers and references. The type deduced by `auto` includes whether the initializer is a pointer, a reference, or a constant.

```
int x = 42;
int* ptr = &x;

auto p1 = ptr;      // p1 is int*
auto& r1 = x;       // r1 is int&
const int cx = 100;
auto y = cx;        // y is int (not const)
const auto& z = cx; // z is const int&
```

Be cautious: `auto` *drops* `const` *qualifiers* when copying. If you need to preserve `const`, use `const auto`.

### 20.1.7 Limitations of `auto`

While `auto` is powerful, it's not always the right tool:

1. **No Initialization = Error**

```
auto x;  // Error: auto requires an initializer
```

The compiler needs the initializer to deduce the type.

2. **Can Reduce Code Clarity**

```
auto x = someComplexFunction();  // What is the type of x?
```

If the function name doesn't clearly indicate what it returns, this can make code harder to understand.

3. **Loss of Explicitness**

Sometimes it's important to *see* the type, especially for API boundaries or public interfaces. Overusing `auto` may obscure what types are being used.

### 20.1.8 Best Practices

To make the most of `auto` without sacrificing code clarity, follow these best practices:

- **Use `auto` when the type is obvious** from the right-hand side (e.g., `auto x = 5.0;`).

- **Use `auto` to avoid long or complex types**, especially with iterators or STL algorithms.
- **Avoid `auto` in public APIs** or where type clarity is important.
- **Prefer `const auto&`** in range-based loops when iterating over collections of complex objects you don't want to copy.
- **Be explicit when necessary** to maintain clarity for future readers of your code.

### 20.1.9 Conclusion

The `auto` keyword is one of the cornerstones of modern C++. It simplifies variable declarations, reduces boilerplate, and improves maintainability when used wisely. While it's tempting to replace all type declarations with `auto`, doing so indiscriminately can hurt readability. Use `auto` where it enhances clarity and simplifies complex types, but don't hesitate to write out the type when it improves understanding. Like all powerful features in C++, balance is key.

## 20.2 Range-Based For Loops

In C++, iterating over elements in arrays, containers, or other sequences is a common task. Traditionally, this has been done using classic `for` loops with counters or iterators, which can be verbose and error-prone. To simplify iteration and make code cleaner and safer, C++11 introduced the **range-based for loop** — a language construct that enables straightforward traversal of any container or array.

### 20.2.1 What Is a Range-Based For Loop?

A *range-based for loop* allows you to iterate over every element in a sequence without explicitly managing indices or iterators. It abstracts away the complexity of traversing containers and makes the code easier to write and read.

#### Basic Syntax

```
for (declaration : expression) {
    // use declaration (the element) inside the loop
}
```

- **declaration:** declares a variable that will hold each element of the sequence during iteration.
- **expression:** represents the container, array, or any range you want to iterate over.

### 20.2.2 Why Use Range-Based For Loops?

Before range-based for loops, iterating over a vector looked like this:

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

for (size_t i = 0; i < nums.size(); ++i) {
    std::cout << nums[i] << " ";
}
```

Or using iterators:

```
for (auto it = nums.begin(); it != nums.end(); ++it) {
    std::cout << *it << " ";
}
```

Both are correct, but involve manual management of indices or iterators, which increases the chance of bugs such as off-by-one errors or invalid iterator use.

With a range-based for loop, the same iteration becomes:

```
for (int num : nums) {
    std::cout << num << " ";
}
```

This is much cleaner, easier to read, and less error-prone.

### 20.2.3 Examples with STL Containers

The range-based for loop works with all standard containers that provide `begin()` and `end()` iterators.

Full runnable code:

```
#include <iostream>
#include <vector>
#include <string>
#include <map>

int main() {
    std::vector<std::string> fruits = {"apple", "banana", "cherry"};

    for (const auto& fruit : fruits) {
        std::cout << fruit << " ";
    }
    std::cout << std::endl;

    std::map<int, std::string> idToName = {
        {1, "Alice"},
        {2, "Bob"},
        {3, "Charlie"}
    };

    for (const auto& pair : idToName) {
        std::cout << pair.first << ": " << pair.second << std::endl;
    }

    return 0;
}
```

Output:

```
apple banana cherry
1: Alice
2: Bob
3: Charlie
```

- Notice the use of `const auto&` in the loop variable. This avoids copying the elements (which could be expensive for complex objects) and protects against accidental modification.

### 20.2.4 Using Range-Based For with Raw Arrays

Range-based for loops also work with built-in arrays:

```
int arr[] = {10, 20, 30, 40, 50};

for (int value : arr) {
    std::cout << value << " ";
}
```

Output:

```
10 20 30 40 50
```

This provides a safer and more concise alternative to:

```
for (int i = 0; i < 5; ++i) {
    std::cout << arr[i] << " ";
}
```

Here, you don't have to hardcode the size or risk accessing invalid indices.

### 20.2.5 How Does It Work Under the Hood?

The compiler translates a range-based for loop roughly into something like this:

```
{
    auto&& __range = expression;                    // evaluates the container or array
    auto __begin = std::begin(__range);             // gets iterator to first element
    auto __end = std::end(__range);                 // gets iterator to end
    for (; __begin != __end; ++__begin) {
        declaration = *__begin;                      // extracts the element
        // loop body
    }
}
```

- `std::begin` and `std::end` are used to retrieve iterators or pointers, which means the range-based for loop works with any type that supports these functions.
- For raw arrays, `std::begin` and `std::end` return pointers to the first and one-past-last elements.
- For user-defined types, you can customize `begin()` and `end()` functions to enable range-based for loop support.

### 20.2.6 When Are Range-Based For Loops Most Useful?

- **Iterating over all elements without needing the index.** When the index isn't required, range-based loops improve clarity.
- **Preventing off-by-one errors.** Because you don't manage the loop counter or iterators explicitly.
- **Improving maintainability.** If the container changes, there is no need to update loop conditions.
- **Simplifying code with complex container types.** No need to write verbose iterator types or complex indexing logic.

### 20.2.7 Modifying Elements in a Range-Based For Loop

By default, the loop variable is a copy of the element. To modify elements in place, use references:

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

for (auto& num : nums) {
    num *= 2;  // double each number in the vector
}

for (auto num : nums) {
    std::cout << num << " ";
}
```

Output:

```
2 4 6 8
```

If you only want to read elements without modification, use `const auto&` to avoid unnecessary copying:

```
for (const auto& num : nums) {
    std::cout << num << " ";
}
```

### 20.2.8 Limitations and When Not to Use Range-Based For Loops

- **When you need the element index.** Range-based for loops don't provide the index, so if you need it, you may still have to use a traditional for loop.
- **When skipping or iterating partially.** If you need to start in the middle or skip elements conditionally, a range-based loop might be less straightforward.
- **When modifying the container during iteration.** Inserting or erasing elements while iterating requires care and often explicit iterator management.

### 20.2.9 Summary

Range-based for loops are a powerful and elegant tool introduced in C++11 that make iterating over containers and arrays simpler and safer. They eliminate boilerplate code related to loop counters and iterators, reducing the chance of errors and improving code readability.

By embracing range-based loops, you write code that is:

- **Clear:** The intent to iterate is immediately obvious.
- **Safe:** No risk of out-of-bounds errors from incorrect indices.
- **Concise:** Less verbose, easier to maintain.

As you progress in C++, range-based for loops will become a fundamental part of your toolkit for clean and expressive iteration.

## 20.3 constexpr and Constexpr If

Modern C++ continually evolves to empower developers to write more efficient, safer, and expressive code. Two powerful features introduced in recent standards—`constexpr` and `constexpr if`—enable computations and decisions to happen at compile time, resulting in optimized and robust programs. This section explores these features and shows how they improve both regular and template-based programming.

### 20.3.1 What Is `constexpr`?

The keyword `constexpr` was introduced in C++11 and has since been enhanced in newer standards. It marks variables and functions that can be evaluated **at compile time**. This means the compiler computes their value during compilation rather than at runtime, often leading to faster and safer code.

### 20.3.2 `constexpr` Variables

Declaring a variable as `constexpr` makes it a compile-time constant:

```cpp
constexpr int maxSize = 100;
```

Here, `maxSize` is a constant known during compilation. Unlike `const`, `constexpr` guarantees that the value can be used in compile-time contexts such as array sizes, template parameters, or other `constexpr` expressions.

Example:

```cpp
constexpr int arrayLength = 10;
int arr[arrayLength];  // valid in C++11 and later
```

### 20.3.3 `constexpr` Functions

Functions marked with `constexpr` can be evaluated by the compiler if all their arguments are compile-time constants. This enables writing logic that can produce compile-time values, improving performance and safety.

Example:

```cpp
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : (n * factorial(n - 1));
}

int main() {
    constexpr int fact5 = factorial(5);  // computed at compile time
    int arr[fact5];  // array size is 120
}
```

In this example, `factorial(5)` is computed during compilation. This prevents any runtime cost and can catch errors early.

**Important notes:**

- From C++11, `constexpr` functions had limitations (one return statement, no loops).
- C++14 relaxed these constraints, allowing loops, multiple statements, and local variables inside `constexpr` functions.

- To enable compile-time evaluation, arguments and all used functions must themselves be `constexpr`.

### 20.3.4 Benefits of `constexpr`

- **Performance:** Calculations done at compile time reduce runtime overhead.
- **Safety:** Errors can be caught during compilation rather than at runtime.
- **Expressiveness:** Enables more expressive code by allowing constant expressions to be computed by functions, not just simple literals.
- **Template Metaprogramming:** Compile-time functions enhance template programming by enabling more flexible and powerful template logic.

### 20.3.5 Introducing `constexpr if` (C17)

C++17 introduced `constexpr if`, a new form of compile-time conditional branching. Unlike regular `if` statements, which evaluate at runtime, `constexpr if` lets the compiler choose which branch to compile based on a constant expression evaluated during compilation.

**Syntax**

```
if constexpr (condition) {
    // code compiled if condition is true
} else {
    // code compiled if condition is false
}
```

If the `condition` is false, the compiler **completely ignores** the corresponding branch, even if it contains code that would normally cause a compile error.

### 20.3.6 Why Is `constexpr if` Useful?

`constexpr if` dramatically simplifies template programming and enables more readable, efficient code that depends on compile-time properties.

Before C++17, you might use **SFINAE** (Substitution Failure Is Not An Error) or template specialization to select different code paths depending on types. This could lead to verbose and hard-to-read code.

With `constexpr if`, you can write cleaner templates that contain conditional code that only compiles when needed.

### 20.3.7 Example: Using `constexpr if` in Templates

Suppose you want to write a function template that behaves differently for integral and floating-point types:

Full runnable code:

```cpp
#include <iostream>
#include <type_traits>

template <typename T>
void process(T value) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << value << " is an integral type.\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << value << " is a floating-point type.\n";
    } else {
        std::cout << value << " is of some other type.\n";
    }
}

int main() {
    process(42);        // integral
    process(3.14);      // floating-point
    process("hello");   // other type
}
```

Output:

```
42 is an integral type.
3.14 is a floating-point type.
hello is of some other type.
```

In this example:

- The `if constexpr` checks are evaluated at compile time.
- Only the branch that matches the type is compiled; the others are discarded.
- This avoids compile errors from invalid code in discarded branches, making template code safer and more readable.

### 20.3.8 Practical Use Case: `constexpr` and `constexpr if` Together

Combining `constexpr` functions and `constexpr if` lets you write flexible and efficient code with decisions made at compile time.

Full runnable code:

```cpp
#include <iostream>
#include <type_traits>

constexpr int square(int x) {
    return x * x;
}

template <typename T>
constexpr T compute(T value) {
    if constexpr (std::is_integral_v<T>) {
        return square(value);
    } else {
        return value * 2;
    }
}

int main() {
    constexpr int i = compute(5);        // 25 (square)
    double d = compute(3.14);             // 6.28 (double the value)

    std::cout << "Int result: " << i << "\n";
    std::cout << "Double result: " << d << "\n";
}
```

- For integral types, `compute` squares the input at compile time.
- For floating-point types, it doubles the value at runtime.

### 20.3.9 Summary

- **`constexpr`** lets you define variables and functions whose values can be computed at compile time, enabling faster and safer code.
- **`constexpr if`** (from C++17) introduces compile-time conditional branching that simplifies template code by compiling only the necessary branches.
- Together, they allow writing expressive, efficient, and maintainable code, especially in generic programming.
- Proper use of these features leads to early error detection, optimized executables, and clearer intent in your programs.

Mastering `constexpr` and `constexpr if` is an essential step toward writing modern, high-performance C++ code that leverages the compiler's ability to do work ahead of time.

## 20.4 Structured Bindings and Concepts (Intro)

C++ continues to evolve, introducing features that make code more expressive, readable, and easier to maintain. Two important additions in recent standards are **structured bindings** (introduced in C++17) and **concepts** (introduced in C++20). Structured bindings simplify working with tuples, pairs, and structs by allowing unpacking into individual variables. Concepts, on the other hand, provide a clear and expressive way to specify template requirements, improving template programming clarity and error messages.

### 20.4.1 Structured Bindings: Unpacking Made Easy

Before C++17, when working with data structures like tuples, pairs, or structs, you often had to access individual members through verbose syntax:

Full runnable code:

```cpp
#include <iostream>
#include <tuple>

int main() {
    std::tuple<int, double, char> data(42, 3.14, 'a');

    // Accessing elements before C++17
    int i = std::get<0>(data);
    double d = std::get<1>(data);
    char c = std::get<2>(data);

    std::cout << i << ", " << d << ", " << c << std::endl;

    return 0;
}
```

While this works, it is verbose and error-prone—especially with larger tuples or complex structs.

#### Enter Structured Bindings

C++17 introduced *structured bindings*, a syntax that allows you to unpack a tuple, pair, array, or struct directly into separate variables in a single, concise statement.

```cpp
auto [i, d, c] = data;
std::cout << i << ", " << d << ", " << c << std::endl;
```

Here, the compiler automatically creates variables `i`, `d`, and `c` initialized with the corresponding elements from `data`.

### 20.4.2 How Structured Bindings Work

You declare multiple variables inside square brackets `[]`, and initialize them with an expression on the right side that returns a structured object.

This syntax works with:

- **`std::tuple` and `std::pair`:** Unpack elements directly.
- **Arrays:** Bind each element to a variable.
- **User-defined structs/classes:** If the class has public members or suitable accessors, the binding unpacks them in order.

Example with an array:

```
int arr[] = {10, 20, 30};
auto [x, y, z] = arr;
std::cout << x << " " << y << " " << z << std::endl;
```

Example with a struct:

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

Point p{3, 4};
auto [a, b] = p;
std::cout << a << ", " << b << std::endl;
```

### 20.4.3 Benefits of Structured Bindings

- **Cleaner Code:** Eliminates repetitive and verbose element access.
- **Less Error-Prone:** Avoids hard-coded indices like `get<0>`, which can lead to bugs.
- **More Readable:** Variable names clearly describe each component.
- **Works with Different Types:** Consistent syntax for tuples, pairs, arrays, and structs.

### 20.4.4 Concepts: Expressive Template Requirements

Templates in C++ provide powerful generic programming capabilities, but traditional template syntax can be difficult to understand and debug. Template constraints were usually enforced indirectly via `static_assert` or SFINAE (Substitution Failure Is Not An Error), which can lead to cryptic errors.

C++20 introduced **Concepts**—a way to specify **requirements on template parameters explicitly and clearly**.

### 20.4.5 What Are Concepts?

A *concept* is a predicate that checks if a type satisfies certain properties or operations. By applying concepts to template parameters, you tell the compiler exactly what requirements the template expects.

This improves code clarity, helps with compile-time checks, and generates clearer error messages.

### 20.4.6 Defining and Using Concepts

Here is a simple example defining a concept that checks if a type supports addition:

Full runnable code:

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

template<typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

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

int main() {
    std::cout << add(3, 4) << std::endl;                          // OK: int
    std::cout << add(std::string("a"), std::string("b")) << std::endl; // OK: std::string

    // Uncommenting the line below causes a compile-time error (as expected):
    // std::cout << add("a", "b") << std::endl; // Error: const char* not Addable

    return 0;
}
```

- `Addable` is a concept that requires a type `T` to support the `+` operator returning a type convertible to `T`.
- The function `add` uses the concept as a template constraint, meaning it will only compile for types satisfying `Addable`.

### 20.4.7 Concepts vs. Traditional Template Programming

Before concepts, you might write:

```cpp
template <typename T>
T add(T a, T b) {
    static_assert(std::is_arithmetic_v<T>, "T must be arithmetic");
    return a + b;
}
```

While `static_assert` catches errors, it does so inside the function body, and error messages can be long or unclear. Concepts move these checks to the template interface, improving expressiveness.

### 20.4.8 Practical Example: Concepts with Structured Bindings

Concepts can work hand-in-hand with structured bindings and other modern features:

Full runnable code:

```cpp
#include <concepts>
#include <iostream>
#include <tuple>

template<typename T>
concept TupleLike = requires {
    std::tuple_size<T>::value; // Ensure tuple_size<T> is defined
};

template <TupleLike T>
void printFirstElement(const T& t) {
    std::cout << std::get<0>(t) << std::endl;
}

int main() {
    std::tuple<int, double, char> t{1, 2.5, 'a'};
    printFirstElement(t);  // prints 1
}
```

(Note: The ellipsis ... here is conceptual for illustration; structured bindings require exact matching variables.)

### 20.4.9 Summary

- **Structured bindings** let you unpack tuples, pairs, arrays, and structs into named variables succinctly, making code clearer and less error-prone.
- **Concepts** provide a new way to specify template parameter requirements explicitly, improving template code readability and compiler diagnostics.
- Both features help modernize C++ code, making it easier to write, understand, and maintain.

By mastering structured bindings and concepts, you unlock the power to write clean, expressive, and robust generic code — a key skill for any modern C++ programmer.

# Chapter 21.

# Project: Console-Based Bank Management System

1. Classes and Object-Oriented Design
2. File Storage and Persistence
3. Basic Exception Handling and User Input Validation

# 21 Project: Console-Based Bank Management System

## 21.1 Classes and Object-Oriented Design

Building a console-based bank management system provides an excellent opportunity to apply Object-Oriented Programming (OOP) principles. At the heart of this project are core entities such as **Bank Accounts**, **Customers**, and **Transactions**. Modeling these real-world concepts using classes allows us to organize code cleanly, encapsulate data, and reuse logic through inheritance.

### 21.1.1 Core OOP Principles in Bank Management

Before diving into code, let's recall some OOP principles important for designing our system:

- **Encapsulation:** Bundling data (attributes) and operations (methods) that manipulate the data into a single unit (class). It protects data by exposing only necessary interfaces.
- **Inheritance:** Enables new classes to derive properties and behavior from existing ones, promoting code reuse.
- **Abstraction:** Hiding complex implementation details behind simple interfaces.
- **Responsibility Assignment:** Each class should have a clear responsibility, focusing on a specific part of the system.

### 21.1.2 Identifying Core Classes

Our bank system will revolve around these classes:

1. **Customer**: Represents a bank customer, storing personal details and the accounts they own.
2. **BankAccount**: Abstract base class that defines common account operations like deposit and withdrawal.
3. **Derived Account Types**: Such as **SavingsAccount** and **CheckingAccount**, inheriting from `BankAccount` and adding specific rules.
4. **Transaction**: Records deposits, withdrawals, and other account activities.

### 21.1.3 Modeling the Customer Class

The `Customer` class stores essential customer details and maintains a list of accounts owned.

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

class BankAccount;  // Forward declaration

class Customer {
private:
    std::string name;
    std::string address;
    std::string phoneNumber;
    std::vector<std::shared_ptr<BankAccount>> accounts;

public:
    Customer(const std::string& name, const std::string& address, const std::string& phone)
        : name(name), address(address), phoneNumber(phone) {}

    void addAccount(std::shared_ptr<BankAccount> account) {
        accounts.push_back(account);
    }

    void displayAccounts() const;

    // Getters
    const std::string& getName() const { return name; }
};
```

Here:

- We use `std::shared_ptr` to manage accounts dynamically, allowing multiple owners or references if needed.
- The `addAccount` method links accounts to the customer.
- `displayAccounts` will show summaries of all owned accounts (we'll implement it after defining `BankAccount`).

### 21.1.4 Designing the BankAccount Base Class

`BankAccount` models general account properties and behaviors, such as balance and basic operations.

```cpp
class BankAccount {
protected:
    int accountNumber;
    double balance;

public:
    BankAccount(int accNum, double initialBalance)
        : accountNumber(accNum), balance(initialBalance) {}

    virtual ~BankAccount() = default;

    virtual void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            std::cout << "Deposited $" << amount << ". New balance: $" << balance << "\n";
        } else {
            std::cout << "Invalid deposit amount.\n";
        }
    }

    virtual void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            std::cout << "Withdrew $" << amount << ". Remaining balance: $" << balance << "\n";
        } else {
            std::cout << "Invalid withdrawal amount or insufficient funds.\n";
        }
    }

    double getBalance() const { return balance; }
    int getAccountNumber() const { return accountNumber; }

    virtual void displayAccountInfo() const {
        std::cout << "Account Number: " << accountNumber << ", Balance: $" << balance << "\n";
    }
};
```

Key points:

- `deposit` and `withdraw` enforce simple validation and update the balance.
- The class uses **protected** members to allow derived classes access while keeping attributes hidden from outside code.
- Methods are marked `virtual` allowing derived classes to override behavior if needed.
- A virtual destructor ensures proper cleanup for derived classes.

### 21.1.5 Extending with Derived Account Types

Different account types can have specific rules. For example, a **SavingsAccount** might enforce minimum balance requirements or interest calculation, while a **CheckingAccount** could allow overdraft.

```
class SavingsAccount : public BankAccount {
private:
    double interestRate;  // e.g., 0.02 for 2%

public:
    SavingsAccount(int accNum, double initialBalance, double rate)
        : BankAccount(accNum, initialBalance), interestRate(rate) {}

    void applyInterest() {
        double interest = balance * interestRate;
        deposit(interest);
        std::cout << "Interest of $" << interest << " applied.\n";
    }
```

```cpp
    void displayAccountInfo() const override {
        std::cout << "Savings Account - ";
        BankAccount::displayAccountInfo();
        std::cout << "Interest Rate: " << interestRate * 100 << "%\n";
    }
};
```

```cpp
class CheckingAccount : public BankAccount {
private:
    double overdraftLimit;

public:
    CheckingAccount(int accNum, double initialBalance, double overdraft)
        : BankAccount(accNum, initialBalance), overdraftLimit(overdraft) {}

    void withdraw(double amount) override {
        if (amount > 0 && balance + overdraftLimit >= amount) {
            balance -= amount;
            std::cout << "Withdrew $" << amount << ". New balance: $" << balance << "\n";
        } else {
            std::cout << "Withdrawal exceeds overdraft limit or invalid amount.\n";
        }
    }

    void displayAccountInfo() const override {
        std::cout << "Checking Account - ";
        BankAccount::displayAccountInfo();
        std::cout << "Overdraft Limit: $" << overdraftLimit << "\n";
    }
};
```

These derived classes:

- Override methods to customize behavior (e.g., overdraft logic in `CheckingAccount`).
- Add new data members related to their account type.
- Maintain the base class interface, allowing polymorphic use.

### 21.1.6 Linking Customers and Accounts

Now, let's implement `Customer::displayAccounts` to show all accounts owned by a customer:

```cpp
void Customer::displayAccounts() const {
    std::cout << "Accounts for " << name << ":\n";
    for (const auto& account : accounts) {
        account->displayAccountInfo();
        std::cout << "-------------------\n";
    }
}
```

This illustrates encapsulation and delegation: the `Customer` class delegates account-specific display logic to each account.

### 21.1.7 Example Usage

Here is a simple program that creates customers, accounts, and performs some transactions:

```cpp
int main() {
    // Create customers
    Customer alice("Alice Johnson", "123 Maple St", "555-1234");
    Customer bob("Bob Smith", "456 Oak Rd", "555-5678");

    // Create accounts
    auto aliceSavings = std::make_shared<SavingsAccount>(1001, 1000.0, 0.02);
    auto bobChecking = std::make_shared<CheckingAccount>(2001, 500.0, 200.0);

    // Link accounts to customers
    alice.addAccount(aliceSavings);
    bob.addAccount(bobChecking);

    // Transactions
    aliceSavings->deposit(200);
    aliceSavings->applyInterest();
    aliceSavings->withdraw(150);

    bobChecking->withdraw(600);  // Allowed due to overdraft
    bobChecking->withdraw(200);  // Exceeds overdraft, denied

    // Display accounts
    alice.displayAccounts();
    bob.displayAccounts();

    return 0;
}
```

### 21.1.8 Output

```
Deposited $200. New balance: $1200
Deposited $24. New balance: $1224
Withdrew $150. Remaining balance: $1074
Withdrew $600. New balance: $-100
Withdrawal exceeds overdraft limit or invalid amount.
Accounts for Alice Johnson:
Savings Account - Account Number: 1001, Balance: $1074
Interest Rate: 2%
--------------------
Accounts for Bob Smith:
Checking Account - Account Number: 2001, Balance: $-100
Overdraft Limit: $200
--------------------
```

### 21.1.9 Summary

- Use **classes** to model real-world entities like customers and accounts.
- **Encapsulation** protects data and exposes only necessary operations.
- **Inheritance** enables specialized account types to reuse and extend common functionality.
- Model relationships by linking customers and accounts through pointers or references.
- Define clear responsibilities: Customers manage personal info and accounts, accounts manage balance and transactions.

This design lays a strong foundation for extending the bank system with file storage, transaction history, and input validation — all built on robust object-oriented principles.

## 21.2 File Storage and Persistence

A key feature of any practical bank management system is the ability to **save data persistently** between program runs. Without persistence, all customer information, accounts, and transactions would be lost once the program terminates. This section explains how to use file input/output (I/O) in C++ to store and retrieve bank data, focusing on simple yet effective serialization and deserialization techniques, along with error handling to ensure file integrity.

### 21.2.1 Why File Storage?

In-memory data structures like vectors and classes are transient. When your program ends, their contents vanish. To keep track of customers and accounts over time, you must:

- Write data to files before exiting.
- Read data back when the program starts.

This is called **persistence**, and file storage is the most straightforward method for console applications.

### 21.2.2 File I/O Basics in C

C++ provides file I/O through the `<fstream>` library:

- `std::ofstream` for writing files.
- `std::ifstream` for reading files.
- `std::fstream` for both reading and writing.

Files can be opened in text or binary modes. For simplicity, this project uses **text files** with a structured format to serialize data.

### 21.2.3 Designing a Simple Data Format

For persistence, your data must be converted into a storable format — **serialization** — and then restored — **deserialization**.

A common approach is to write each customer and their accounts as plain text lines with fields separated by delimiters (e.g., commas or spaces).

Example format:

```
Customer|Alice Johnson|123 Maple St|555-1234
Account|Savings|1001|1074|0.02
Account|Checking|2001|-100|200
Customer|Bob Smith|456 Oak Rd|555-5678
Account|Checking|3001|500|100
```

Each customer starts with a line beginning with `Customer|` followed by their info, then one or more `Account|` lines with account details.

### 21.2.4 Writing Data to a File

Let's add serialization methods to our classes.

```cpp
#include <fstream>

// In Customer class
void save(std::ofstream& outFile) const {
    outFile << "Customer|" << name << "|" << address << "|" << phoneNumber << "\n";
    for (const auto& account : accounts) {
        account->save(outFile);
    }
}
```

For `BankAccount` and derived classes, implement a virtual `save` function:

```cpp
class BankAccount {
public:
    virtual void save(std::ofstream& outFile) const = 0;  // pure virtual
    virtual ~BankAccount() = default;
};

class SavingsAccount : public BankAccount {
public:
    void save(std::ofstream& outFile) const override {
        outFile << "Account|Savings|" << accountNumber << "|" << balance << "|" << interestRate << "\n"
    }
};
```

```
class CheckingAccount : public BankAccount {
public:
    void save(std::ofstream& outFile) const override {
        outFile << "Account|Checking|" << accountNumber << "|" << balance << "|" << overdraftLimit << "
    }
};
```

Then, save all customers and their accounts in the main program:

```
void saveAllData(const std::vector<Customer>& customers, const std::string& filename) {
    std::ofstream outFile(filename);
    if (!outFile) {
        std::cerr << "Error opening file for writing: " << filename << "\n";
        return;
    }
    for (const auto& customer : customers) {
        customer.save(outFile);
    }
    std::cout << "Data saved successfully.\n";
}
```

### 21.2.5 Reading Data from a File

Deserialization reads the file, interprets each line, and reconstructs customers and accounts.

```
#include <sstream>

bool loadAllData(std::vector<Customer>& customers, const std::string& filename) {
    std::ifstream inFile(filename);
    if (!inFile) {
        std::cerr << "Error opening file for reading: " << filename << "\n";
        return false;
    }

    std::string line;
    Customer* currentCustomer = nullptr;

    while (std::getline(inFile, line)) {
        std::istringstream iss(line);
        std::string type;
        if (!std::getline(iss, type, '|')) continue;

        if (type == "Customer") {
            std::string name, address, phone;
            if (!std::getline(iss, name, '|') ||
                !std::getline(iss, address, '|') ||
                !std::getline(iss, phone)) {
                std::cerr << "Malformed Customer line: " << line << "\n";
                continue;
            }
            customers.emplace_back(name, address, phone);
            currentCustomer = &customers.back();

        } else if (type == "Account" && currentCustomer) {
            std::string accountType, accNumStr, balanceStr, extraStr;
            if (!std::getline(iss, accountType, '|') ||
                !std::getline(iss, accNumStr, '|') ||
                !std::getline(iss, balanceStr, '|') ||
                !std::getline(iss, extraStr)) {
                std::cerr << "Malformed Account line: " << line << "\n";
                continue;
            }

            int accNum = std::stoi(accNumStr);
            double balance = std::stod(balanceStr);

            if (accountType == "Savings") {
                double interestRate = std::stod(extraStr);
                auto account = std::make_shared<SavingsAccount>(accNum, balance, interestRate);
                currentCustomer->addAccount(account);

            } else if (accountType == "Checking") {
                double overdraftLimit = std::stod(extraStr);
                auto account = std::make_shared<CheckingAccount>(accNum, balance, overdraftLimit);
                currentCustomer->addAccount(account);
            }
        }
    }
    std::cout << "Data loaded successfully.\n";
    return true;
}
```

### 21.2.6 Error Handling and File Integrity

When dealing with file I/O, always consider:

- **File availability:** Check if the file opens successfully.
- **Format correctness:** Validate that lines have the expected format and number of fields.
- **Robust parsing:** Handle exceptions like `std::stoi` or `std::stod` failures gracefully.
- **Data consistency:** For example, ensure that accounts belong to an existing customer.

To enhance reliability, you can:

- Use try-catch blocks around parsing functions.
- Add versioning to your files to handle future format changes.
- Use checksums or hashes for file integrity if needed.

Example of parsing with exception handling:

```
try {
    int accNum = std::stoi(accNumStr);
    double balance = std::stod(balanceStr);
    // Continue...
} catch (const std::exception& e) {
    std::cerr << "Parsing error: " << e.what() << " in line: " << line << "\n";
    continue;
}
```

### 21.2.7 Summary

- File I/O enables **persistence** by saving customers and accounts to files and loading them later.
- Use simple text-based serialization formats for readability and ease of debugging.
- Implement **serialization** (`save`) and **deserialization** (`load`) methods in your classes.
- Validate input thoroughly when reading files to avoid corrupt or malformed data.
- Handle file errors gracefully, notifying users of issues.
- This design can be extended to include transaction history and more complex data formats like JSON or XML with additional libraries.

By integrating file storage, your bank management system gains the ability to remember its data across runs, forming a foundation for a fully functional application.

## 21.3 Basic Exception Handling and User Input Validation

Building a reliable console-based bank management system requires robust handling of **invalid user inputs** and unexpected runtime errors. Users may enter wrong data types, invalid account numbers, or negative deposit amounts, and your program must detect these issues gracefully to avoid crashes or corrupted data. This section explains how to use C++'s exception handling mechanisms alongside input validation techniques to create a stable and user-friendly application.

### 21.3.1 Why Exception Handling and Input Validation Matter

- **Input Validation** ensures data entered by users meets expected criteria before processing. For example, deposit amounts should be positive, and names should not be empty.
- **Exception Handling** allows the program to detect, report, and recover from unexpected errors during execution without terminating abruptly.

Together, these techniques help maintain **program stability** and improve the user experience by providing meaningful feedback when something goes wrong.

### 21.3.2 C Exception Handling Basics

C++ provides a structured way to handle errors using three keywords:

- `try` — Block of code where exceptions might occur.
- `throw` — Statement to signal an error by raising an exception.
- `catch` — Block to handle the thrown exceptions.

Example:

Full runnable code:

```
#include <iostream>
#include <stdexcept>

int divide(int a, int b) {
    if (b == 0) {
        throw std::runtime_error("Division by zero");
    }
    return a / b;
}

int main() {
    try {
        int result = divide(10, 0);
        std::cout << "Result: " << result << "\n";
    } catch (const std::runtime_error& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
    return 0;
}
```

Output:

```
Error: Division by zero
```

This example demonstrates how exceptions propagate up the call stack until caught, allowing the program to respond without crashing.

### 21.3.3 Validating User Input

When interacting with users via console input (`std::cin`), several issues can arise:

- User enters letters when a number is expected.
- Numeric input is outside the valid range.
- Required fields are left empty.

We must validate inputs before processing and prompt users again if invalid.

### 21.3.4 Example: Reading a Valid Positive Double

Here is a helper function that safely reads a positive `double` value from the user:

```cpp
#include <iostream>
#include <limits>
#include <stdexcept>

double readPositiveDouble(const std::string& prompt) {
    double value;
    while (true) {
        std::cout << prompt;
        std::cin >> value;

        if (std::cin.fail()) {
            // Input type error (e.g., letters instead of number)
            std::cin.clear(); // Clear error flag
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Discard invalid inpu
            std::cerr << "Invalid input. Please enter a numeric value.\n";
            continue;
        }

        if (value <= 0) {
            std::cerr << "Value must be positive. Try again.\n";
            continue;
        }

        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Clean up rest of line
        return value;
    }
}
```

Usage:

```cpp
double amount = readPositiveDouble("Enter deposit amount: ");
```

This function keeps prompting until the user enters a valid positive number, ensuring safe data before proceeding.

### 21.3.5 Throwing Exceptions for Invalid Operations

Inside your bank classes, you can enforce rules by throwing exceptions when invalid operations occur.

For example, modify the `withdraw` method to throw an exception on invalid amounts:

```cpp
#include <stdexcept>

void withdraw(double amount) {
    if (amount <= 0) {
        throw std::invalid_argument("Withdrawal amount must be positive.");
    }
    if (amount > balance) {
        throw std::runtime_error("Insufficient funds.");
    }
    balance -= amount;
    std::cout << "Withdrew $" << amount << ". New balance: $" << balance << "\n";
}
```

### 21.3.6 Handling Exceptions in the User Interface

When calling these functions from your main program or menu, catch exceptions to provide feedback:

```
try {
    double amount = readPositiveDouble("Enter withdrawal amount: ");
    account->withdraw(amount);
} catch (const std::invalid_argument& e) {
    std::cerr << "Invalid input: " << e.what() << "\n";
} catch (const std::runtime_error& e) {
    std::cerr << "Transaction failed: " << e.what() << "\n";
}
```

This way, the program informs users why the operation failed and prompts for corrective action without crashing.

### 21.3.7 Validating Strings and Other Inputs

For string inputs such as customer names, addresses, or phone numbers, check that inputs are **not empty** and meet basic format rules.

Example:

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

std::string readNonEmptyString(const std::string& prompt) {
    std::string input;
    while (true) {
        std::cout << prompt;
        std::getline(std::cin, input);

        if (input.empty()) {
            std::cerr << "Input cannot be empty. Please try again.\n";
            continue;
        }
        return input;
    }
}
```

### 21.3.8 Example: Putting It All Together

Here's a snippet demonstrating reading valid customer info and depositing money, with validation and exception handling:

```cpp
int main() {
    try {
        std::string name = readNonEmptyString("Enter customer name: ");
        std::string address = readNonEmptyString("Enter customer address: ");
        std::string phone = readNonEmptyString("Enter customer phone number: ");

        Customer customer(name, address, phone);

        auto savings = std::make_shared<SavingsAccount>(1001, 0.0, 0.02);
        customer.addAccount(savings);

        double depositAmount = readPositiveDouble("Enter initial deposit amount: ");
        savings->deposit(depositAmount);

        // Attempt withdrawal with exception handling
        double withdrawalAmount = readPositiveDouble("Enter withdrawal amount: ");
        savings->withdraw(withdrawalAmount);

    } catch (const std::exception& e) {
        std::cerr << "An unexpected error occurred: " << e.what() << "\n";
    }

    return 0;
}
```

This approach:

- Validates every input.
- Uses exceptions to handle invalid operations.
- Keeps the program stable and user-friendly.

### 21.3.9 Summary

- Use **input validation loops** to ensure users enter valid data types and ranges.
- Throw **exceptions** inside classes to enforce business rules (e.g., no negative withdrawals).
- Catch exceptions at higher levels to provide informative feedback and prevent crashes.
- Validate strings to prevent empty or malformed entries.
- Clear input error states (`std::cin.clear()`) and discard invalid input to avoid infinite loops.
- Design your program so that **exceptions separate normal flow from error handling**, improving code clarity.

By applying these practices, your bank management system will robustly handle incorrect inputs and runtime errors, leading to a safer and more polished user experience.

# Chapter 22. Project: Mini Game (e.g., Tic-Tac-Toe or Snake)

# 22 Project: Mini Game (e.g., Tic-Tac-Toe or Snake)

## 22.1 Game Logic and State Management

Creating a mini game like Tic-Tac-Toe or Snake is a great way to learn core programming concepts such as state management, input handling, and logic implementation. At the heart of any game is the **game logic**—the rules and flow that determine how the game behaves—and the **game state**, which tracks all the current information needed to make decisions. This section explains how to design and manage these aspects effectively using C++ classes or structs, providing clear, maintainable, and extensible code.

### 22.1.1 Understanding Game State and Logic

**Game state** is the snapshot of all relevant data at any point during gameplay. For example:

- In **Tic-Tac-Toe**, it includes the board cells, which player's turn it is, and whether the game is over.
- In **Snake**, it might track the snake's body positions, food location, score, and current direction.

**Game logic** is the set of rules controlling:

- How players make moves.
- How the game updates the state.
- Conditions for winning, losing, or drawing.
- Turn management and validation.

### 22.1.2 Structuring Game State with Classes or Structs

Using classes or structs to represent the game state helps encapsulate data and related functions. This keeps your code organized, easier to debug, and extensible for future features.

**Example: Tic-Tac-Toe Game State**

```
#include <iostream>
#include <array>

class TicTacToe {
private:
    std::array<char, 9> board;  // 3x3 board stored in a 1D array
    char currentPlayer;          // 'X' or 'O'
    bool gameOver;

public:
    TicTacToe() : board{' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '}, currentPlayer('X'), gameOver(fals

    void displayBoard() const;
    bool makeMove(int position);
    void switchPlayer();
    bool checkWin() const;
    bool checkDraw() const;
    bool isGameOver() const { return gameOver; }
};
```

- The board uses a fixed-size array of 9 chars, initially empty (`' '`).
- `currentPlayer` tracks whose turn it is.
- `gameOver` flags when the game ends.

### 22.1.3 Move Validation

Before accepting a move, you must validate:

- Is the position within the valid range (0 to 8)?
- Is the position unoccupied?

The `makeMove` method returns `true` if the move is valid and applied; otherwise, `false`:

```
bool TicTacToe::makeMove(int position) {
    if (position < 0 || position >= 9) {
        std::cout << "Invalid position. Choose between 0 and 8.\n";
        return false;
    }
    if (board[position] != ' ') {
        std::cout << "Position already taken. Try again.\n";
        return false;
    }
    board[position] = currentPlayer;
    return true;
}
```

This guards against invalid or illegal moves.

### 22.1.4 Switching Turns

After a valid move, switch the player from `'X'` to `'O'` or vice versa:

```
void TicTacToe::switchPlayer() {
    currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';
}
```

This simple toggle ensures alternating turns.

### 22.1.5 Win and Draw Detection

Checking for a **win** requires inspecting all winning combinations (rows, columns, diagonals). If any line contains the same non-empty symbol, that player wins.

```
bool TicTacToe::checkWin() const {
    // All winning combinations: 8 total
    const int winCombos[8][3] = {
        {0, 1, 2}, {3, 4, 5}, {6, 7, 8},  // Rows
        {0, 3, 6}, {1, 4, 7}, {2, 5, 8},  // Columns
        {0, 4, 8}, {2, 4, 6}              // Diagonals
    };

    for (auto& combo : winCombos) {
        if (board[combo[0]] != ' ' &&
            board[combo[0]] == board[combo[1]] &&
            board[combo[1]] == board[combo[2]]) {
            return true;
        }
    }
    return false;
}
```

**Draw** occurs when all cells are filled without any winner:

```
bool TicTacToe::checkDraw() const {
    for (char cell : board) {
        if (cell == ' ') return false;
    }
    return true;
}
```

### 22.1.6 Displaying the Board

For user interaction, a method to print the current board state is helpful:

```
void TicTacToe::displayBoard() const {
    std::cout << "Current board:\n";
    for (int i = 0; i < 9; i++) {
        std::cout << (board[i] == ' ' ? std::to_string(i) : std::string(1, board[i]));
        if (i % 3 != 2) std::cout << " | ";
        else if (i != 8) std::cout << "\n---------\n";
    }
    std::cout << "\n\n";
}
```

Numbers in empty cells help players choose moves by position.

### 22.1.7 Example Main Loop Using the Class

Here is a simple `main` function demonstrating game flow using the `TicTacToe` class:

```
int main() {
    TicTacToe game;

    while (!game.isGameOver()) {
        game.displayBoard();

        int move;
        std::cout << "Player " << (game.currentPlayer) << ", enter your move (0-8): ";
        std::cin >> move;

        if (!game.makeMove(move)) {
            std::cout << "Invalid move, try again.\n";
            continue;
        }

        if (game.checkWin()) {
            game.displayBoard();
            std::cout << "Player " << game.currentPlayer << " wins!\n";
            break;
        }

        if (game.checkDraw()) {
            game.displayBoard();
            std::cout << "It's a draw!\n";
            break;
        }

        game.switchPlayer();
    }

    return 0;
}
```

### 22.1.8 Extending and Improving the Design

- **Separate Input Handling:** You can isolate user input logic from game logic for better modularity.
- **Use Enums:** Represent players and cell states with `enum` types for readability.
- **Support Undo or Save:** Add methods to save/load state.
- **Generalize Board Size:** Instead of fixed 3x3, support dynamic board sizes.
- **For Snake:** Manage the snake body as a list of coordinates, implement movement and collision detection.

### 22.1.9 Summary

- Represent the game state clearly using classes or structs to hold all necessary data.
- Implement move validation to ensure only legal moves modify the state.

- Maintain and update turn information to control player flow.
- Detect game over conditions: wins and draws, using well-defined checks.
- Keep code organized and extensible by encapsulating logic inside methods.
- Provide clear feedback to players by displaying the board and errors.

With this approach, you build a solid foundation for your mini game, ready for enhancements like AI opponents, scoring, or graphical interfaces.

## 22.2 User Interaction via Console or Basic GUI

Interacting with the player is a crucial part of any game. Whether through a console or a simple graphical interface, your game needs to receive input from the player, display the current state clearly, and provide feedback on actions. This section explores effective ways to manage user interaction in C++, focusing on console I/O and introducing basic concepts for simple GUI or enhanced text-based graphics.

### 22.2.1 Console Interaction: The Basics

For many beginner projects like Tic-Tac-Toe or Snake, the **console** is a practical and accessible way to interact with players.

- **Input:** Use `std::cin` to read player commands, such as move positions or direction choices.
- **Output:** Use `std::cout` to display the game board, messages, and prompts.

Console interaction is text-based but can be enhanced with ASCII art or clear formatting.

### 22.2.2 Handling User Input in Console

User input can come in many forms — numbers, letters, or commands. It's important to:

- Prompt clearly what input is expected.
- Validate input to avoid errors or invalid moves.
- Provide instructions or feedback on invalid input.

**Example: Prompting a move in Tic-Tac-Toe**

```cpp
int getPlayerMove() {
    int move;
    std::cout << "Enter your move (0-8): ";
    std::cin >> move;
    while (std::cin.fail() || move < 0 || move > 8) {
        std::cin.clear();  // Clear error state
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n'); // Discard bad input
        std::cout << "Invalid input. Please enter a number between 0 and 8: ";
        std::cin >> move;
    }
    return move;
}
```

This snippet ensures the user inputs a valid integer within the allowed range.

### 22.2.3 Rendering the Game State in Console

Displaying the current game state clearly helps players understand what's happening. For Tic-Tac-Toe, displaying the board as a 3x3 grid with numbered cells or player marks is common:

```
void displayBoard(const std::array<char, 9>& board) {
    for (int i = 0; i < 9; i++) {
        std::cout << (board[i] == ' ' ? std::to_string(i) : std::string(1, board[i]));
        if ((i + 1) % 3 != 0) std::cout << " | ";
        else if (i != 8) std::cout << "\n---------\n";
    }
    std::cout << "\n\n";
}
```

This shows player symbols or cell numbers, helping the user know which positions are available.

### 22.2.4 Enhancing Console Output with ASCII Art

You can spice up the console display by using **ASCII art** to represent game elements more visually. For example, the snake in a Snake game can be rendered as a series of characters on a grid:

```
##########
#        #
#   @    #
#        #
##########
```

Where `#` is a wall, space is empty, and `@` is the snake's head.

### 22.2.5 Clearing the Screen for Updated Displays

To create the illusion of animation or a refreshed display, clear the console before printing the updated state. This keeps the interface clean.

On Windows, you can use:

```
#include <cstdlib>
void clearScreen() {
    std::system("cls");
}
```

On Unix/Linux/macOS:

```
void clearScreen() {
    std::system("clear");
}
```

**Note:** Using `system()` is not the most efficient method but is sufficient for small projects.

Example:

```
clearScreen();
displayBoard(board);
```

This will erase previous output and display the fresh board, simulating a dynamic game screen.

### 22.2.6 Simple GUI with Lightweight Libraries

If you need to go beyond console and try a basic graphical interface, libraries like **SFML** (Simple and Fast Multimedia Library) or **SDL** (Simple DirectMedia Layer) offer accessible options. They provide windows, graphics, and event handling with relatively simple APIs.

**Example (conceptual):** In SFML, you can draw shapes or text to represent the game state, and handle keyboard or mouse input.

```
// Pseudocode, requires SFML setup
sf::RenderWindow window(sf::VideoMode(300, 300), "Tic-Tac-Toe");

while (window.isOpen()) {
    sf::Event event;
    while (window.pollEvent(event)) {
        if (event.type == sf::Event::Closed) window.close();

        // Handle mouse clicks to register moves
    }

    window.clear(sf::Color::White);
    // Draw board grid and X/O marks
    window.display();
}
```

Using such libraries requires setup and linking, so start with console interaction before moving to GUI.

### 22.2.7 Updating the Display and Providing Feedback

User interaction is a two-way street:

- After a player makes a move, update the display immediately so they see the result.
- Provide clear messages on what just happened — e.g., "Invalid move," "Player X wins," or "Game over, it's a draw."

Example snippet:

```
if (!game.makeMove(move)) {
    std::cout << "Invalid move. Try again.\n";
} else {
    clearScreen();
    game.displayBoard();
    if (game.checkWin()) {
        std::cout << "Player " << game.currentPlayer << " wins!\n";
    } else if (game.checkDraw()) {
        std::cout << "It's a draw!\n";
    } else {
        game.switchPlayer();
        std::cout << "Next turn: Player " << game.currentPlayer << "\n";
    }
}
```

### 22.2.8 Summary

- Use **console input/output** to communicate with the player, prompting for input and displaying game state.
- Validate all input to prevent errors and improve usability.
- Enhance output clarity with formatted displays or ASCII art.
- Clear the console between frames to simulate dynamic updates.
- Consider simple GUI libraries like SFML for graphical interfaces, but start with console for simplicity.
- Always provide clear feedback after every player action.

By carefully designing user interaction, you ensure players remain engaged and understand the game's progress, making your mini game both fun and accessible.

## 22.3 Using STL Containers and Algorithms

The C++ Standard Template Library (STL) provides powerful, flexible containers and algorithms that can greatly simplify game development. Using STL containers such as `vector`, `array`, or `deque` helps you efficiently manage game data like board cells, player moves, or game objects. Coupled with STL algorithms, they enable concise, readable, and performant code for common tasks like searching, sorting, or evaluating game state.

This section shows how to integrate STL features into your mini game projects, boosting maintainability and clarity.

### 22.3.1 Choosing the Right STL Container for Game Data

Selecting an appropriate container depends on your game's data needs:

- **`std::array`**: Fixed-size arrays known at compile time. Ideal for small, static boards like Tic-Tac-Toe's 3x3 grid.
- **`std::vector`**: Dynamic arrays with flexible size, suitable for games where board size or number of elements can vary, like Snake's growing body.
- **`std::deque`**: Double-ended queue supporting efficient insertions and deletions at both ends. Perfect for data structures like Snake's body where you frequently add at the front (head) and remove from the back (tail).

### 22.3.2 Example 1: Using `std::array` for Tic-Tac-Toe Board

For a fixed 3x3 board, `std::array<char, 9>` cleanly represents the cells:

```cpp
#include <array>

class TicTacToe {
private:
    std::array<char, 9> board{};  // default initialized to '\0' or ' ' if set explicitly

public:
    TicTacToe() {
        board.fill(' ');
    }

    void display() const {
        for (int i = 0; i < 9; ++i) {
            std::cout << (board[i] == ' ' ? std::to_string(i) : std::string(1, board[i]));
            if ((i + 1) % 3 != 0) std::cout << " | ";
            else std::cout << "\n---------\n";
        }
    }
    // Other methods like makeMove, checkWin, etc.
};
```

Using `std::array` improves safety over raw arrays because it provides bounds checking with `.at()` method and integrates with STL algorithms.

### 22.3.3 Example 2: Using `std::vector` for Snakes Body

Snake's body dynamically grows and shrinks, so `std::vector` is a natural fit:

```
#include <vector>
#include <utility>  // for std::pair

class Snake {
private:
    std::vector<std::pair<int, int>> body;  // list of (x, y) positions

public:
    Snake() {
        body.push_back({5, 5});  // initial position
    }

    void move(int dx, int dy) {
        // Add new head position
        std::pair<int, int> newHead = {body.front().first + dx, body.front().second + dy};
        body.insert(body.begin(), newHead);

        // Remove tail segment to simulate movement
        body.pop_back();
    }

    void grow() {
        // When growing, do not remove tail segment after moving
    }

    const std::vector<std::pair<int, int>>& getBody() const {
        return body;
    }
};
```

Here, `std::vector` provides efficient access and resizing.

### 22.3.4 Using STL Algorithms for Game Logic

STL algorithms simplify common operations, making your code shorter and clearer.

#### Searching: Check if a Position is Occupied

To check if the snake occupies a position:

```
#include <algorithm>

bool isOccupied(const std::vector<std::pair<int, int>>& body, std::pair<int, int> pos) {
    return std::find(body.begin(), body.end(), pos) != body.end();
}
```

This returns `true` if the snake's body contains the coordinate `pos`.

### Counting and Evaluating States

Suppose you want to count how many empty cells remain on a Tic-Tac-Toe board:

```
int countEmptyCells(const std::array<char, 9>& board) {
    return std::count(board.begin(), board.end(), ' ');
}
```

This helps to quickly evaluate if the board is full (a draw condition).

### Sorting: Managing Scores or Leaderboards

If your game includes scoring, use `std::sort` to rank players:

```
#include <vector>
#include <algorithm>
#include <iostream>

struct Player {
    std::string name;
    int score;
};

void sortPlayersByScore(std::vector<Player>& players) {
    std::sort(players.begin(), players.end(), [](const Player& a, const Player& b) {
        return a.score > b.score;  // descending order
    });
}
```

This lambda-based comparator sorts players from highest to lowest score.

## 22.3.5 Example: Integrating STL into Game Loop

In Tic-Tac-Toe, checking if a player has won can be made cleaner with `std::all_of`:

```
bool checkWin(const std::array<char, 9>& board, char player) {
    const int winPatterns[8][3] = {
        {0,1,2}, {3,4,5}, {6,7,8},  // rows
        {0,3,6}, {1,4,7}, {2,5,8},  // columns
        {0,4,8}, {2,4,6}            // diagonals
    };

    for (auto& pattern : winPatterns) {
        if (std::all_of(std::begin(pattern), std::end(pattern),
            [&](int index) { return board[index] == player; })) {
            return true;
        }
    }
    return false;
}
```

`std::all_of` improves readability by expressing "all positions in this pattern belong to the player."

### 22.3.6 Benefits of Using STL Containers and Algorithms

- **Safety:** Containers handle memory management and bounds checking.
- **Expressiveness:** Algorithms like `find`, `count`, `sort`, and `all_of` make intentions clear.
- **Efficiency:** STL implementations are optimized for performance.
- **Maintainability:** Clear, concise code is easier to update and debug.
- **Reusability:** STL components work seamlessly together across projects.

### 22.3.7 Summary

- Use `std::array` for fixed-size game boards, like Tic-Tac-Toe.
- Use `std::vector` or `std::deque` for dynamic or double-ended data, such as Snake's growing body.
- Apply STL algorithms (`find`, `count`, `all_of`, `sort`) to simplify searching, counting, sorting, and state evaluation.
- Integrate STL features into your game loop for clearer, more maintainable code.

Harnessing STL's power helps you build robust, efficient, and readable game code while focusing on game logic rather than low-level data management.

# Chapter 23.

# Project: Data Processing and Analysis Tool

# 23 Project: Data Processing and Analysis Tool

## 23.1 Reading and Processing Large Data Files

When building a data processing and analysis tool in C++, efficiently reading and parsing large data files is fundamental. Large files can be gigabytes in size and may not fit entirely in memory, so your program must handle input streams carefully to avoid crashes or performance bottlenecks.

This section covers key techniques to read large files using C++ file streams, parse common file formats like CSV and TSV, detect errors, and manage memory wisely. Practical code examples demonstrate chunked reading, parsing lines into data fields, and simple preprocessing steps.

### 23.1.1 Using File Streams for Large Files

The primary way to read files in C++ is through the standard input file stream `std::ifstream`. It allows reading data sequentially, which is important for large files since you typically cannot or should not load the entire file into memory at once.

Example: Open and read a file line by line

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

void readFileLineByLine(const std::string& filename) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Failed to open file: " << filename << "\n";
        return;
    }

    std::string line;
    while (std::getline(file, line)) {
        // Process each line here
        std::cout << line << "\n";
    }

    file.close();
}
```

This reads the file one line at a time, making it memory efficient regardless of file size.

### 23.1.2 Handling Different File Formats: CSV and TSV

**CSV (Comma-Separated Values)** and **TSV (Tab-Separated Values)** are common plain-text formats where each line represents a data record, and fields are separated by commas or tabs.

To parse these formats, you need to:

- Split each line into fields based on the delimiter (`,` for CSV, `\t` for TSV).
- Handle cases where fields might be quoted or contain delimiter characters.
- Convert fields into appropriate data types.

### 23.1.3 Simple Parsing by Delimiter

For many simple cases, you can split lines on delimiter characters using `std::getline` with a custom delimiter.

Example: Parse a CSV line into fields:

```cpp
#include <sstream>
#include <vector>

std::vector<std::string> parseCSVLine(const std::string& line) {
    std::vector<std::string> fields;
    std::stringstream ss(line);
    std::string field;

    while (std::getline(ss, field, ',')) {
        fields.push_back(field);
    }

    return fields;
}
```

This approach works for basic CSV files without quoted fields.

### 23.1.4 Chunked Reading for Large Files

Sometimes reading line by line is sufficient, but for extremely large files, you might want to process the file in **chunks** — reading a block of data, then processing records inside that block before moving on.

This is useful when your data records might span multiple lines, or when using lower-level optimizations.

Example: Reading fixed-size chunks (buffered reading)

```
#include <fstream>
#include <vector>

void readFileInChunks(const std::string& filename, std::size_t chunkSize = 4096) {
    std::ifstream file(filename, std::ios::binary);
    if (!file) {
        std::cerr << "Cannot open file\n";
        return;
    }

    std::vector<char> buffer(chunkSize);
    while (file.read(buffer.data(), buffer.size()) || file.gcount() > 0) {
        std::streamsize bytesRead = file.gcount();
        // Process buffer[0..bytesRead-1] here
        std::cout.write(buffer.data(), bytesRead);
    }
}
```

This reads the file in 4 KB blocks and processes the raw bytes. For text files, you must handle cases where chunks split lines, which adds complexity.

### 23.1.5 Error Handling and Validation

When reading files, always check:

- Whether the file opened successfully.
- Whether reads fail unexpectedly.
- Whether lines or fields conform to expected format.

Example:

```
if (!file.is_open()) {
    throw std::runtime_error("Cannot open input file");
}
if (file.bad()) {
    std::cerr << "I/O error while reading file\n";
}
```

Validate data fields to catch corrupted or malformed data early.

### 23.1.6 Memory Considerations

- Avoid loading entire large files into memory.
- Use **streaming** approaches: process data as it is read.
- If you must store parsed data, consider container efficiency (`std::vector` vs `std::list`), and reserve capacity ahead of time when possible.
- For truly large datasets, consider using databases or memory-mapped files (outside scope here).

### 23.1.7 Example: Reading and Preprocessing CSV File

This example reads a CSV file line by line, parses fields, and converts the first two fields to numeric values for analysis.

Full runnable code:

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

struct DataRecord {
    int id;
    double value;
    std::string label;
};

std::vector<DataRecord> readCSVData(const std::string& filename) {
    std::vector<DataRecord> records;
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "Error opening file\n";
        return records;
    }

    std::string line;
    // Optionally skip header line
    std::getline(file, line);

    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string field;
        std::vector<std::string> fields;

        while (std::getline(ss, field, ',')) {
            fields.push_back(field);
        }

        if (fields.size() < 3) continue;  // Skip invalid lines

        try {
            int id = std::stoi(fields[0]);
            double value = std::stod(fields[1]);
            std::string label = fields[2];

            records.push_back({id, value, label});
        } catch (const std::exception& e) {
            std::cerr << "Parsing error: " << e.what() << "\n";
            // Skip malformed line
        }
    }
    return records;
}

int main() {
    auto data = readCSVData("data.csv");
    std::cout << "Loaded " << data.size() << " records.\n";

    // Example preprocessing: filter values > threshold
    double threshold = 50.0;
    for (const auto& record : data) {
        if (record.value > threshold) {
            std::cout << "ID: " << record.id << ", Value: " << record.value
                      << ", Label: " << record.label << "\n";
        }
    }
    return 0;
}
```

This example shows:

- Reading a CSV file line by line.
- Splitting lines into fields.
- Converting string fields to numeric types with error checking.
- Simple data filtering as a preprocessing step.

### 23.1.8 Summary

- Use `std::ifstream` and line-by-line reading for memory-efficient input.
- Parse CSV, TSV, or other delimited formats by splitting lines on delimiters.
- Handle malformed data and file errors gracefully.
- For very large files, consider chunked reading but handle incomplete records carefully.
- Keep memory usage minimal by processing data as streams, avoiding loading entire files at once.
- Use exceptions or error flags to detect and report problems early.

Efficient file reading and parsing lay the foundation for powerful data processing tools and enable handling real-world large datasets with confidence.

## 23.2 Using STL Algorithms and Data Structures

In building a data processing and analysis tool, efficiently organizing and manipulating your data is key. The C++ Standard Template Library (STL) offers versatile containers such as `vector`, `map`, and `unordered_map`, along with powerful algorithms like `sort`, `accumulate`, and `find_if`. These allow you to perform common data processing tasks—grouping, filtering, searching, and summarizing—with concise, readable, and high-performance code.

This section demonstrates how to harness STL containers and algorithms to transform raw data into meaningful insights.

### 23.2.1 Choosing STL Containers for Data Storage

Before analyzing data, you must decide how to store it.

- **`std::vector`**: Dynamic array for ordered sequences. Ideal for datasets you want to process sequentially or sort.
- **`std::map`**: Sorted associative container mapping keys to values. Useful for grouping data by keys in sorted order.
- **`std::unordered_map`**: Hash table-based mapping with faster average lookup. Great when order is not required but quick access is.

### 23.2.2 Example Dataset Structure

Suppose you have a dataset of `DataRecords` (from the previous chapter):

```cpp
struct DataRecord {
    int id;
    double value;
    std::string category;
};
```

We will perform grouping, filtering, and summarizing based on `category` and `value`.

### 23.2.3 Grouping Data with `std::map`

To group records by their category and compute aggregates like sums or counts, `std::map` is very useful.

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

void groupByCategory(const std::vector<DataRecord>& records) {
    std::map<std::string, double> sumByCategory;

    for (const auto& record : records) {
        sumByCategory[record.category] += record.value;  // accumulate sums per category
    }

    // Display results
    for (const auto& [category, total] : sumByCategory) {
        std::cout << "Category: " << category << ", Total Value: " << total << "\n";
    }
}
```

Here, each category string maps to the sum of values belonging to that category.

### 23.2.4 Fast Lookup with `std::unordered_map`

If order of categories does not matter but you want faster access, use `std::unordered_map`:

```cpp
#include <unordered_map>

void fastLookupCategory(const std::vector<DataRecord>& records) {
    std::unordered_map<std::string, int> countByCategory;

    for (const auto& record : records) {
        countByCategory[record.category]++;
    }

    for (const auto& [category, count] : countByCategory) {
        std::cout << "Category: " << category << ", Count: " << count << "\n";
    }
}
```

This counts how many records belong to each category.

### 23.2.5 Sorting Data with `std::sort`

Suppose you want to sort records by their value in descending order to identify top performers.

```cpp
#include <algorithm>

void sortByValue(std::vector<DataRecord>& records) {
    std::sort(records.begin(), records.end(),
              [](const DataRecord& a, const DataRecord& b) {
                  return a.value > b.value;  // descending order
              });
}
```

You can then process or display the sorted vector.

### 23.2.6 Summarizing Data with `std::accumulate`

To compute a total or average of a numeric field, `std::accumulate` is very handy.

```cpp
#include <numeric>

double totalValue(const std::vector<DataRecord>& records) {
    return std::accumulate(records.begin(), records.end(), 0.0,
                           [](double sum, const DataRecord& rec) {
                               return sum + rec.value;
                           });
}
```

To calculate an average, divide by the number of records after checking it's not zero.

### 23.2.7 Filtering Data with `std::copy_if` and `std::remove_if`

You can filter records matching specific criteria with `std::copy_if`:

```
#include <algorithm>

std::vector<DataRecord> filterByThreshold(const std::vector<DataRecord>& records, double threshold) {
    std::vector<DataRecord> filtered;
    std::copy_if(records.begin(), records.end(), std::back_inserter(filtered),
                 [threshold](const DataRecord& rec) {
                     return rec.value > threshold;
                 });
    return filtered;
}
```

Alternatively, to remove elements from the original container (in-place), use `std::remove_if` followed by `erase`:

```
void removeLowValues(std::vector<DataRecord>& records, double threshold) {
    auto it = std::remove_if(records.begin(), records.end(),
                             [threshold](const DataRecord& rec) {
                                 return rec.value <= threshold;
                             });
    records.erase(it, records.end());
}
```

### 23.2.8 Searching with `std::find_if`

To find the first record matching a condition:

```
auto it = std::find_if(records.begin(), records.end(),
                       [](const DataRecord& rec) {
                           return rec.category == "Important" && rec.value > 100;
                       });

if (it != records.end()) {
    std::cout << "Found record with id " << it->id << " meeting criteria.\n";
} else {
    std::cout << "No matching record found.\n";
}
```

### 23.2.9 Combining STL Algorithms for Complex Queries

STL algorithms can be composed to perform multi-step analyses clearly.

Example: Find the category with the highest total value.

```
std::map<std::string, double> sumByCategory;
for (const auto& rec : records) {
    sumByCategory[rec.category] += rec.value;
}
```

```
auto maxPair = std::max_element(sumByCategory.begin(), sumByCategory.end(),
                                [](const auto& a, const auto& b) {
                                    return a.second < b.second;
                                });

if (maxPair != sumByCategory.end()) {
    std::cout << "Category with highest total value: " << maxPair->first
              << " = " << maxPair->second << "\n";
}
```

### 23.2.10 Practical Tips

- When possible, reserve space in vectors upfront to minimize reallocations:
  ```
  std::vector<DataRecord> data;
  data.reserve(10000); // for large datasets
  ```
- Choose `map` if you need sorted keys, `unordered_map` for speed.
- Use lambdas for concise custom predicates in algorithms.
- Combine STL algorithms with range-based `for` loops for elegant code.

### 23.2.11 Summary

- Use **`vector`** for flexible ordered collections of data records.
- Use **`map`** and **`unordered_map`** for grouping and fast lookups by keys.
- Apply **`sort`** to order data by criteria.
- Use **`accumulate`** for summing or aggregating numeric data.
- Use **`copy_if`, `remove_if`, and `find_if`** for filtering and searching datasets.
- Combining STL containers and algorithms enables concise, efficient data analysis pipelines.

Mastering STL's data structures and algorithms will empower you to build powerful and maintainable data processing tools with minimal code complexity.

## 23.3 Basic Multithreading for Performance

As datasets grow larger, processing them sequentially can become slow and inefficient. C++ supports **multithreading**, allowing your program to run multiple tasks concurrently on different CPU cores. This can significantly improve the performance of data processing tools by parallelizing compute-intensive tasks like parsing, filtering, and aggregation.

This section introduces simple multithreading concepts and techniques to parallelize your data processing tasks, including thread creation, dividing workloads, and synchronizing shared results safely. Practical examples demonstrate how to split data processing across threads and combine outcomes correctly.

### 23.3.1 Why Use Multithreading?

Modern computers typically have multiple CPU cores. Multithreading leverages this hardware by running independent tasks simultaneously, reducing overall execution time.

For example, if you have a million data records to analyze, you can divide them into chunks and process each chunk on a separate thread, then combine the partial results. This can make your tool much faster compared to a single-threaded approach.

### 23.3.2 Creating Threads in C

The C++ Standard Library offers `std::thread` to create and manage threads easily.

Basic example creating and running a thread:

Full runnable code:

```
#include <iostream>
#include <thread>

void printMessage() {
    std::cout << "Hello from thread!\n";
}

int main() {
    std::thread t(printMessage);  // Start thread running printMessage()
    t.join();  // Wait for thread to finish before exiting main
    return 0;
}
```

The `join()` call ensures the main thread waits for the new thread to finish, preventing premature exit.

### 23.3.3 Partitioning Workload

To use multithreading for data processing, split your dataset into parts, then assign each part to a thread.

Example: Suppose you have a vector of records and want to sum their values in parallel.

```
#include <vector>
#include <thread>
#include <numeric>
#include <iostream>

struct DataRecord {
    double value;
    // Other fields...
};

void partialSum(const std::vector<DataRecord>& data, size_t start, size_t end, double& result) {
    result = 0.0;
    for (size_t i = start; i < end; ++i) {
        result += data[i].value;
    }
}

int main() {
    std::vector<DataRecord> data(1000000, {1.0}); // Sample data: 1 million records with value 1.0

    size_t mid = data.size() / 2;
    double sum1 = 0.0, sum2 = 0.0;

    std::thread t1(partialSum, std::ref(data), 0, mid, std::ref(sum1));
    std::thread t2(partialSum, std::ref(data), mid, data.size(), std::ref(sum2));

    t1.join();
    t2.join();

    double totalSum = sum1 + sum2;
    std::cout << "Total sum: " << totalSum << "\n";
}
```

**Explanation:**

- The vector is split into two halves.
- Two threads compute the partial sums concurrently.
- Using `std::ref` to pass references so results are stored correctly.
- The main thread waits for both threads with `join()`, then combines results.

### 23.3.4 Synchronization and Data Safety

When threads share data, **race conditions** may occur if multiple threads read/write simultaneously.

In the example above, each thread writes its own separate `sum` variable, so no synchronization is needed.

However, if threads update shared data structures, use synchronization primitives like `std::mutex` to avoid data corruption.

Example:

```cpp
#include <mutex>

std::mutex mtx;
double globalSum = 0.0;

void threadSafePartialSum(const std::vector<DataRecord>& data, size_t start, size_t end) {
    double localSum = 0.0;
    for (size_t i = start; i < end; ++i) {
        localSum += data[i].value;
    }
    std::lock_guard<std::mutex> lock(mtx); // Lock mutex during update
    globalSum += localSum;
}
```

Using `std::lock_guard` ensures the mutex is unlocked automatically when the scope ends, avoiding deadlocks.

### 23.3.5 Using a Thread Pool for More Threads

Manually managing many threads can get complicated. For more advanced use cases, thread pools manage a fixed number of threads and distribute tasks efficiently.

Although the C++ standard library does not include thread pools (until C++23's `std::jthread`), you can implement simple pools or use third-party libraries.

For beginners, splitting the work into a few threads, as shown above, is usually sufficient.

### 23.3.6 Handling Exceptions in Threads

If a thread throws an exception, it must be handled inside that thread. Otherwise, `std::terminate()` is called.

Wrap thread functions in try-catch blocks for safety:

```cpp
void safePartialSum(...) {
    try {
        // Processing code...
    } catch (const std::exception& e) {
        std::cerr << "Thread error: " << e.what() << "\n";
    }
}
```

### 23.3.7 Example: Parallel Filtering

Suppose you want to filter records with values above a threshold using two threads:

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

void filterRange(const std::vector<DataRecord>& input, size_t start, size_t end,
                 double threshold, std::vector<DataRecord>& output) {
    for (size_t i = start; i < end; ++i) {
        if (input[i].value > threshold) {
            output.push_back(input[i]);  // Note: vector push_back not thread-safe
        }
    }
}

int main() {
    std::vector<DataRecord> data(100000, {1.0});
    double threshold = 0.5;

    std::vector<DataRecord> filtered1, filtered2;

    std::thread t1(filterRange, std::ref(data), 0, data.size() / 2, threshold, std::ref(filtered1));
    std::thread t2(filterRange, std::ref(data), data.size() / 2, data.size(), threshold, std::ref(filte

    t1.join();
    t2.join();

    // Combine results
    filtered1.insert(filtered1.end(), filtered2.begin(), filtered2.end());

    std::cout << "Filtered count: " << filtered1.size() << "\n";
}
```

**Note:** Each thread writes to its own vector to avoid race conditions, then results are merged after threads finish.

### 23.3.8 Best Practices

- **Divide work evenly** to balance thread workloads.
- **Avoid sharing writable data** without synchronization.
- Use **thread-safe containers or local buffers** to prevent conflicts.
- **Join threads** before accessing their results.
- Start with a small number of threads (e.g., equal to hardware concurrency).
- Profile your program: not all tasks benefit from parallelism due to overhead.

### 23.3.9 Summary

- Multithreading enables running data processing tasks concurrently to improve performance.
- Use `std::thread` to create threads and assign parts of data to each.
- Synchronize shared data access with `std::mutex` to prevent race conditions.
- Design thread functions to work on local or partitioned data.
- Join threads before combining results.
- Handle exceptions within threads carefully.
- Parallelize operations like summing, filtering, and aggregation by splitting input data.

By carefully applying these multithreading concepts, your data processing tool can scale to handle large datasets more quickly and efficiently, fully utilizing modern multi-core processors.