# C++ Notes Index

## [Chapter 1: Introduction to C++](Chapter_1_Introduction_to_CPP.md)

* [1.1 What is C++?](Chapter_1_Introduction_to_CPP.md#11-what-is-c)
* [1.2 History & Evolution of C++](Chapter_1_Introduction_to_CPP.md#12-history--evolution-of-c)
* [1.3 Why Use C++? (Use Cases & Industry Adoption)](Chapter_1_Introduction_to_CPP.md#13-why-use-c-use-cases--industry-adoption)
* [1.4 C++ vs C vs Java vs Python](Chapter_1_Introduction_to_CPP.md#14-c-vs-c-vs-java-vs-python)
* [1.5 C++ Compilation Model](Chapter_1_Introduction_to_CPP.md#15-c-compilation-model)
* [1.6 Writing and Running a Simple C++ Program](Chapter_1_Introduction_to_CPP.md#16-writing-and-running-a-simple-c-program)
* [1.7 Input and Output (cin, cout)](Chapter_1_Introduction_to_CPP.md#17-input-and-output-cin-cout)
* [1.8 Basic Syntax & Program Structure](Chapter_1_Introduction_to_CPP.md#18-basic-syntax--program-structure)

## [Chapter 2: Variables, Data Types & Operators](Chapter_2_Variables_DataTypes_Operators.md)

* [2.1 Variable Declaration & Initialization](Chapter_2_Variables_DataTypes_Operators.md#21-variable-declaration--initialization)
* [2.2 Data Types: Primitive & Derived](Chapter_2_Variables_DataTypes_Operators.md#22-data-types-primitive--derived)
* [2.3 Type Conversion & Typecasting](Chapter_2_Variables_DataTypes_Operators.md#23-type-conversion--typecasting)
* [2.4 Operators in C++](Chapter_2_Variables_DataTypes_Operators.md#24-operators-in-c)
  + [2.4.1 Arithmetic](Chapter_2_Variables_DataTypes_Operators.md#241-arithmetic-operators)
  + [2.4.2 Relational](Chapter_2_Variables_DataTypes_Operators.md#242-relational-operators)
  + [2.4.3 Logical](Chapter_2_Variables_DataTypes_Operators.md#243-logical-operators)
  + [2.4.4 Bitwise](Chapter_2_Variables_DataTypes_Operators.md#244-bitwise-operators)
  + [2.4.5 Assignment](Chapter_2_Variables_DataTypes_Operators.md#245-assignment-operators)
  + [2.4.6 Ternary](Chapter_2_Variables_DataTypes_Operators.md#246-ternary-operator)
  + [2.4.7 Operator Precedence & Associativity](Chapter_2_Variables_DataTypes_Operators.md#247-operator-precedence--associativity)

## [Chapter 3: Control Flow & Functions](Chapter_3_Control_Flow_and_Functions.md)

* [3.1 Conditional Statements (if, else, switch)](Chapter_3_Control_Flow_and_Functions.md#31-conditional-statements-if-else-switch)
* [3.2 Loops (for, while, do-while)](Chapter_3_Control_Flow_and_Functions.md#32-loops-for-while-do-while)
* [3.3 Break, Continue, Goto](Chapter_3_Control_Flow_and_Functions.md#33-break-continue-goto)
* [3.4 Functions: Declaration, Definition, Calling](Chapter_3_Control_Flow_and_Functions.md#34-functions-declaration-definition-calling)
* [3.5 Function Parameters: Call by Value & Reference](Chapter_3_Control_Flow_and_Functions.md#35-function-parameters-call-by-value--reference)
* [3.6 Recursion in C++](Chapter_3_Control_Flow_and_Functions.md#36-recursion-in-c)
* [3.7 Inline Functions](Chapter_3_Control_Flow_and_Functions.md#37-inline-functions)
* [3.8 Default and Constant Arguments](Chapter_3_Control_Flow_and_Functions.md#38-default-and-constant-arguments)
* [3.9 Function Overloading](Chapter_3_Control_Flow_and_Functions.md#39-function-overloading)

## [Chapter 4: Arrays and Strings](Chapter_4_Arrays_and_Strings.md)

* [4.1 Arrays: 1D, 2D, and Multi-dimensional](Chapter_4_Arrays_and_Strings.md#41-arrays-1d-2d-and-multi-dimensional)
* [4.2 Operations on Arrays (Insert, Delete, Search, Traverse)](Chapter_4_Arrays_and_Strings.md#42-operations-on-arrays-insert-delete-search-traverse)
* [4.3 Strings in C++](Chapter_4_Arrays_and_Strings.md#43-strings-in-c)
  + [4.3.1 Character Arrays vs std::string](Chapter_4_Arrays_and_Strings.md#431-character-arrays-vs-stdstring)
  + [4.3.2 String Functions](Chapter_4_Arrays_and_Strings.md#432-string-functions)
  + [4.3.3 Input/Output of Strings](Chapter_4_Arrays_and_Strings.md#433-inputoutput-of-strings)
* [4.4 String Manipulation Problems](Chapter_4_Arrays_and_Strings.md#44-string-manipulation-problems)

## [Chapter 5: Pointers and Memory Management](Chapter_5_Pointers_and_Memory_Management.md)

* [5.1 Introduction to Pointers](Chapter_5_Pointers_and_Memory_Management.md#51-introduction-to-pointers)
* [5.2 Pointer Arithmetic](Chapter_5_Pointers_and_Memory_Management.md#52-pointer-arithmetic)
* [5.3 Pointers and Arrays](Chapter_5_Pointers_and_Memory_Management.md#53-pointers-and-arrays)
* [5.4 Pointers to Functions](Chapter_5_Pointers_and_Memory_Management.md#54-pointers-to-functions)
* [5.5 Dynamic Memory Allocation (new, delete)](Chapter_5_Pointers_and_Memory_Management.md#55-dynamic-memory-allocation-new-delete)
* [5.6 Dangling Pointers and Memory Leaks](Chapter_5_Pointers_and_Memory_Management.md#56-dangling-pointers-and-memory-leaks)
* [5.7 Smart Pointers (unique\_ptr, shared\_ptr, weak\_ptr)](Chapter_5_Pointers_and_Memory_Management.md#57-smart-pointers-unique_ptr-shared_ptr-weak_ptr)
* [5.8 RAII and Scope-Based Resource Management](Chapter_5_Pointers_and_Memory_Management.md#58-raii-and-scope-based-resource-management)

## [Chapter 6: Object-Oriented Programming](Chapter_6_Object-Oriented_Programming.md)

* [6.1 Introduction to OOP](Chapter_6_Object-Oriented_Programming.md#61-introduction-to-oop)
* [6.2 Classes and Objects](Chapter_6_Object-Oriented_Programming.md#62-classes-and-objects)
* [6.3 Constructors and Destructors](Chapter_6_Object-Oriented_Programming.md#63-constructors-and-destructors)
* [6.4 Access Specifiers (Public, Private, Protected)](Chapter_6_Object-Oriented_Programming.md#64-access-specifiers-public-private-protected)
* [6.5 Inheritance](Chapter_6_Object-Oriented_Programming.md#65-inheritance)
  + [6.5.1 Single](Chapter_6_Object-Oriented_Programming.md#651-single)
  + [6.5.2 Multiple](Chapter_6_Object-Oriented_Programming.md#652-multiple)
  + [6.5.3 Multilevel](Chapter_6_Object-Oriented_Programming.md#653-multilevel)
  + [6.5.4 Hierarchical](Chapter_6_Object-Oriented_Programming.md#654-hierarchical)
  + [6.5.5 Hybrid](Chapter_6_Object-Oriented_Programming.md#655-hybrid)
* [6.6 Polymorphism](Chapter_6_Object-Oriented_Programming.md#66-polymorphism)
  + [6.6.1 Compile-time (Function/Operator Overloading)](Chapter_6_Object-Oriented_Programming.md#661-compile-time-functionoperator-overloading)
  + [6.6.2 Run-time (Virtual Functions, vtables)](Chapter_6_Object-Oriented_Programming.md#662-run-time-virtual-functions-vtables)
* [6.7 Encapsulation & Abstraction](Chapter_6_Object-Oriented_Programming.md#67-encapsulation--abstraction)
* [6.8 this Pointer](Chapter_6_Object-Oriented_Programming.md#68-this-pointer)
* [6.9 Friend Functions and Classes](Chapter_6_Object-Oriented_Programming.md#69-friend-functions-and-classes)
* [6.10 Static Members](Chapter_6_Object-Oriented_Programming.md#610-static-members)
* [6.11 Object Slicing](Chapter_6_Object-Oriented_Programming.md#611-object-slicing)

## [Chapter 7: Advanced C++ Features](Chapter_7_Advanced_CPP_Features.md)

* [7.1 Templates (Function and Class)](Chapter_7_Advanced_CPP_Features.md#71-templates-function-and-class)
* [7.2 Exception Handling (try, catch, throw)](Chapter_7_Advanced_CPP_Features.md#72-exception-handling-try-catch-throw)
* [7.3 Namespaces](Chapter_7_Advanced_CPP_Features.md#73-namespaces)
* [7.4 Type Inference with auto](Chapter_7_Advanced_CPP_Features.md#74-type-inference-with-auto)
* [7.5 const, mutable, volatile](Chapter_7_Advanced_CPP_Features.md#75-const-mutable-volatile)
* [7.6 Lambdas and Closures](Chapter_7_Advanced_CPP_Features.md#76-lambdas-and-closures)
* [7.7 Move Semantics & Rvalue References](Chapter_7_Advanced_CPP_Features.md#77-move-semantics--rvalue-references)
* [7.8 Rule of 3 / 5 / 0](Chapter_7_Advanced_CPP_Features.md#78-rule-of-3--5--0)
* [7.9 Copy Constructor vs Move Constructor](Chapter_7_Advanced_CPP_Features.md#79-copy-constructor-vs-move-constructor)

## [Chapter 8: Standard Template Library (STL)](Chapter_8_Standard_Template_Library.md)

* [8.1 STL Overview](Chapter_8_Standard_Template_Library.md#81-stl-overview)
* [8.2 Containers](Chapter_8_Standard_Template_Library.md#82-containers)
  + [8.2.1 Vector](Chapter_8_Standard_Template_Library.md#821-vector)
  + [8.2.2 List](Chapter_8_Standard_Template_Library.md#822-list)
  + [8.2.3 Deque](Chapter_8_Standard_Template_Library.md#823-deque)
  + [8.2.4 Stack](Chapter_8_Standard_Template_Library.md#824-stack)
  + [8.2.5 Queue](Chapter_8_Standard_Template_Library.md#825-queue)
  + [8.2.6 Priority Queue](Chapter_8_Standard_Template_Library.md#826-priority-queue)
  + [8.2.7 Set & Multiset](Chapter_8_Standard_Template_Library.md#827-set--multiset)
  + [8.2.8 Map & Multimap](Chapter_8_Standard_Template_Library.md#828-map--multimap)
* [8.3 Iterators](Chapter_8_Standard_Template_Library.md#83-iterators)
* [8.4 Algorithms (sort, find, count, etc.)](Chapter_8_Standard_Template_Library.md#84-algorithms-sort-find-count-etc)
* [8.5 Function Objects (Functors)](Chapter_8_Standard_Template_Library.md#85-function-objects-functors)
* [8.6 Custom Comparators](Chapter_8_Standard_Template_Library.md#86-custom-comparators)

## [Chapter 9: File Handling in C++](Chapter_9_File_Handling_in_CPP.md)

* [9.1 Streams (ifstream, ofstream, fstream)](Chapter_9_File_Handling_in_CPP.md#91-streams-ifstream-ofstream-fstream)
* [9.2 Reading and Writing Text Files](Chapter_9_File_Handling_in_CPP.md#92-reading-and-writing-text-files)
* [9.3 Binary File I/O](Chapter_9_File_Handling_in_CPP.md#93-binary-file-io)
* [9.4 File Pointers and Random Access](Chapter_9_File_Handling_in_CPP.md#94-file-pointers-and-random-access)
* [9.5 Error Handling in File Operations](Chapter_9_File_Handling_in_CPP.md#95-error-handling-in-file-operations)

## [Chapter 10: C++ and System-Level Programming](Chapter_10_CPP_and_System_Level_Programming.md)

* [10.1 Bit Manipulation](Chapter_10_CPP_and_System_Level_Programming.md#101-bit-manipulation)
* [10.2 Working with Low-Level Memory](Chapter_10_CPP_and_System_Level_Programming.md#102-working-with-low-level-memory)
* [10.3 Inline Assembly in C++](Chapter_10_CPP_and_System_Level_Programming.md#103-inline-assembly-in-c)
* [10.4 Interfacing with C Libraries](Chapter_10_CPP_and_System_Level_Programming.md#104-interfacing-with-c-libraries)
* [10.5 Case Study: Memory Layout of Objects](Chapter_10_CPP_and_System_Level_Programming.md#105-case-study-memory-layout-of-objects)
* [10.6 Multithreading (C++11 and beyond)](Chapter_10_CPP_and_System_Level_Programming.md#106-multithreading-c11-and-beyond)
  + [10.6.1 Threads, Mutexes, Locks](Chapter_10_CPP_and_System_Level_Programming.md#1061-threads-mutexes-locks)
  + [10.6.2 Deadlock and Race Conditions](Chapter_10_CPP_and_System_Level_Programming.md#1062-deadlock-and-race-conditions)
  + [10.6.3 Condition Variables](Chapter_10_CPP_and_System_Level_Programming.md#1063-condition-variables)

## [Chapter 11: C++ in the Real World](Chapter_11_CPP_in_the_Real_World.md)

* [11.1 Competitive Programming with C++](Chapter_11_CPP_in_the_Real_World.md#111-competitive-programming-with-c)
* [11.2 C++ in Game Engines](Chapter_11_CPP_in_the_Real_World.md#112-c-in-game-engines)
* [11.3 C++ for System Software (e.g., Windows, Office)](Chapter_11_CPP_in_the_Real_World.md#113-c-for-system-software-eg-windows-office)
* [11.4 C++ in Embedded Systems](Chapter_11_CPP_in_the_Real_World.md#114-c-in-embedded-systems)
* [11.5 C++ in Finance and Trading Systems](Chapter_11_CPP_in_the_Real_World.md#115-c-in-finance-and-trading-systems)
* [11.6 Designing a Real Project in C++ (e.g., File Manager, Chat App)](Chapter_11_CPP_in_the_Real_World.md#116-designing-a-real-project-in-c-eg-file-manager-chat-app)

# Chapter 1: Introduction to C++

## 1.1 What is C++?

C++ is a powerful, general-purpose programming language that extends the C programming language with object-oriented features. Created by Bjarne Stroustrup at Bell Labs in 1979, C++ was designed to be an efficient and flexible language that combines low-level memory manipulation with high-level abstractions.

**Key characteristics of C++:**

* **Multi-paradigm**: Supports procedural, object-oriented, functional, and generic programming styles
* **Statically typed**: Type checking is performed at compile-time
* **Performance-oriented**: Designed for systems where performance is critical
* **Direct hardware access**: Allows low-level memory manipulation similar to C
* **Rich standard library**: Provides extensive functionality through the Standard Template Library (STL)

C++ is often described as a “middle-level language,” as it combines features of both high-level and low-level programming languages, giving programmers precise control over system resources while supporting high-level abstractions.

## 1.2 History & Evolution of C++

The evolution of C++ spans several decades, with each version introducing significant improvements:

### Origins and Early Development

* **1979**: Bjarne Stroustrup begins work on “C with Classes” at Bell Labs
* **1983**: The language is renamed to C++, with “++” being the increment operator in C
* **1985**: First commercial C++ implementation released and Stroustrup’s book “The C++ Programming Language” published

### Standardization

* **1989**: ANSI committee formed to standardize C++
* **1998**: First ISO C++ standard (C++98) published, establishing the core language features
* **2003**: C++03 standard released with minor improvements and bug fixes

### Modern C++ Revolution

* **2011**: C++11 released, a major overhaul adding numerous features:
  + Auto type deduction
  + Lambda expressions
  + Smart pointers
  + Move semantics
  + Range-based for loops
  + Threading library
* **2014**: C++14 published with incremental improvements
* **2017**: C++17 added more features including:
  + File system library
  + Parallel algorithms
  + std::variant and std::optional
* **2020**: C++20 introduced:
  + Concepts
  + Ranges
  + Modules
  + Coroutines
* **2023**: C++23 (in progress)

This ongoing evolution has transformed C++ from a “C with Classes” to a highly sophisticated multi-paradigm language while maintaining backward compatibility and performance.

## 1.3 Why Use C++? (Use Cases & Industry Adoption)

C++ excels in scenarios requiring performance, resource control, and complex systems development.

### Key Strengths

* **Performance**: Near-native execution speed with minimal overhead
* **Memory Control**: Direct memory management for efficient resource utilization
* **Portability**: Code can run on various platforms with appropriate compilation
* **Scalability**: Suitable for both small applications and large-scale systems
* **Mature Ecosystem**: Extensive libraries, tools, and community support

### Industry Applications

#### Game Development

* **Used by**: Unreal Engine, Unity (parts), CryEngine
* **Examples**: AAA games like Fortnite, Gears of War, Far Cry
* **Why C++**: Provides the performance needed for real-time graphics, physics, and AI

#### Systems Programming

* **Operating Systems**: Windows, parts of macOS and Linux
* **Device Drivers**: Hardware interfaces requiring direct memory access
* **Embedded Systems**: Resource-constrained devices where efficiency is paramount

#### High-Performance Computing

* **Scientific Simulation**: Weather modeling, physics simulations
* **Financial Systems**: High-frequency trading platforms
* **Data Processing**: Big data processing frameworks

#### Application Software

* **Productivity Software**: Microsoft Office, Adobe Creative Suite
* **Database Systems**: MySQL, MongoDB (parts)
* **Web Browsers**: Chrome/Chromium (V8 engine), Firefox

#### Telecommunications

* **Network Infrastructure**: Routers, switches
* **Communication Protocols**: Implementation of networking stacks

The wide adoption of C++ across these diverse fields demonstrates its versatility and enduring value in modern software development.

## 1.4 C++ vs C vs Java vs Python

Understanding how C++ compares to other popular languages helps identify when to use each language effectively.

### C++ vs C

| Feature | C++ | C |
| --- | --- | --- |
| **Paradigm** | Multi-paradigm (OOP, procedural, generic) | Procedural |
| **Object-Orientation** | Yes, with classes and inheritance | No native support |
| **Memory Management** | Manual + RAII + smart pointers | Completely manual |
| **Standard Library** | Extensive (STL) | Basic |
| **Function Overloading** | Supported | Not supported |
| **Templates** | Supported | Not supported |
| **Exception Handling** | Yes | No (requires manual error codes) |
| **Performance** | High | High |
| **Complexity** | Higher | Lower |

**Key Difference**: C++ extends C with object-oriented features while maintaining compatibility with most C code.

### C++ vs Java

| Feature | C++ | Java |
| --- | --- | --- |
| **Compilation** | Compiles to native code | Compiles to bytecode, runs on JVM |
| **Memory Management** | Manual + smart pointers | Automatic garbage collection |
| **Performance** | Higher, direct hardware access | Somewhat lower due to JVM |
| **Portability** | Compile per platform | “Write once, run anywhere” |
| **Multiple Inheritance** | Supported | Only interface inheritance |
| **Pointers** | Direct pointer manipulation | References only, no direct pointers |
| **Platform Independence** | Less (requires recompilation) | More (through JVM) |
| **Operator Overloading** | Supported | Not supported |

**Key Difference**: C++ offers more control and performance but requires more careful memory management; Java provides better safety and platform independence.

### C++ vs Python

| Feature | C++ | Python |
| --- | --- | --- |
| **Typing** | Static typing | Dynamic typing |
| **Execution** | Compiled, fast execution | Interpreted, slower execution |
| **Syntax** | Verbose, complex | Concise, readable |
| **Learning Curve** | Steeper | Gentler |
| **Memory Management** | Manual | Automatic |
| **Use Cases** | System/performance-critical software | Scripting, web, data science |
| **Development Speed** | Slower, more boilerplate | Rapid development |
| **Library Ecosystem** | Extensive but less centralized | Rich, accessible via pip |

**Key Difference**: C++ prioritizes performance and control; Python prioritizes development speed and readability.

### When to Choose C++

* Performance-critical applications
* Systems programming
* Resource-constrained environments
* Large-scale complex systems requiring fine control
* Games and graphics-intensive applications

## 1.5 C++ Compilation Model

Understanding the compilation process is essential for effective C++ development.

### The Four Stages of C++ Compilation

#### 1. Preprocessing

* **Process**: Handles directives like #include, #define, and conditional compilation
* **Input**: Source code (.cpp files)
* **Output**: Modified source code with expanded macros and included files
* **Command**: g++ -E file.cpp -o file.i

// Before preprocessing  
#include <iostream>  
#define MAX 100  
  
// After preprocessing  
// (contents of iostream header)  
const int MAX = 100;

#### 2. Compilation

* **Process**: Translates preprocessed code to assembly language
* **Input**: Preprocessed code
* **Output**: Assembly code (.s files)
* **Command**: g++ -S file.i -o file.s

This stage performs: - Syntax checking - Semantic analysis - Code optimization - Generation of assembly code

#### 3. Assembly

* **Process**: Converts assembly code to machine code (object files)
* **Input**: Assembly code (.s files)
* **Output**: Object files (.o or .obj files)
* **Command**: g++ -c file.s -o file.o

Object files contain: - Machine code - Symbol tables - Relocation information - Debug information (if enabled)

#### 4. Linking

* **Process**: Combines multiple object files and libraries into an executable
* **Input**: Object files and libraries
* **Output**: Executable program
* **Command**: g++ file.o -o program

The linker performs: - Resolution of external references - Library linking - Address assignment - Generation of final executable

### Header Files and Translation Units

* **Translation Unit**: A source file after preprocessing (including all headers)
* **Header Files**: Contain declarations shared across multiple source files
* **Forward Declarations**: Allow referencing symbols before their definition
* **Include Guards**: Prevent multiple inclusions (#ifndef, #define, #endif or #pragma once)

### One Definition Rule (ODR)

The ODR states that: - Every entity must be defined exactly once in a program - Inline functions and template definitions can appear in multiple translation units - All definitions of the same entity must be identical

### Types of Linking

* **Static Linking**: Libraries are copied into the executable at compile time
  + Advantages: Self-contained executable
  + Disadvantages: Larger file size
* **Dynamic Linking**: Libraries are loaded at runtime
  + Advantages: Smaller executables, shared library code
  + Disadvantages: Dependency on external libraries

Understanding this compilation process helps diagnose compilation errors, organize code effectively, and optimize build times.

## 1.6 Writing and Running a Simple C++ Program

Let’s break down the process of creating and running your first C++ program.

### Setting Up Your Development Environment

#### For Windows:

1. **Install a compiler**:
   * Visual Studio Community (IDE + compiler)
   * MinGW or MinGW-w64 (GCC for Windows)
   * Clang for Windows
2. **IDE options**:
   * Visual Studio
   * Visual Studio Code with C++ extensions
   * CLion
   * Code::Blocks

#### For macOS:

1. **Install XCode** (includes Clang compiler)
2. **Alternative**: Install GCC via Homebrew (brew install gcc)

#### For Linux:

1. **Install GCC/G++**: sudo apt install build-essential (Ubuntu/Debian)
2. **IDE options**:
   * Visual Studio Code
   * CLion
   * Code::Blocks

### Your First Program: Hello World

1. **Create a file** named hello.cpp
2. **Write the code**:

#include <iostream>  
  
int main() {  
 std::cout << "Hello, World!" << std::endl;  
 return 0;  
}

1. **Explanation of this code**:
   * #include <iostream>: Imports the input/output stream library
   * int main(): Entry point function that returns an integer
   * std::cout << "Hello, World!" << std::endl;: Outputs text to the console
   * return 0;: Indicates successful program completion

### Compiling and Running

#### Using Command Line (GCC/G++):

# Compile  
g++ hello.cpp -o hello  
  
# Run  
./hello # Unix/Linux/macOS  
hello.exe # Windows

#### Using Visual Studio:

1. Create a new C++ project
2. Add your source file
3. Build the solution (F7)
4. Run the program (F5)

#### Using VS Code:

1. Install C/C++ extension
2. Configure tasks.json for building
3. Configure launch.json for debugging
4. Build with Ctrl+Shift+B
5. Run with F5

### Common Compilation Errors

* **Syntax errors**: Missing semicolons, incorrect brackets
* **Undefined references**: Missing function implementations
* **Include errors**: Missing or incorrect header files
* **Type errors**: Assigning incompatible types

### Best Practices for Beginners

1. **Incremental development**: Write, compile, and test small pieces at a time
2. **Consistent formatting**: Use consistent indentation and spacing
3. **Descriptive names**: Use meaningful variable and function names
4. **Comments**: Explain complex logic or intentions
5. **Version control**: Learn to use Git for tracking changes

### Program Structure

// Include libraries  
#include <iostream>  
#include <string>  
  
// Constants and global variables (use sparingly)  
const double PI = 3.14159;  
  
// Function declarations  
void greetUser(const std::string& name);  
  
// Main function  
int main() {  
 std::string userName = "Student";  
 greetUser(userName);  
 return 0;  
}  
  
// Function definitions  
void greetUser(const std::string& name) {  
 std::cout << "Hello, " << name << "! Welcome to C++." << std::endl;  
}

As you practice writing more complex programs, you’ll become familiar with this basic structure and expand upon it.

## 1.7 Input and Output (cin, cout)

C++ provides comprehensive I/O capabilities through streams, with cin and cout being the most commonly used.

### Stream-Based I/O

C++ uses the concept of streams for input and output operations:

* **Stream**: An abstraction representing a sequence of bytes
* **Input Stream**: Bytes flow from a device (keyboard, file) into your program
* **Output Stream**: Bytes flow from your program to a device (screen, file)

### Standard I/O Streams

The <iostream> header provides several predefined streams:

* std::cin: Standard input stream (usually keyboard)
* std::cout: Standard output stream (usually console)
* std::cerr: Standard error output stream (unbuffered)
* std::clog: Standard logging output stream (buffered)

### Output with cout

The cout object uses the insertion operator (<<) to output data:

#include <iostream>  
  
int main() {  
 // Basic output  
 std::cout << "Hello, world!" << std::endl;  
   
 // Multiple outputs can be chained  
 int age = 25;  
 std::cout << "I am " << age << " years old." << std::endl;  
   
 // Formatting options  
 double pi = 3.14159265359;  
 std::cout << "Pi is approximately " << pi << std::endl;  
   
 return 0;  
}

#### Output Manipulators

Manipulators modify the formatting of output:

#include <iostream>  
#include <iomanip> // Required for most manipulators  
  
int main() {  
 // Set precision (total digits)  
 std::cout << std::setprecision(4) << 3.14159 << std::endl; // 3.142  
   
 // Fixed precision (digits after decimal)  
 std::cout << std::fixed << std::setprecision(2);  
 std::cout << 3.14159 << std::endl; // 3.14  
   
 // Field width  
 std::cout << std::setw(10) << "Hello" << std::endl; // " Hello"  
   
 // Fill character  
 std::cout << std::setfill('-') << std::setw(10) << "Hello" << std::endl; // "-----Hello"  
   
 // Number base  
 std::cout << std::hex << 255 << std::endl; // ff  
 std::cout << std::oct << 255 << std::endl; // 377  
 std::cout << std::dec << 255 << std::endl; // 255  
   
 // Boolean format  
 std::cout << std::boolalpha << true << std::endl; // "true" instead of 1  
   
 return 0;  
}

### Input with cin

The cin object uses the extraction operator (>>) to read data:

#include <iostream>  
#include <string>  
  
int main() {  
 // Reading integers  
 int age;  
 std::cout << "Enter your age: ";  
 std::cin >> age;  
 std::cout << "You are " << age << " years old." << std::endl;  
   
 // Reading multiple values  
 int x, y;  
 std::cout << "Enter two numbers: ";  
 std::cin >> x >> y;  
 std::cout << "Sum: " << x + y << std::endl;  
   
 // Reading strings (word)  
 std::string name;  
 std::cout << "Enter your name: ";  
 std::cin >> name; // Only reads until the first whitespace  
 std::cout << "Hello, " << name << "!" << std::endl;  
   
 // Clear the input buffer  
 std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');  
   
 // Reading a full line  
 std::string fullName;  
 std::cout << "Enter your full name: ";  
 std::getline(std::cin, fullName);  
 std::cout << "Hello, " << fullName << "!" << std::endl;  
   
 return 0;  
}

### Common Input Issues

1. **Input validation**: Always check if input was successful
2. **Type mismatch**: Entering a string when expecting a number
3. **Buffer issues**: Mixing >> and getline()

#include <iostream>  
#include <string>  
#include <limits>  
  
int main() {  
 int age;  
 std::string name;  
   
 // Read age with validation  
 std::cout << "Enter your age: ";  
 while (!(std::cin >> age) || age < 0) {  
 std::cout << "Invalid input. Enter a positive number: ";  
 std::cin.clear(); // Clear error flags  
 std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n'); // Clear buffer  
 }  
   
 // Clear buffer before getline  
 std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');  
   
 // Read name  
 std::cout << "Enter your name: ";  
 std::getline(std::cin, name);  
   
 std::cout << "Hello " << name << " (" << age << " years old)!" << std::endl;  
   
 return 0;  
}

### File I/O

For file operations, use <fstream> which provides: - std::ifstream: Input file stream - std::ofstream: Output file stream - std::fstream: Input/output file stream

#include <iostream>  
#include <fstream>  
#include <string>  
  
int main() {  
 // Writing to a file  
 std::ofstream outFile("example.txt");  
 if (outFile.is\_open()) {  
 outFile << "Hello, file I/O!" << std::endl;  
 outFile << "This is a second line." << std::endl;  
 outFile.close();  
 } else {  
 std::cerr << "Failed to open file for writing!" << std::endl;  
 }  
   
 // Reading from a file  
 std::ifstream inFile("example.txt");  
 if (inFile.is\_open()) {  
 std::string line;  
 while (std::getline(inFile, line)) {  
 std::cout << line << std::endl;  
 }  
 inFile.close();  
 } else {  
 std::cerr << "Failed to open file for reading!" << std::endl;  
 }  
   
 return 0;  
}

### Advanced Stream Operations

* **String streams**: Convert between strings and other types
* **Binary I/O**: Read and write raw binary data
* **Custom stream manipulators**: Create your own formatting tools
* **Stream states**: Check for errors during I/O operations

#include <iostream>  
#include <sstream>  
#include <iomanip>  
  
int main() {  
 // String streams for conversion  
 std::stringstream ss;  
 ss << "Age: " << 25 << ", Height: " << std::fixed << std::setprecision(2) << 1.85;  
 std::string result = ss.str();  
 std::cout << result << std::endl;  
   
 // Parsing with string streams  
 std::stringstream parser("123 456 789");  
 int a, b, c;  
 parser >> a >> b >> c;  
 std::cout << "Sum: " << (a + b + c) << std::endl;  
   
 return 0;  
}

Understanding these I/O mechanisms is crucial for almost all C++ programs, as they provide the primary means of interaction with users and external data.

## 1.8 Basic Syntax & Program Structure

Understanding C++’s core syntax and structure is fundamental to writing effective programs.

### Program Structure Components

A typical C++ program consists of:

1. **Preprocessor directives**: Instructions that run before compilation
2. **Comments**: Non-executable annotations for code documentation
3. **Functions**: Reusable blocks of code
4. **Variables**: Named storage locations
5. **Statements**: Instructions that perform operations
6. **Expressions**: Combinations of values, variables, operators

### Preprocessor Directives

Preprocessor directives begin with # and are processed before compilation:

// Include standard library headers  
#include <iostream> // Angle brackets for standard libraries  
#include "myheader.h" // Quotes for user-defined headers  
  
// Define constants and macros  
#define PI 3.14159  
#define SQUARE(x) ((x) \* (x))  
  
// Conditional compilation  
#ifdef DEBUG  
 #define LOG(msg) std::cout << msg << std::endl  
#else  
 #define LOG(msg)  
#endif

### Comments

C++ supports two types of comments:

// Single-line comment  
  
/\* Multi-line comment  
 that can span  
 several lines \*/  
  
/\*\*  
 \* Documentation comments (not standard C++, but widely used)  
 \* @param x Description of parameter  
 \* @return Description of return value  
 \*/

### Variables and Data Types

Variables must be declared before use:

// Basic variable declarations  
int count = 0; // Integer  
double temperature = 98.6; // Floating-point  
char grade = 'A'; // Single character  
bool isActive = true; // Boolean  
  
// Constants (immutable values)  
const int MAX\_USERS = 100;

### Basic Operators

C++ provides various operators for different operations:

// Assignment  
int x = 5;  
  
// Arithmetic  
int sum = 5 + 3; // Addition  
int difference = 5 - 3; // Subtraction  
int product = 5 \* 3; // Multiplication  
int quotient = 5 / 3; // Integer division (result: 1)  
int remainder = 5 % 3; // Modulus (result: 2)  
  
// Compound assignment  
x += 2; // Equivalent to x = x + 2  
  
// Increment/decrement  
x++; // Postfix increment  
++x; // Prefix increment  
x--; // Postfix decrement  
--x; // Prefix decrement  
  
// Comparison  
bool isEqual = (x == 5); // Equality  
bool isNotEqual = (x != 5); // Inequality  
bool isGreater = (x > 5); // Greater than  
bool isLessOrEqual = (x <= 5); // Less than or equal  
  
// Logical  
bool result = (x > 0 && x < 10); // Logical AND  
bool anotherResult = (x < 0 || x > 10); // Logical OR  
bool negation = !isActive; // Logical NOT

### Functions

Functions encapsulate reusable code:

// Function declaration (prototype)  
int add(int a, int b);  
  
// Function definition  
int add(int a, int b) {  
 return a + b;  
}  
  
// Function with no return value  
void printMessage(const std::string& message) {  
 std::cout << message << std::endl;  
}  
  
// Default parameters  
void greet(const std::string& name = "Guest") {  
 std::cout << "Hello, " << name << "!" << std::endl;  
}  
  
// Main function (program entry point)  
int main() {  
 int result = add(5, 3);  
 printMessage("The result is: " + std::to\_string(result));  
 greet(); // Uses default parameter  
 greet("Alice"); // Uses provided parameter  
 return 0; // Return 0 indicates successful execution  
}

### Namespaces

Namespaces help avoid name conflicts:

// Declaring a namespace  
namespace Math {  
 const double PI = 3.14159;  
   
 double square(double x) {  
 return x \* x;  
 }  
}  
  
// Using a namespace  
int main() {  
 // Fully qualified name  
 double area = Math::PI \* Math::square(2.0);  
   
 // Using directive (avoid in larger programs)  
 using namespace Math;  
 double circumference = 2 \* PI \* 2.0;  
   
 // Using declaration (preferred)  
 using Math::PI;  
 double anotherArea = PI \* 4.0;  
   
 return 0;  
}

### Basic Program Flow

C++ programs execute statements sequentially, with control structures altering the flow:

#include <iostream>  
  
int main() {  
 // Sequential execution  
 std::cout << "Starting program..." << std::endl;  
   
 int x = 10;  
 x = x + 5;  
   
 // Conditional execution (covered in detail in Chapter 3)  
 if (x > 10) {  
 std::cout << "x is greater than 10" << std::endl;  
 }  
   
 // Loop (covered in detail in Chapter 3)  
 for (int i = 0; i < 3; i++) {  
 std::cout << "Loop iteration: " << i << std::endl;  
 }  
   
 // Function call  
 std::cout << "Program finished." << std::endl;  
 return 0;  
}

### Scope and Lifetime

Variables have different scopes and lifetimes:

#include <iostream>  
  
// Global scope  
int globalVar = 100;  
  
void demoScope() {  
 // Function scope  
 int functionVar = 200;  
   
 {  
 // Block scope  
 int blockVar = 300;  
 std::cout << "Inside block: " << globalVar << ", "   
 << functionVar << ", " << blockVar << std::endl;  
 }  
   
 // blockVar is no longer accessible here  
 std::cout << "Inside function: " << globalVar << ", "   
 << functionVar << std::endl;  
}  
  
int main() {  
 demoScope();  
 // functionVar is not accessible here  
 std::cout << "In main: " << globalVar << std::endl;  
 return 0;  
}

### Coding Style Conventions

While C++ doesn’t enforce a specific style, consistent formatting improves readability:

* **Indentation**: Typically 2 or 4 spaces
* **Braces**: Opening brace on same line or next line (be consistent)
* **Naming**: camelCase or snake\_case for variables/functions, PascalCase for classes
* **Constants**: Often ALL\_CAPS with underscores
* **Comments**: Add meaningful comments explaining “why” not just “what”

### Error Handling Basics

C++ provides mechanisms for handling errors:

#include <iostream>  
#include <stdexcept>  
  
double divide(double a, double b) {  
 if (b == 0) {  
 throw std::runtime\_error("Division by zero");  
 }  
 return a / b;  
}  
  
int main() {  
 try {  
 double result = divide(10, 2);  
 std::cout << "Result: " << result << std::endl;  
   
 result = divide(10, 0); // This will throw an exception  
 }  
 catch (const std::exception& e) {  
 std::cerr << "Error: " << e.what() << std::endl;  
 }  
   
 return 0;  
}

### Best Practices

1. **Keep functions short and focused** on a single task
2. **Avoid global variables** when possible
3. **Initialize variables** when declaring them
4. **Use meaningful names** that clearly indicate purpose
5. **Include proper error handling** for robust programs
6. **Comment your code** to explain complex logic or decisions
7. **Follow consistent formatting** throughout your code

Understanding these fundamental syntax elements and structure provides the foundation for writing effective C++ programs. As you progress, you’ll build on these basics to create more complex and powerful applications.

# Chapter 2: Variables, Data Types & Operators

## 2.1 Variable Declaration & Initialization

Variables are named storage locations in memory that hold values that can be modified during program execution. In C++, every variable must be declared before it can be used.

### Declaration Syntax

data\_type variable\_name; // Declaration  
data\_type variable\_name = initial\_value; // Declaration with initialization

### Examples

int count; // Declaration  
int score = 100; // Declaration with initialization  
double price = 45.60, tax = 5.5; // Multiple declarations  
char grade = 'A'; // Character initialization  
bool isActive = true; // Boolean initialization

### Variable Naming Rules

* Must begin with a letter or underscore (\_)
* Can consist of letters, digits, and underscores
* Cannot use C++ keywords (like int, class, return, etc.)
* C++ is case-sensitive (count and Count are different variables)

### Best Practices

* Use meaningful names that indicate the variable’s purpose
* Follow a consistent naming convention (e.g., camelCase or snake\_case)
* Initialize variables when declared to avoid undefined behavior
* Keep variables scoped as tightly as possible

### Memory Sizes

Memory allocated depends on the data type and platform:

#include <iostream>  
using namespace std;  
  
int main() {  
 cout << "Size of int: " << sizeof(int) << " bytes" << endl;  
 cout << "Size of float: " << sizeof(float) << " bytes" << endl;  
 cout << "Size of double: " << sizeof(double) << " bytes" << endl;  
 cout << "Size of char: " << sizeof(char) << " byte" << endl;  
 cout << "Size of bool: " << sizeof(bool) << " byte" << endl;  
 return 0;  
}

### Variable Scope and Lifetime

* **Local variables**: Declared within a function or block, exist only within that scope
* **Global variables**: Declared outside any function, accessible throughout the program
* **Static variables**: Retain value between function calls

#include <iostream>  
using namespace std;  
  
int globalVar = 10; // Global variable  
  
void demoFunction() {  
 static int staticVar = 0; // Static variable  
 int localVar = 5; // Local variable  
   
 staticVar++;  
 localVar++;  
   
 cout << "Global: " << globalVar << ", Static: " << staticVar << ", Local: " << localVar << endl;  
}  
  
int main() {  
 demoFunction(); // Output: Global: 10, Static: 1, Local: 6  
 demoFunction(); // Output: Global: 10, Static: 2, Local: 6  
 demoFunction(); // Output: Global: 10, Static: 3, Local: 6  
 return 0;  
}

## 2.2 Data Types: Primitive & Derived

C++ offers various data types that can be broadly categorized as primitive (built-in) and derived (user-defined).

### Primitive Data Types

#### Integer Types

* **int**: Standard integer type (-2,147,483,648 to 2,147,483,647)
* **short**: Small integer (-32,768 to 32,767)
* **long**: Large integer (at least 32 bits)
* **long long**: Very large integer (at least 64 bits)
* **unsigned** variants: Store only positive numbers, doubling the positive range

int regular = 42;  
short small = 32767;  
long large = 2147483647L; // Note the 'L' suffix  
long long veryLarge = 9223372036854775807LL; // Note the 'LL' suffix  
unsigned int positiveOnly = 4294967295U; // Note the 'U' suffix

#### Floating Point Types

* **float**: Single precision (7 digits precision), ~1.2E-38 to ~3.4E+38
* **double**: Double precision (15 digits precision), ~2.3E-308 to ~1.7E+308
* **long double**: Extended precision (platform dependent)

float price = 19.99f; // Note the 'f' suffix  
double pi = 3.14159265359;  
long double veryPrecise = 3.14159265359L; // Note the 'L' suffix

#### Character Types

* **char**: Single character (1 byte)
* **wchar\_t**: Wide character (typically 2 or 4 bytes)
* **char16\_t**: UTF-16 character (C++11)
* **char32\_t**: UTF-32 character (C++11)

char letter = 'A';  
wchar\_t wideLetter = L'Ω'; // Note the 'L' prefix  
char16\_t utf16Char = u'π'; // Note the 'u' prefix  
char32\_t utf32Char = U'😊'; // Note the 'U' prefix

#### Boolean Type

* **bool**: Represents true or false values

bool isValid = true;  
bool isEmpty = false;

#### Void Type

* **void**: Represents absence of type, used primarily for functions that don’t return a value

### Modified Data Types

C++ provides modifiers to alter the range and memory usage of basic types:

* **signed**: Can represent both positive and negative numbers (default for int)
* **unsigned**: Can only represent positive numbers
* **short**: Reduces size
* **long**: Increases size

signed int canBeNegative = -42; // Same as just "int"  
unsigned int onlyPositive = 42;  
short int smallRange = 100;  
long int largeRange = 1000000L;

### Derived Data Types

#### Arrays

Collection of elements of the same type.

int numbers[5] = {1, 2, 3, 4, 5};  
char name[10] = "C++";

#### Pointers

Store memory addresses of variables.

int\* ptr;  
int value = 42;  
ptr = &value; // ptr now holds the address of value

#### References

An alias for an existing variable.

int original = 10;  
int& ref = original; // ref is a reference to original  
ref = 20; // Changes original to 20 as well

#### User-Defined Types

* **struct**: Collection of variables of different types

struct Person {  
 string name;  
 int age;  
 double height;  
};  
  
Person person1 = {"John Doe", 30, 5.9};

* **class**: Foundation of object-oriented programming

class BankAccount {  
private:  
 double balance;  
public:  
 BankAccount(double initialBalance) : balance(initialBalance) {}  
 void deposit(double amount) { balance += amount; }  
 double getBalance() { return balance; }  
};

* **union**: Stores different data types in the same memory location

union Data {  
 int i;  
 float f;  
 char str[20];  
};  
  
Data data;  
data.i = 10; // Now data.i is active  
data.f = 3.14f; // data.i is no longer valid, data.f is active

* **enum**: User-defined type consisting of named constants

enum Color {RED, GREEN, BLUE};  
Color myColor = RED;

### Type Aliases

C++ allows creating aliases for types:

// Using typedef (traditional approach)  
typedef unsigned long ulong;  
ulong counter = 0;  
  
// Using using (modern approach, C++11)  
using Integer = int;  
Integer value = 42;

## 2.3 Type Conversion & Typecasting

Type conversion refers to converting one data type to another. It can happen implicitly (automatically) or explicitly (through casting).

### Implicit Type Conversion (Automatic)

C++ automatically converts between types when an operation involves different data types:

int i = 10;  
double d = i; // Implicit conversion from int to double (d = 10.0)  
  
char c = 'A';  
int ascii = c; // Implicit conversion from char to int (ascii = 65)  
  
bool b = 42; // Implicit conversion from int to bool (b = true)

#### Conversion Rules

1. During an expression evaluation, smaller data types are converted to larger data types
2. Integral types (int, char, etc.) are converted to floating-point types when mixed
3. The result of an operation has the type of the most “capacious” operand

#### Promotion

Implicit conversion of smaller types to larger types:

short s = 10;  
int i = s; // Short to int  
float f = i; // Int to float  
double d = f; // Float to double

#### Demotion

Implicit conversion of larger types to smaller types (may result in data loss):

double d = 9.99;  
int i = d; // i becomes 9, fractional part is truncated

### Explicit Type Conversion (Typecasting)

C++ provides several ways to explicitly convert data types:

#### C-style Cast

double d = 3.14159;  
int i = (int)d; // i becomes 3

#### Function-style Cast

float f = 3.14f;  
int i = int(f); // i becomes 3

#### C++ Style Cast Operators

1. **static\_cast**: For “well-behaved” conversions

double d = 3.14159;  
int i = static\_cast<int>(d); // i becomes 3

1. **dynamic\_cast**: For safe downcasting in inheritance hierarchies

Base\* basePtr = new Derived();  
Derived\* derivedPtr = dynamic\_cast<Derived\*>(basePtr); // Safe conversion, checks at runtime

1. **const\_cast**: To add or remove const qualifier

const int value = 10;  
int\* ptr = const\_cast<int\*>(&value); // Removes const

1. **reinterpret\_cast**: For low-level reinterpreting of bit patterns

int\* p = new int(65);  
char\* ch = reinterpret\_cast<char\*>(p); // Dangerous, reinterprets memory

### Potential Issues with Type Conversion

#### Overflow

int largeValue = 2147483647; // Maximum int value  
largeValue++; // Overflow: largeValue becomes -2147483648

#### Precision Loss

double precise = 3.14159265359;  
float lessPrecise = precise; // Precision is lost

#### Sign Issues

unsigned int positive = 10;  
int value = -5;  
if (positive < value) {  
 cout << "This will execute unexpectedly!"; // This executes because -5 converts to a large unsigned value  
}

### Best Practices

1. Avoid mixing signed and unsigned types in expressions
2. Prefer C++ style casts over C-style casts
3. Be aware of potential data loss when converting between types
4. Use explicit casting when the conversion may not be intuitive
5. Use numeric\_limits to check value ranges before conversion

#include <iostream>  
#include <limits>  
using namespace std;  
  
int main() {  
 double d = 123456789.0;  
 if (d <= numeric\_limits<int>::max() && d >= numeric\_limits<int>::min()) {  
 int i = static\_cast<int>(d);  
 cout << "Safe conversion: " << i << endl;  
 } else {  
 cout << "Unsafe conversion: value out of range" << endl;  
 }  
 return 0;  
}

## 2.4 Operators in C++

Operators allow you to perform various operations on variables and values. C++ provides a rich set of operators.

### 2.4.1 Arithmetic Operators

Arithmetic operators perform mathematical calculations.

| Operator | Description | Example |
| --- | --- | --- |
| + | Addition | a + b |
| - | Subtraction | a - b |
| \* | Multiplication | a \* b |
| / | Division | a / b |
| % | Modulus (remainder) | a % b |
| ++ | Increment | a++ or ++a |
| -- | Decrement | a-- or --a |

#### Examples

#include <iostream>  
using namespace std;  
  
int main() {  
 int a = 10, b = 3;  
   
 cout << "a + b = " << (a + b) << endl; // 13  
 cout << "a - b = " << (a - b) << endl; // 7  
 cout << "a \* b = " << (a \* b) << endl; // 30  
 cout << "a / b = " << (a / b) << endl; // 3 (integer division)  
 cout << "a % b = " << (a % b) << endl; // 1 (remainder)  
   
 int c = a++; // c = 10, then a becomes 11 (post-increment)  
 int d = ++b; // b becomes 4, then d = 4 (pre-increment)  
   
 cout << "a after post-increment: " << a << endl; // 11  
 cout << "b after pre-increment: " << b << endl; // 4  
 cout << "c after assigning a++: " << c << endl; // 10  
 cout << "d after assigning ++b: " << d << endl; // 4  
   
 return 0;  
}

#### Important Notes

* Integer division truncates decimal part: 5 / 2 gives 2, not 2.5
* Modulus works only with integers and gives the remainder: 5 % 2 gives 1
* Pre-increment (++a) increments first, then uses the value
* Post-increment (a++) uses the current value, then increments
* Division by zero causes runtime errors

### 2.4.2 Relational Operators

Relational operators compare two values and return a boolean result.

| Operator | Description | Example |
| --- | --- | --- |
| == | Equal to | a == b |
| != | Not equal to | a != b |
| > | Greater than | a > b |
| < | Less than | a < b |
| >= | Greater than or equal to | a >= b |
| <= | Less than or equal to | a <= b |

#### Examples

#include <iostream>  
using namespace std;  
  
int main() {  
 int a = 10, b = 20;  
   
 cout << "a == b: " << (a == b) << endl; // 0 (false)  
 cout << "a != b: " << (a != b) << endl; // 1 (true)  
 cout << "a > b: " << (a > b) << endl; // 0 (false)  
 cout << "a < b: " << (a < b) << endl; // 1 (true)  
 cout << "a >= b: " << (a >= b) << endl; // 0 (false)  
 cout << "a <= b: " << (a <= b) << endl; // 1 (true)  
   
 return 0;  
}

#### Best Practices

* Use == for equality comparison, not = (which is assignment)
* Be careful when comparing floating-point numbers due to precision issues
* Use appropriate epsilon values for floating-point comparisons
* Avoid comparing unsigned with signed integers

// Proper floating-point comparison  
#include <cmath>  
#include <iostream>  
using namespace std;  
  
int main() {  
 double a = 0.1 + 0.2;  
 double b = 0.3;  
   
 // Direct comparison may fail due to precision  
 cout << "a == b: " << (a == b) << endl; // May print 0 (false)  
   
 // Better approach with epsilon  
 const double epsilon = 1e-9;  
 cout << "Proper comparison: " << (abs(a - b) < epsilon) << endl; // 1 (true)  
   
 return 0;  
}

### 2.4.3 Logical Operators

Logical operators combine boolean expressions.

| Operator | Description | Example |
| --- | --- | --- |
| && | Logical AND | a && b |
| \|\| | Logical OR | a \|\| b |
| ! | Logical NOT | !a |

#### Examples

#include <iostream>  
using namespace std;  
  
int main() {  
 bool a = true, b = false;  
   
 cout << "a && b: " << (a && b) << endl; // 0 (false)  
 cout << "a || b: " << (a || b) << endl; // 1 (true)  
 cout << "!a: " << (!a) << endl; // 0 (false)  
 cout << "!b: " << (!b) << endl; // 1 (true)  
   
 // Compound expressions  
 cout << "(a || b) && (!b): " << ((a || b) && (!b)) << endl; // 1 (true)  
   
 // Short-circuit evaluation  
 int x = 5, y = 0;  
 if (y != 0 && x / y > 2) { // y != 0 is false, so x / y is not evaluated (prevents division by zero)  
 cout << "This won't execute" << endl;  
 }  
   
 return 0;  
}

#### Short-Circuit Evaluation

* In expr1 && expr2, if expr1 is false, expr2 is not evaluated
* In expr1 || expr2, if expr1 is true, expr2 is not evaluated
* This prevents potential errors like division by zero or null pointer access

#### Truth Table

| A | B | A && B | A || B | !A |
| --- | --- | --- | --- | --- |
| T | T | T | T | F |
| T | F | F | T | F |
| F | T | F | T | T |
| F | F | F | F | T |

### 2.4.4 Bitwise Operators

Bitwise operators perform operations at the binary (bit) level.

| Operator | Description | Example |
| --- | --- | --- |
| & | Bitwise AND | a & b |
| \| | Bitwise OR | a \| b |
| ^ | Bitwise XOR | a ^ b |
| ~ | Bitwise NOT | ~a |
| << | Left shift | a << n |
| >> | Right shift | a >> n |

#### Examples

#include <iostream>  
#include <bitset> // For binary representation  
using namespace std;  
  
int main() {  
 int a = 5; // 0101 in binary  
 int b = 3; // 0011 in binary  
   
 cout << "Binary a: " << bitset<8>(a) << endl;  
 cout << "Binary b: " << bitset<8>(b) << endl;  
   
 cout << "a & b: " << (a & b) << " (" << bitset<8>(a & b) << ")" << endl; // 1 (0001)  
 cout << "a | b: " << (a | b) << " (" << bitset<8>(a | b) << ")" << endl; // 7 (0111)  
 cout << "a ^ b: " << (a ^ b) << " (" << bitset<8>(a ^ b) << ")" << endl; // 6 (0110)  
 cout << "~a: " << (~a) << " (" << bitset<8>(~a) << ")" << endl; // -6  
 cout << "a << 1: " << (a << 1) << " (" << bitset<8>(a << 1) << ")" << endl; // 10 (1010)  
 cout << "a >> 1: " << (a >> 1) << " (" << bitset<8>(a >> 1) << ")" << endl; // 2 (0010)  
   
 return 0;  
}

#### Common Bitwise Applications

1. **Setting a bit**: number |= (1 << position)
2. **Clearing a bit**: number &= ~(1 << position)
3. **Toggling a bit**: number ^= (1 << position)
4. **Checking a bit**: (number & (1 << position)) != 0
5. **Power of 2 multiplication**: number << n (multiplies by 2ⁿ)
6. **Power of 2 division**: number >> n (divides by 2ⁿ)

int num = 42; // 00101010 in binary  
int position = 3;  
  
// Set bit at position 3  
num |= (1 << position); // num becomes 50 (00110010)  
  
// Clear bit at position 3  
num &= ~(1 << position); // num becomes 42 again (00101010)  
  
// Toggle bit at position 3  
num ^= (1 << position); // num becomes 50 (00110010)  
  
// Check bit at position 3  
bool isBitSet = (num & (1 << position)) != 0; // true

#### Bit Manipulation Techniques

* **Swapping without temp variable**:

a ^= b;  
b ^= a;  
a ^= b;

* **Finding the rightmost set bit**:

int rightmostSetBit = num & -num;

* **Counting set bits (Brian Kernighan’s Algorithm)**:

int countSetBits(int n) {  
 int count = 0;  
 while (n) {  
 n &= (n - 1); // Clear the rightmost set bit  
 count++;  
 }  
 return count;  
}

### 2.4.5 Assignment Operators

Assignment operators assign values to variables.

| Operator | Description | Example | Equivalent |
| --- | --- | --- | --- |
| = | Simple assignment | a = b | a = b |
| += | Add and assign | a += b | a = a + b |
| -= | Subtract and assign | a -= b | a = a - b |
| \*= | Multiply and assign | a \*= b | a = a \* b |
| /= | Divide and assign | a /= b | a = a / b |
| %= | Modulus and assign | a %= b | a = a % b |
| &= | Bitwise AND and assign | a &= b | a = a & b |
| \|= | Bitwise OR and assign | a \|= b | a = a \| b |
| ^= | Bitwise XOR and assign | a ^= b | a = a ^ b |
| <<= | Left shift and assign | a <<= b | a = a << b |
| >>= | Right shift and assign | a >>= b | a = a >> b |

#### Examples

#include <iostream>  
using namespace std;  
  
int main() {  
 int a = 10;  
   
 a += 5; // a becomes 15  
 cout << "After a += 5: a = " << a << endl;  
   
 a -= 3; // a becomes 12  
 cout << "After a -= 3: a = " << a << endl;  
   
 a \*= 2; // a becomes 24  
 cout << "After a \*= 2: a = " << a << endl;  
   
 a /= 4; // a becomes 6  
 cout << "After a /= 4: a = " << a << endl;  
   
 a %= 4; // a becomes 2  
 cout << "After a %= 4: a = " << a << endl;  
   
 a <<= 3; // a becomes 16  
 cout << "After a <<= 3: a = " << a << endl;  
   
 a >>= 2; // a becomes 4  
 cout << "After a >>= 2: a = " << a << endl;  
   
 return 0;  
}

#### Chained Assignments

int x, y, z;  
x = y = z = 10; // All three variables get value 10

#### Compound Assignment Benefits

1. More concise code
2. Potentially more efficient (the expression is evaluated once)
3. Less prone to typing errors

### 2.4.6 Ternary Operator

The ternary conditional operator ?: is the only operator in C++ that takes three operands.

**Syntax**: condition ? expression1 : expression2

* If condition is true, expression1 is evaluated
* If condition is false, expression2 is evaluated

#### Examples

#include <iostream>  
using namespace std;  
  
int main() {  
 // Simple usage  
 int a = 10, b = 20;  
 int max = (a > b) ? a : b;  
 cout << "Maximum: " << max << endl; // 20  
   
 // Using with cout  
 cout << "a is " << ((a % 2 == 0) ? "even" : "odd") << endl; // a is even  
   
 // Nested ternary  
 int x = 5;  
 string result = (x > 10) ? "Greater than 10" :   
 (x > 5) ? "Between 6 and 10" :  
 (x == 5) ? "Equal to 5" : "Less than 5";  
 cout << result << endl; // Equal to 5  
   
 // Assignment with ternary  
 int score = 85;  
 char grade = (score >= 90) ? 'A' :   
 (score >= 80) ? 'B' :  
 (score >= 70) ? 'C' :  
 (score >= 60) ? 'D' : 'F';  
 cout << "Grade: " << grade << endl; // B  
   
 return 0;  
}

#### Benefits Over if-else

* More concise for simple conditions
* Can be used in expressions where if-else can’t (like initializations)
* Often allows assigning a value in a single line

#### Best Practices

* Use for simple conditions where readability isn’t compromised
* Avoid deeply nested ternary operations as they can be hard to read
* Use parentheses to make the intent clear
* Consider if-else for complex conditions for better readability

// Less readable  
string status = age < 13 ? "child" : age < 20 ? "teenager" : age < 65 ? "adult" : "senior";  
  
// More readable  
string status;  
if (age < 13) status = "child";  
else if (age < 20) status = "teenager";  
else if (age < 65) status = "adult";  
else status = "senior";

### 2.4.7 Operator Precedence & Associativity

Operator precedence determines the order in which operations are performed when an expression has multiple operators. Associativity determines the order of operations with the same precedence level.

#### Common Precedence Levels (from highest to lowest)

| Precedence | Operator | Description | Associativity |
| --- | --- | --- | --- |
| 1 | :: | Scope resolution | Left to right |
| 2 | ++ -- (postfix) | Postfix increment/decrement | Left to right |
| 2 | () | Function call | Left to right |
| 2 | [] | Array subscript | Left to right |
| 2 | . -> | Member access | Left to right |
| 3 | ++ -- (prefix) | Prefix increment/decrement | Right to left |
| 3 | + - (unary) | Unary plus/minus | Right to left |
| 3 | ! ~ | Logical NOT, Bitwise NOT | Right to left |
| 3 | (type) | Type cast | Right to left |
| 3 | \* | Dereference | Right to left |
| 3 | & | Address-of | Right to left |
| 3 | sizeof | Size-of | Right to left |
| 3 | new delete | Dynamic memory allocation | Right to left |
| 4 | \* / % | Multiplication, Division, Modulus | Left to right |
| 5 | + - | Addition, Subtraction | Left to right |
| 6 | << >> | Bitwise shift | Left to right |
| 7 | < <= > >= | Relational operators | Left to right |
| 8 | == != | Equality operators | Left to right |
| 9 | & | Bitwise AND | Left to right |
| 10 | ^ | Bitwise XOR | Left to right |
| 11 | \| | Bitwise OR | Left to right |
| 12 | && | Logical AND | Left to right |
| 13 | \|\| | Logical OR | Left to right |
| 14 | ?: | Ternary conditional | Right to left |
| 15 | = += -= etc. | Assignment operators | Right to left |
| 16 | , | Comma | Left to right |

#### Examples

#include <iostream>  
using namespace std;  
  
int main() {  
 // Example 1: Arithmetic precedence  
 int result1 = 10 + 5 \* 2; // \* has higher precedence than +  
 cout << "10 + 5 \* 2 = " << result1 << endl; // 20, not 30  
   
 // Example 2: Using parentheses to override precedence  
 int result2 = (10 + 5) \* 2;  
 cout << "(10 + 5) \* 2 = " << result2 << endl; // 30  
   
 // Example 3: Assignment vs equality  
 int x = 5;  
 if (x = 10) { // Assignment, not comparison! Sets x to 10, then evaluates to 10 (true)  
 cout << "This will execute because x = 10 is an assignment that returns 10" << endl;  
 }  
   
 // Example 4: Associativity  
 int a = 10, b = 5, c = 2;  
 int result3 = a - b - c; // Left-to-right: (a - b) - c  
 cout << "10 - 5 - 2 = " << result3 << endl; // 3, not 7  
   
 // Example 5: Mixed operators  
 int result4 = 10 \* 5 + 3 \* 2; // (10 \* 5) + (3 \* 2)  
 cout << "10 \* 5 + 3 \* 2 = " << result4 << endl; // 56  
   
 // Example 6: Complex expression  
 int result5 = 5 + 3 \* 8 / 4 - 2; // 5 + ((3 \* 8) / 4) - 2  
 cout << "5 + 3 \* 8 / 4 - 2 = " << result5 << endl; // 9  
   
 // Example 7: Assignment associativity (right to left)  
 int d, e, f;  
 d = e = f = 42; // f = 42, then e = 42, then d = 42  
 cout << "d = " << d << ", e = " << e << ", f = " << f << endl;  
   
 return 0;  
}

#### Common Precedence Mistakes

1. **Assignment vs. Equality**

// Wrong  
if (x = 5) { ... } // Assigns 5 to x, then evaluates to true  
  
// Correct  
if (x == 5) { ... } // Compares x with 5

1. **Boolean Logic Order**

// This condition might not work as expected  
if (a > 0 && b > 0 || c > 0 && d > 0) { ... }  
  
// Better with explicit parentheses  
if ((a > 0 && b > 0) || (c > 0 && d > 0)) { ... }

1. **Increment/Decrement with Other Operations**

int i = 5;  
int j = i++ + ++i; // Undefined behavior due to multiple modifications  
  
// Better approach: separate the operations  
int i = 5;  
int temp = i++; // temp = 5, i = 6  
int j = temp + ++i; // j = 5 + 7 = 12, i = 7

#### Best Practices

1. Use parentheses to make your intentions clear and avoid relying on operator precedence rules
2. Break complex expressions into simpler ones using intermediate variables
3. Be particularly careful with:
   * Assignment in conditions
   * Bitwise operators vs logical operators
   * Increment/decrement operators in complex expressions
4. When in doubt, add parentheses to clarify the order of operations

// Unclear  
if (a == b && c == d || e == f && g == h) { ... }  
  
// Clear  
if (((a == b) && (c == d)) || ((e == f) && (g == h))) { ... }  
  
// Or even better: break into smaller pieces  
bool condition1 = (a == b) && (c == d);  
bool condition2 = (e == f) && (g == h);  
if (condition1 || condition2) { ... }

By understanding operator precedence and associativity, you can write more reliable and bug-free code, while also making your intentions clearer to others who read your code.

# Chapter 3: Control Flow & Functions

## 3.1 Conditional Statements (if, else, switch)

Conditional statements allow your program to make decisions and execute different code blocks based on certain conditions.

### if Statement

The if statement executes a block of code if a specified condition evaluates to true.

if (condition) {  
 // Code executed if condition is true  
}

Example:

#include <iostream>  
using namespace std;  
  
int main() {  
 int x = 10;  
   
 if (x > 5) {  
 cout << "x is greater than 5" << endl;  
 }  
   
 return 0;  
}

### if-else Statement

The if-else statement provides an alternative execution path when the condition is false.

if (condition) {  
 // Code executed if condition is true  
} else {  
 // Code executed if condition is false  
}

Example:

#include <iostream>  
using namespace std;  
  
int main() {  
 int score = 75;  
   
 if (score >= 60) {  
 cout << "You passed!" << endl;  
 } else {  
 cout << "You failed." << endl;  
 }  
   
 return 0;  
}

### if-else if-else Statement

For multiple conditions, you can chain together if and else if statements.

if (condition1) {  
 // Code executed if condition1 is true  
} else if (condition2) {  
 // Code executed if condition1 is false and condition2 is true  
} else {  
 // Code executed if both condition1 and condition2 are false  
}

Example:

#include <iostream>  
using namespace std;  
  
int main() {  
 int score = 85;  
   
 if (score >= 90) {  
 cout << "Grade A" << endl;  
 } else if (score >= 80) {  
 cout << "Grade B" << endl;  
 } else if (score >= 70) {  
 cout << "Grade C" << endl;  
 } else if (score >= 60) {  
 cout << "Grade D" << endl;  
 } else {  
 cout << "Grade F" << endl;  
 }  
   
 return 0;  
}

### Nested if Statements

You can place an if statement inside another if or else block.

if (condition1) {  
 // Code executed if condition1 is true  
 if (condition2) {  
 // Code executed if both condition1 and condition2 are true  
 }  
}

Example:

#include <iostream>  
using namespace std;  
  
int main() {  
 int age = 25;  
 bool hasLicense = true;  
   
 if (age >= 18) {  
 cout << "You are an adult." << endl;  
   
 if (hasLicense) {  
 cout << "You can drive." << endl;  
 } else {  
 cout << "You cannot drive without a license." << endl;  
 }  
 } else {  
 cout << "You are a minor." << endl;  
 }  
   
 return 0;  
}

### switch Statement

The switch statement tests a variable against multiple values and executes the corresponding code block.

switch (expression) {  
 case constant1:  
 // Code executed if expression equals constant1  
 break;  
 case constant2:  
 // Code executed if expression equals constant2  
 break;  
 // Additional cases  
 default:  
 // Code executed if expression doesn't match any case  
}

Example:

#include <iostream>  
using namespace std;  
  
int main() {  
 int day = 3;  
   
 switch (day) {  
 case 1:  
 cout << "Monday" << endl;  
 break;  
 case 2:  
 cout << "Tuesday" << endl;  
 break;  
 case 3:  
 cout << "Wednesday" << endl;  
 break;  
 case 4:  
 cout << "Thursday" << endl;  
 break;  
 case 5:  
 cout << "Friday" << endl;  
 break;  
 case 6:  
 cout << "Saturday" << endl;  
 break;  
 case 7:  
 cout << "Sunday" << endl;  
 break;  
 default:  
 cout << "Invalid day number" << endl;  
 }  
   
 return 0;  
}

#### Key Points About switch

* The expression must evaluate to an integral or enumeration type (int, char, enum)
* Each case must be a constant expression (literals or const values)
* The break statement terminates the switch block
* Without break, execution “falls through” to the next case
* The default case is executed if no case matches (optional)

#### Fall-through Behavior

#include <iostream>  
using namespace std;  
  
int main() {  
 int month = 3;  
   
 switch (month) {  
 case 12:  
 case 1:  
 case 2:  
 cout << "Winter" << endl;  
 break;  
 case 3:  
 case 4:  
 case 5:  
 cout << "Spring" << endl;  
 break;  
 case 6:  
 case 7:  
 case 8:  
 cout << "Summer" << endl;  
 break;  
 case 9:  
 case 10:  
 case 11:  
 cout << "Fall" << endl;  
 break;  
 default:  
 cout << "Invalid month" << endl;  
 }  
   
 return 0;  
}

### Best Practices for Conditional Statements

1. **Be Clear and Concise**
   * Keep conditions simple and readable
   * Use parentheses to clarify complex conditions
2. **Consider the Order of Conditions**
   * Place the most common conditions first for efficiency
   * Ensure mutually exclusive conditions are properly ordered
3. **Avoid Common Pitfalls**
   * Don’t use assignment (=) when you mean equality (==)
   * Be cautious with boolean expressions (use if (isReady) instead of if (isReady == true))
4. **Choose the Right Tool**
   * Use if/else if/else for complex conditions with different types
   * Use switch for comparing a single value against multiple constants
5. **Guard Against Edge Cases**
   * Consider boundary conditions
   * Handle unexpected values with default cases

## 3.2 Loops (for, while, do-while)

Loops allow you to execute a block of code repeatedly. There are three main types of loops in C++.

### for Loop

The for loop executes a block of code for a specified number of iterations.

for (initialization; condition; update) {  
 // Code to be repeated  
}

* **Initialization**: Executed once before the loop begins
* **Condition**: Checked before each iteration; loop continues while true
* **Update**: Executed after each iteration

Example:

#include <iostream>  
using namespace std;  
  
int main() {  
 // Print numbers from 1 to 5  
 for (int i = 1; i <= 5; i++) {  
 cout << i << " ";  
 }  
 cout << endl; // Output: 1 2 3 4 5  
   
 return 0;  
}

#### Variations of for Loop

1. **Multiple Initialization/Update Expressions**:

for (int i = 0, j = 10; i < j; i++, j--) {  
 cout << "i = " << i << ", j = " << j << endl;  
}

1. **Omitting Parts**:

int i = 0;  
for (; i < 5; i++) { // Initialization omitted  
 cout << i << " ";  
}  
  
for (int j = 0; j < 5;) { // Update omitted  
 cout << j << " ";  
 j++;  
}  
  
int k = 0;  
for (;;) { // All parts omitted (infinite loop)  
 cout << k << " ";  
 k++;  
 if (k >= 5) break;  
}

1. **Range-based for Loop (C++11)**:

#include <iostream>  
#include <vector>  
using namespace std;  
  
int main() {  
 vector<int> numbers = {1, 2, 3, 4, 5};  
   
 // Iterate through each element in numbers  
 for (int num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl; // Output: 1 2 3 4 5  
   
 // Using auto for type deduction  
 for (auto num : numbers) {  
 cout << num \* 2 << " ";  
 }  
 cout << endl; // Output: 2 4 6 8 10  
   
 // Using reference to modify elements  
 for (auto& num : numbers) {  
 num \*= 3;  
 }  
   
 // Print modified elements  
 for (int num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl; // Output: 3 6 9 12 15  
   
 return 0;  
}

### while Loop

The while loop executes a block of code as long as a specified condition is true.

while (condition) {  
 // Code to be repeated  
}

Example:

#include <iostream>  
using namespace std;  
  
int main() {  
 int count = 1;  
   
 while (count <= 5) {  
 cout << count << " ";  
 count++;  
 }  
 cout << endl; // Output: 1 2 3 4 5  
   
 return 0;  
}

#### Use Cases for while Loop

1. **When the number of iterations is unknown beforehand**:

#include <iostream>  
using namespace std;  
  
int main() {  
 int number = 100;  
   
 // Count how many divisions by 2 until number < 1  
 int steps = 0;  
 while (number >= 1) {  
 number /= 2;  
 steps++;  
 }  
   
 cout << "Steps needed: " << steps << endl; // Output: Steps needed: 7  
   
 return 0;  
}

1. **Input validation**:

#include <iostream>  
using namespace std;  
  
int main() {  
 int input;  
   
 cout << "Enter a positive number: ";  
 cin >> input;  
   
 while (input <= 0) {  
 cout << "Invalid input. Enter a positive number: ";  
 cin >> input;  
 }  
   
 cout << "You entered: " << input << endl;  
   
 return 0;  
}

### do-while Loop

The do-while loop executes a block of code once, then repeats as long as a specified condition is true.

do {  
 // Code to be repeated  
} while (condition);

Example:

#include <iostream>  
using namespace std;  
  
int main() {  
 int count = 1;  
   
 do {  
 cout << count << " ";  
 count++;  
 } while (count <= 5);  
 cout << endl; // Output: 1 2 3 4 5  
   
 return 0;  
}

#### do-while vs while

The key difference is that a do-while loop always executes the code block at least once, even if the condition is initially false:

#include <iostream>  
using namespace std;  
  
int main() {  
 int x = 10;  
   
 // while loop - condition is false, block never executes  
 while (x < 10) {  
 cout << "This will not be printed (while)" << endl;  
 }  
   
 // do-while loop - block executes once even though condition is false  
 do {  
 cout << "This will be printed once (do-while)" << endl;  
 } while (x < 10);  
   
 return 0;  
}

#### Common Use Case: Menus

#include <iostream>  
using namespace std;  
  
int main() {  
 int choice;  
   
 do {  
 cout << "\nMenu:" << endl;  
 cout << "1. Option One" << endl;  
 cout << "2. Option Two" << endl;  
 cout << "3. Option Three" << endl;  
 cout << "4. Exit" << endl;  
 cout << "Enter your choice: ";  
 cin >> choice;  
   
 switch (choice) {  
 case 1:  
 cout << "You selected Option One" << endl;  
 break;  
 case 2:  
 cout << "You selected Option Two" << endl;  
 break;  
 case 3:  
 cout << "You selected Option Three" << endl;  
 break;  
 case 4:  
 cout << "Exiting..." << endl;  
 break;  
 default:  
 cout << "Invalid choice! Try again." << endl;  
 }  
 } while (choice != 4);  
   
 return 0;  
}

### Nested Loops

You can place one loop inside another loop.

#include <iostream>  
using namespace std;  
  
int main() {  
 // Print a 3x3 pattern  
 for (int i = 1; i <= 3; i++) {  
 for (int j = 1; j <= 3; j++) {  
 cout << i << "," << j << " ";  
 }  
 cout << endl;  
 }  
   
 return 0;  
}

Output:

1,1 1,2 1,3   
2,1 2,2 2,3   
3,1 3,2 3,3

#### Common Applications of Nested Loops

1. **Matrix Operations**:

#include <iostream>  
using namespace std;  
  
int main() {  
 int matrix[3][3] = {  
 {1, 2, 3},  
 {4, 5, 6},  
 {7, 8, 9}  
 };  
   
 // Print matrix  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 cout << matrix[i][j] << " ";  
 }  
 cout << endl;  
 }  
   
 return 0;  
}

1. **Pattern Printing**:

#include <iostream>  
using namespace std;  
  
int main() {  
 // Print triangle pattern  
 for (int i = 1; i <= 5; i++) {  
 for (int j = 1; j <= i; j++) {  
 cout << "\* ";  
 }  
 cout << endl;  
 }  
   
 return 0;  
}

Output:

\*   
\* \*   
\* \* \*   
\* \* \* \*   
\* \* \* \* \*

### Loop Control & Efficiency

1. **Loop Variables**
   * Use meaningful variable names
   * Prefer smaller data types when appropriate (e.g., size\_t for array indices)
2. **Loop Invariant Code Motion**
   * Move calculations outside the loop when possible

// Less efficient  
for (int i = 0; i < n; i++) {  
 result += data[i] \* (10 \* factor / 2);  
}  
  
// More efficient  
double temp = 10 \* factor / 2;  
for (int i = 0; i < n; i++) {  
 result += data[i] \* temp;  
}

1. **Loop Unrolling**
   * Manually repeating loop body to reduce overhead of loop control
   * Modern compilers often do this automatically

// Standard loop  
for (int i = 0; i < 1000; i++) {  
 array[i] = i \* 2;  
}  
  
// Manually unrolled loop (4x)  
for (int i = 0; i < 1000; i += 4) {  
 array[i] = i \* 2;  
 array[i+1] = (i+1) \* 2;  
 array[i+2] = (i+2) \* 2;  
 array[i+3] = (i+3) \* 2;  
}

1. **Compiler Optimizations**
   * Modern compilers may optimize away loops that don’t have side effects
   * Be cautious with micro-optimization - focus on algorithm quality first

## 3.3 Break, Continue, Goto

C++ provides control statements to alter the normal flow of loops and conditional statements.

### break Statement

The break statement terminates the innermost enclosing loop or switch statement.

#### In Loops

#include <iostream>  
using namespace std;  
  
int main() {  
 // Find the first number divisible by 7 between 1 and 100  
 for (int i = 1; i <= 100; i++) {  
 if (i % 7 == 0) {  
 cout << "First number divisible by 7: " << i << endl;  
 break; // Exit the loop once found  
 }  
 }  
   
 return 0;  
}

#### In switch Statements

#include <iostream>  
using namespace std;  
  
int main() {  
 int choice = 2;  
   
 switch (choice) {  
 case 1:  
 cout << "Option 1 selected" << endl;  
 break; // Exit switch statement  
 case 2:  
 cout << "Option 2 selected" << endl;  
 break; // Exit switch statement  
 default:  
 cout << "Invalid option" << endl;  
 }  
   
 return 0;  
}

#### In Nested Loops

break only terminates the innermost loop:

#include <iostream>  
using namespace std;  
  
int main() {  
 for (int i = 1; i <= 3; i++) {  
 cout << "i = " << i << endl;  
   
 for (int j = 1; j <= 5; j++) {  
 if (j == 3) {  
 break; // Terminates only the inner loop  
 }  
 cout << " j = " << j << endl;  
 }  
 }  
   
 return 0;  
}

Output:

i = 1  
 j = 1  
 j = 2  
i = 2  
 j = 1  
 j = 2  
i = 3  
 j = 1  
 j = 2

### continue Statement

The continue statement skips the rest of the current iteration and proceeds to the next iteration of the loop.

#include <iostream>  
using namespace std;  
  
int main() {  
 // Print only odd numbers from 1 to 10  
 for (int i = 1; i <= 10; i++) {  
 if (i % 2 == 0) {  
 continue; // Skip even numbers  
 }  
 cout << i << " ";  
 }  
 cout << endl; // Output: 1 3 5 7 9  
   
 return 0;  
}

#### With Nested Loops

continue only affects the innermost loop:

#include <iostream>  
using namespace std;  
  
int main() {  
 for (int i = 1; i <= 3; i++) {  
 cout << "i = " << i << endl;  
   
 for (int j = 1; j <= 5; j++) {  
 if (j % 2 == 0) {  
 continue; // Skip even j values  
 }  
 cout << " j = " << j << endl;  
 }  
 }  
   
 return 0;  
}

Output:

i = 1  
 j = 1  
 j = 3  
 j = 5  
i = 2  
 j = 1  
 j = 3  
 j = 5  
i = 3  
 j = 1  
 j = 3  
 j = 5

### goto Statement

The goto statement transfers control to a labeled statement within the same function.

#include <iostream>  
using namespace std;  
  
int main() {  
 int i = 1;  
   
start: // Label  
 if (i <= 5) {  
 cout << i << " ";  
 i++;  
 goto start; // Jump to the labeled statement  
 }  
 cout << endl; // Output: 1 2 3 4 5  
   
 return 0;  
}

#### Avoiding goto in Modern C++

While goto exists in the language, its use is generally discouraged in modern C++ for several reasons:

1. **Reduces code readability** - Makes control flow hard to follow
2. **Makes debugging difficult** - Leads to “spaghetti code”
3. **Complicates maintenance** - Harder to understand and modify
4. **Better alternatives exist** - Loop statements, functions, exceptions

#### Limited Valid Use Cases

1. **Breaking out of nested loops**

#include <iostream>  
using namespace std;  
  
int main() {  
 bool found = false;  
 int target = 42;  
 int matrix[3][3] = {{1, 5, 9}, {10, 42, 99}, {7, 13, 21}};  
   
 // Search for target in matrix  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 if (matrix[i][j] == target) {  
 cout << "Found at position (" << i << "," << j << ")" << endl;  
 goto end\_search; // Break out of both loops  
 }  
 }  
 }  
 cout << "Target not found" << endl;  
   
end\_search:  
 // Continue with program  
   
 return 0;  
}

1. **Error handling in resource cleanup** (though exceptions or RAII are preferred in modern C++)

### Best Practices for Control Flow Statements

1. **Use break and continue judiciously**
   * Break when you need to exit a loop early
   * Continue when you need to skip an iteration
   * Use them to simplify logic and avoid deeply nested conditions
2. **Avoid goto in modern code**
   * Use structured programming constructs instead
   * If you must use goto, limit it to very specific scenarios
   * Comment thoroughly to explain your reasoning
3. **Consider alternatives to complex control flow**
   * Extract complex logic into separate functions
   * Use early returns for guard clauses
   * Use the algorithm library for common operations
4. **Be consistent with your approach**
   * Team conventions often dictate which control flow patterns are preferred
   * Document unusual control patterns

## 3.4 Functions: Declaration, Definition, Calling

Functions are reusable blocks of code that perform a specific task. They help organize code, reduce redundancy, and improve maintainability.

### Function Declaration (Prototype)

A function declaration tells the compiler about a function’s name, return type, and parameters.

return\_type function\_name(parameter\_list);

Example:

int add(int a, int b); // Function prototype  
double calculateArea(double radius);  
void displayMessage();

### Function Definition

A function definition includes the function’s implementation.

return\_type function\_name(parameter\_list) {  
 // Function body  
 return value; // If return\_type is not void  
}

Example:

int add(int a, int b) {  
 return a + b;  
}  
  
double calculateArea(double radius) {  
 const double PI = 3.14159;  
 return PI \* radius \* radius;  
}  
  
void displayMessage() {  
 cout << "Hello, world!" << endl;  
 // No return statement needed for void functions  
}

### Function Calling

To use a function, you call it by name and provide any required arguments.

#include <iostream>  
using namespace std;  
  
// Function declaration  
int add(int a, int b);  
  
int main() {  
 // Function calls  
 int sum = add(5, 3);  
 cout << "5 + 3 = " << sum << endl;  
   
 cout << "7 + 2 = " << add(7, 2) << endl; // Call and use result directly  
   
 return 0;  
}  
  
// Function definition  
int add(int a, int b) {  
 return a + b;  
}

### Parts of a Function

1. **Return Type**: The data type of the value returned by the function
2. **Function Name**: Identifier used to call the function
3. **Parameters**: Variables that receive values passed when the function is called
4. **Function Body**: Code that executes when the function is called
5. **Return Statement**: Returns a value and exits the function

### Void Functions

Functions that do not return a value have the void return type.

#include <iostream>  
using namespace std;  
  
void printStars(int count) {  
 for (int i = 0; i < count; i++) {  
 cout << "\*";  
 }  
 cout << endl;  
}  
  
int main() {  
 printStars(5); // Outputs: \*\*\*\*\*  
 printStars(10); // Outputs: \*\*\*\*\*\*\*\*\*\*  
   
 return 0;  
}

### Return Statement

The return statement ends a function’s execution and optionally returns a value.

#include <iostream>  
using namespace std;  
  
bool isEven(int number) {  
 if (number % 2 == 0) {  
 return true;  
 } else {  
 return false;  
 }  
}  
  
// More concise version  
bool isEvenConcise(int number) {  
 return number % 2 == 0;  
}  
  
int main() {  
 cout << "Is 4 even? " << (isEven(4) ? "Yes" : "No") << endl;  
 cout << "Is 7 even? " << (isEven(7) ? "Yes" : "No") << endl;  
   
 return 0;  
}

### Early Return

You can use multiple return statements, which is useful for handling special cases or implementing guard clauses.

#include <iostream>  
#include <cmath>  
using namespace std;  
  
double squareRoot(double x) {  
 if (x < 0) {  
 cout << "Error: Cannot calculate square root of negative number" << endl;  
 return -1; // Error indicator  
 }  
   
 return sqrt(x);  
}  
  
int main() {  
 cout << "Square root of 16: " << squareRoot(16) << endl;  
 cout << "Square root of -4: " << squareRoot(-4) << endl;  
   
 return 0;  
}

### Function Scope

Variables declared inside a function are local to that function.

#include <iostream>  
using namespace std;  
  
void function1() {  
 int x = 10;  
 cout << "In function1, x = " << x << endl;  
}  
  
void function2() {  
 int x = 20; // Different variable, same name  
 cout << "In function2, x = " << x << endl;  
}  
  
int main() {  
 function1(); // Outputs: In function1, x = 10  
 function2(); // Outputs: In function2, x = 20  
   
 // cout << x << endl; // Error: x is not defined in this scope  
   
 return 0;  
}

### Forward Declaration

Forward declaration allows you to use a function before its definition appears in the code.

#include <iostream>  
using namespace std;  
  
// Forward declarations  
int multiply(int a, int b);  
void displayResult(int result);  
  
int main() {  
 int product = multiply(4, 5);  
 displayResult(product);  
   
 return 0;  
}  
  
// Function definitions  
int multiply(int a, int b) {  
 return a \* b;  
}  
  
void displayResult(int result) {  
 cout << "The result is: " << result << endl;  
}

### Function Signatures

A function signature includes the function name and parameter types (not the return type).

#include <iostream>  
using namespace std;  
  
// These functions have different signatures  
void display(int x);  
void display(double x);  
void display(int x, int y);  
  
int main() {  
 display(5); // Calls display(int)  
 display(3.14); // Calls display(double)  
 display(10, 20); // Calls display(int, int)  
   
 return 0;  
}  
  
void display(int x) {  
 cout << "Integer: " << x << endl;  
}  
  
void display(double x) {  
 cout << "Double: " << x << endl;  
}  
  
void display(int x, int y) {  
 cout << "Two integers: " << x << " and " << y << endl;  
}

### Best Practices for Functions

1. **Single Responsibility Principle**
   * Each function should do one thing and do it well
   * Break complex tasks into smaller functions
2. **Meaningful Names**
   * Use descriptive function names (verbs for actions)
   * Parameter names should clearly indicate purpose
3. **Keep Functions Short**
   * Aim for less than 20-30 lines per function
   * Longer functions are harder to test and maintain
4. **Consistent Parameter Order**
   * Keep similar parameters in the same order across functions
   * Consider using named parameters for complex interfaces
5. **Documentation**
   * Comment purpose, parameters, return values, and exceptions
   * Include examples for complex functions
6. **Error Handling**
   * Define how errors are reported (return codes, exceptions)
   * Validate input parameters

## 3.5 Function Parameters: Call by Value & Reference

Function parameters can be passed in different ways, which affects how the function can interact with the arguments.

### Call by Value

In call by value, a copy of the argument is passed to the function. Changes to the parameter inside the function do not affect the original argument.

#include <iostream>  
using namespace std;  
  
void incrementValue(int x) {  
 x++; // Modifies the local copy, not the original  
 cout << "Inside function: x = " << x << endl;  
}  
  
int main() {  
 int num = 10;  
   
 cout << "Before function call: num = " << num << endl;  
 incrementValue(num);  
 cout << "After function call: num = " << num << endl;  
   
 return 0;  
}

Output:

Before function call: num = 10  
Inside function: x = 11  
After function call: num = 10

#### Advantages of Call by Value

* Original value is protected from changes
* Prevents unintended side effects
* Makes function behavior more predictable

#### Disadvantages of Call by Value

* Copying large objects can be inefficient
* Cannot modify the original argument

### Call by Reference

In call by reference, a reference to the original argument is passed to the function. Changes to the parameter inside the function affect the original argument.

#include <iostream>  
using namespace std;  
  
void incrementReference(int& x) {  
 x++; // Modifies the original value  
 cout << "Inside function: x = " << x << endl;  
}  
  
int main() {  
 int num = 10;  
   
 cout << "Before function call: num = " << num << endl;  
 incrementReference(num);  
 cout << "After function call: num = " << num << endl;  
   
 return 0;  
}

Output:

Before function call: num = 10  
Inside function: x = 11  
After function call: num = 11

#### Advantages of Call by Reference

* No copy is made, which is efficient for large objects
* Allows functions to modify the original argument
* Can return multiple values via reference parameters

#### Disadvantages of Call by Reference

* Original value can be changed (may be unintended)
* The relationship between argument and parameter is less obvious

### Const References

To get the efficiency of call by reference without allowing modifications, use const references:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Efficient (no copying) but prevents modification  
void displayDetails(const string& name, const int& age) {  
 cout << "Name: " << name << ", Age: " << age << endl;  
 // name = "John"; // Error: cannot modify a const reference  
}  
  
int main() {  
 string person = "Alice";  
 int years = 30;  
   
 displayDetails(person, years);  
   
 return 0;  
}

### Call by Pointer

Another way to achieve call by reference is by using pointers:

#include <iostream>  
using namespace std;  
  
void incrementPointer(int\* x) {  
 (\*x)++; // Dereference and increment  
 cout << "Inside function: \*x = " << \*x << endl;  
}  
  
int main() {  
 int num = 10;  
   
 cout << "Before function call: num = " << num << endl;  
 incrementPointer(&num); // Pass the address of num  
 cout << "After function call: num = " << num << endl;  
   
 return 0;  
}

Output:

Before function call: num = 10  
Inside function: \*x = 11  
After function call: num = 11

#### References vs. Pointers

1. **Syntax**
   * References: void func(int& x)
   * Pointers: void func(int\* x)
2. **Calling Convention**
   * References: func(num)
   * Pointers: func(&num)
3. **Nullability**
   * References: Cannot be null, must be initialized
   * Pointers: Can be null, must be checked
4. **Reassignment**
   * References: Cannot be reassigned to reference another variable
   * Pointers: Can be redirected to point to different variables

### When to Use Each Method

1. **Call by Value**:
   * For small, primitive types (int, char, bool)
   * When you don’t need to modify the original value
   * When you want to ensure function doesn’t modify arguments
2. **Call by Reference**:
   * For large objects to avoid copying
   * When you need to modify the original value
   * When returning multiple results
3. **Call by Const Reference**:
   * For large objects when you only need to read them
   * Most string and container parameters
4. **Call by Pointer**:
   * When parameter might be null
   * When you need to reassign the parameter
   * When working with C-style APIs

### Multiple Return Values via References

#include <iostream>  
using namespace std;  
  
void calculateStats(const int arr[], int size, double& average, int& min, int& max) {  
 if (size <= 0) return;  
   
 int sum = 0;  
 min = arr[0];  
 max = arr[0];  
   
 for (int i = 0; i < size; i++) {  
 sum += arr[i];  
 if (arr[i] < min) min = arr[i];  
 if (arr[i] > max) max = arr[i];  
 }  
   
 average = static\_cast<double>(sum) / size;  
}  
  
int main() {  
 int data[] = {5, 3, 8, 1, 9, 4};  
 int size = sizeof(data) / sizeof(data[0]);  
   
 double avg;  
 int minimum, maximum;  
   
 calculateStats(data, size, avg, minimum, maximum);  
   
 cout << "Average: " << avg << endl;  
 cout << "Minimum: " << minimum << endl;  
 cout << "Maximum: " << maximum << endl;  
   
 return 0;  
}

### Using References for Efficiency

When working with large objects, passing by reference is much more efficient:

#include <iostream>  
#include <vector>  
#include <string>  
using namespace std;  
  
// Inefficient: copies the entire vector  
void processByValue(vector<string> data) {  
 // Function body  
}  
  
// Efficient: uses a reference  
void processByReference(const vector<string>& data) {  
 // Function body  
}  
  
int main() {  
 vector<string> largeData(10000, "Large String");  
   
 processByValue(largeData); // Copies 10,000 strings  
 processByReference(largeData); // No copying  
   
 return 0;  
}

## 3.6 Recursion in C++

Recursion is a technique where a function calls itself to solve a problem by breaking it down into smaller instances of the same problem.

### Basic Recursion Concept

A recursive function has two main components: 1. **Base Case**: The condition(s) under which the function stops calling itself 2. **Recursive Case**: The condition(s) under which the function calls itself

#include <iostream>  
using namespace std;  
  
void countdown(int n) {  
 // Base case  
 if (n <= 0) {  
 cout << "Blastoff!" << endl;  
 return;  
 }  
   
 // Current step  
 cout << n << endl;  
   
 // Recursive case  
 countdown(n - 1);  
}  
  
int main() {  
 countdown(5);  
 return 0;  
}

Output:

5  
4  
3  
2  
1  
Blastoff!

### Factorial Calculation

A classic example of recursion is calculating the factorial of a number.

#include <iostream>  
using namespace std;  
  
// Recursive factorial function  
unsigned long long factorial(unsigned int n) {  
 // Base case  
 if (n == 0 || n == 1) {  
 return 1;  
 }  
   
 // Recursive case  
 return n \* factorial(n - 1);  
}  
  
int main() {  
 cout << "5! = " << factorial(5) << endl; // 120  
 cout << "10! = " << factorial(10) << endl; // 3628800  
   
 return 0;  
}

### Fibonacci Sequence

Another common example is generating Fibonacci numbers.

#include <iostream>  
using namespace std;  
  
int fibonacci(int n) {  
 // Base cases  
 if (n <= 0) return 0;  
 if (n == 1) return 1;  
   
 // Recursive case  
 return fibonacci(n - 1) + fibonacci(n - 2);  
}  
  
int main() {  
 cout << "Fibonacci sequence (first 10 numbers):" << endl;  
 for (int i = 0; i < 10; i++) {  
 cout << fibonacci(i) << " ";  
 }  
 cout << endl; // Output: 0 1 1 2 3 5 8 13 21 34  
   
 return 0;  
}

### Understanding the Call Stack

Each recursive call adds a new frame to the call stack, which stores local variables and the return address.

Execution of factorial(5):  
  
Call Stack (bottom to top):  
main()  
factorial(5)  
 return 5 \* factorial(4)  
 factorial(4)  
 return 4 \* factorial(3)  
 factorial(3)  
 return 3 \* factorial(2)  
 factorial(2)  
 return 2 \* factorial(1)  
 factorial(1)  
 return 1  
 returns: 2 \* 1 = 2  
 returns: 3 \* 2 = 6  
 returns: 4 \* 6 = 24  
 returns: 5 \* 24 = 120  
final result: 120

### Recursion vs. Iteration

Both recursion and iteration can solve the same problems, but they have different characteristics:

1. **Memory Usage**:
   * Recursion: Uses more memory due to call stack overhead
   * Iteration: Typically uses less memory
2. **Code Clarity**:
   * Recursion: Often more elegant and clear for certain problems
   * Iteration: Can be more straightforward for simple problems
3. **Performance**:
   * Recursion: Often slower due to function call overhead
   * Iteration: Usually faster

Example of factorial using iteration:

#include <iostream>  
using namespace std;  
  
unsigned long long factorialIterative(unsigned int n) {  
 unsigned long long result = 1;  
 for (unsigned int i = 2; i <= n; i++) {  
 result \*= i;  
 }  
 return result;  
}  
  
int main() {  
 cout << "5! = " << factorialIterative(5) << endl; // 120  
 return 0;  
}

### Tail Recursion

Tail recursion is a special form of recursion where the recursive call is the last operation in the function. Modern compilers can optimize tail-recursive functions to use constant stack space.

#include <iostream>  
using namespace std;  
  
// Non-tail-recursive factorial  
unsigned long long factorial(unsigned int n) {  
 if (n == 0 || n == 1) return 1;  
 return n \* factorial(n - 1); // Must multiply after the recursive call returns  
}  
  
// Tail-recursive factorial  
unsigned long long factorialTail(unsigned int n, unsigned long long accumulator = 1) {  
 if (n == 0 || n == 1) return accumulator;  
 return factorialTail(n - 1, n \* accumulator); // No pending operations after the recursive call  
}  
  
int main() {  
 cout << "5! = " << factorial(5) << endl;  
 cout << "5! (tail recursive) = " << factorialTail(5) << endl;  
   
 return 0;  
}

### Indirect Recursion

Indirect recursion occurs when function A calls function B, which in turn calls function A.

#include <iostream>  
using namespace std;  
  
void functionB(int n); // Forward declaration  
  
void functionA(int n) {  
 if (n > 0) {  
 cout << "A: " << n << endl;  
 functionB(n - 1);  
 }  
}  
  
void functionB(int n) {  
 if (n > 0) {  
 cout << "B: " << n << endl;  
 functionA(n - 1);  
 }  
}  
  
int main() {  
 functionA(3);  
 return 0;  
}

Output:

A: 3  
B: 2  
A: 1  
B: 0

### Common Applications of Recursion

1. **Tree and Graph Traversal**:

void traverseTree(TreeNode\* node) {  
 if (node == nullptr) return;  
   
 // Process current node  
 cout << node->value << " ";  
   
 // Recursively process children  
 traverseTree(node->left);  
 traverseTree(node->right);  
}

1. **Divide and Conquer Algorithms**:

int binarySearch(int arr[], int left, int right, int target) {  
 if (right >= left) {  
 int mid = left + (right - left) / 2;  
   
 if (arr[mid] == target) return mid;  
   
 if (arr[mid] > target)  
 return binarySearch(arr, left, mid - 1, target);  
   
 return binarySearch(arr, mid + 1, right, target);  
 }  
   
 return -1; // Not found  
}

1. **Backtracking**:

bool solveNQueens(int board[N][N], int col) {  
 // Base case: if all queens are placed  
 if (col >= N) return true;  
   
 // Try placing queen in each row of the current column  
 for (int row = 0; row < N; row++) {  
 if (isSafe(board, row, col)) {  
 // Place this queen in board[row][col]  
 board[row][col] = 1;  
   
 // Recur to place rest of the queens  
 if (solveNQueens(board, col + 1))  
 return true;  
   
 // If placing queen doesn't lead to a solution, backtrack  
 board[row][col] = 0;  
 }  
 }  
   
 return false; // No solution found  
}

### Pitfalls of Recursion

1. **Stack Overflow**: If recursion is too deep, it can exhaust the available stack space.

// This will cause stack overflow for large values of n  
int badRecursion(int n) {  
 return badRecursion(n-1) + badRecursion(n-2); // Missing base case!  
}

1. **Redundant Calculations**: Naive recursive implementations can repeat calculations.

// Inefficient fibonacci - many calculations are repeated  
int fib(int n) {  
 if (n <= 1) return n;  
 return fib(n-1) + fib(n-2); // fib(n-2) is calculated multiple times  
}

1. **Solutions**:
   * **Memoization**: Store already computed results
   * **Dynamic Programming**: Build solutions from bottom-up
   * **Iteration**: Convert to iterative form when appropriate

### Memoization Example

#include <iostream>  
#include <vector>  
using namespace std;  
  
int fibMemoized(int n, vector<int>& memo) {  
 if (n <= 1) return n;  
   
 // Return cached result if available  
 if (memo[n] != -1) return memo[n];  
   
 // Calculate and cache the result  
 memo[n] = fibMemoized(n - 1, memo) + fibMemoized(n - 2, memo);  
 return memo[n];  
}  
  
int fibonacci(int n) {  
 // Initialize memoization array with -1  
 vector<int> memo(n + 1, -1);  
 return fibMemoized(n, memo);  
}  
  
int main() {  
 cout << "Fibonacci(40): " << fibonacci(40) << endl; // Much faster than naive recursion  
 return 0;  
}

## 3.7 Inline Functions

Inline functions are a C++ enhancement to improve performance by avoiding function call overhead, especially for small functions.

### Function Call Overhead

When a function is called, several operations occur: 1. Parameter values are pushed onto the stack 2. Control jumps to the function location 3. Local variables are allocated 4. The function executes 5. Return value is placed where the caller can access it 6. Control returns to the caller 7. Stack is cleaned up

For small functions, this overhead can be significant relative to the function’s actual work.

### Inline Functions Concept

Inline functions suggest to the compiler that it should try to replace the function call with the function’s code at the call site.

#include <iostream>  
using namespace std;  
  
// Inline function declaration  
inline int square(int x) {  
 return x \* x;  
}  
  
int main() {  
 int result = square(5); // The compiler may replace this with: int result = 5 \* 5;  
   
 cout << "The square is: " << result << endl;  
   
 return 0;  
}

### Defining Inline Functions

Two common ways to define inline functions:

1. **Using the inline keyword**:

inline int max(int a, int b) {  
 return (a > b) ? a : b;  
}

1. **Defining the function within a class definition** (implicitly inline):

class Rectangle {  
public:  
 // Implicitly inline  
 int area() {  
 return width \* height;  
 }  
   
 int width;  
 int height;  
};

### When to Use Inline Functions

Inline functions are most beneficial when:

1. **The function is small** (a few lines of code)
2. **The function is called frequently**
3. **Function call overhead is significant relative to the function’s work**
4. **Performance is critical** in that section of code

Common examples: - Simple accessors and mutators (getters/setters) - Small utility functions - Simple mathematical operations

### Advantages of Inline Functions

1. **Reduced function call overhead**
2. **Potential for additional compiler optimizations** since the code is visible at the call site
3. **No stack operations** for parameters and return values
4. **Maintains the abstraction of functions** unlike macros

### Disadvantages of Inline Functions

1. **Code bloat**: If an inline function is large and called in many places, the compiled code size increases
2. **Limited debugging**: Breakpoints might behave unexpectedly
3. **Not suitable for functions that change frequently**: Changes require recompilation of all calling code
4. **The inline keyword is only a suggestion**: The compiler may ignore it

### Inline vs. Macros

Before inline functions, C-style macros served a similar purpose:

// Macro definition  
#define SQUARE(x) ((x) \* (x))  
  
// Inline function  
inline int square(int x) {  
 return x \* x;  
}

Advantages of inline functions over macros: 1. **Type safety**: Inline functions respect C++ type system 2. **No unexpected behavior**: Macros are simple text substitution which can lead to bugs 3. **Proper evaluation of arguments**: Each argument in a function is evaluated exactly once 4. **Debugging support**: Inline functions can be debugged; macros cannot

### Example of Macro Problems

#include <iostream>  
using namespace std;  
  
#define SQUARE(x) (x \* x)  
  
int main() {  
 int a = 5;  
 cout << "SQUARE(a+1) = " << SQUARE(a+1) << endl; // Expands to: (a+1 \* a+1) = 5+1\*5+1 = 11, not 36!  
   
 return 0;  
}

Corrected macro (but still not as good as inline function):

#define SQUARE(x) ((x) \* (x))

### Compiler Decisions on Inlining

Modern compilers make sophisticated decisions about inlining:

1. They may inline functions even without the inline keyword
2. They may refuse to inline functions marked as inline if they determine it’s not beneficial
3. They consider factors like:
   * Function size
   * How often it’s called
   * Call context
   * Optimization level

### Best Practices for Inline Functions

1. **Use inline for very small functions** (1-3 lines of code)
2. **Don’t inline complex functions** with loops or complex control structures
3. **Be aware that inline is a suggestion**, not a command
4. **Let the compiler decide** when unsure (modern compilers are good at this)
5. **Profile before optimizing** to identify if function call overhead is actually a bottleneck
6. **Consider templates for type-generic inline functions**

### Example of a Practical Inline Function

#include <iostream>  
using namespace std;  
  
class Vector3D {  
private:  
 double x, y, z;  
  
public:  
 Vector3D(double x = 0.0, double y = 0.0, double z = 0.0) : x(x), y(y), z(z) {}  
   
 // Inline accessors  
 inline double getX() const { return x; }  
 inline double getY() const { return y; }  
 inline double getZ() const { return z; }  
   
 // Inline mutators  
 inline void setX(double val) { x = val; }  
 inline void setY(double val) { y = val; }  
 inline void setZ(double val) { z = val; }  
   
 // Inline utility function  
 inline double dotProduct(const Vector3D& other) const {  
 return x \* other.x + y \* other.y + z \* other.z;  
 }  
   
 // Not a good candidate for inline due to complexity  
 Vector3D crossProduct(const Vector3D& other) const;  
};  
  
// Complex function defined outside the class  
Vector3D Vector3D::crossProduct(const Vector3D& other) const {  
 return Vector3D(  
 y \* other.z - z \* other.y,  
 z \* other.x - x \* other.z,  
 x \* other.y - y \* other.x  
 );  
}  
  
int main() {  
 Vector3D v1(1.0, 2.0, 3.0);  
 Vector3D v2(4.0, 5.0, 6.0);  
   
 // These calls may be inlined  
 cout << "Dot product: " << v1.dotProduct(v2) << endl;  
   
 // This call might not be inlined due to complexity  
 Vector3D v3 = v1.crossProduct(v2);  
   
 return 0;  
}

### Function Inlining in Headers vs. Source Files

1. **Inline functions in headers**:
   * Must be fully defined in the header file
   * Each translation unit that includes the header gets its own copy
   * No linking issues when defined in multiple files
2. **Regular functions in headers**:
   * If defined (not just declared) in headers, cause multiple definition errors
   * Should only be declared in headers, with definition in one source file

// In header file "utils.h"  
inline int add(int a, int b) {  
 return a + b; // Definition in header is fine for inline functions  
}  
  
int subtract(int a, int b); // Only declaration for regular functions  
  
// In source file "utils.cpp"  
#include "utils.h"  
  
int subtract(int a, int b) {  
 return a - b; // Definition goes in source file  
}

With these notes, the student will have comprehensive details on Chapter 3 topics including control flow statements, functions, function parameters, recursion, and inline functions. The content includes detailed explanations, code examples, best practices, and important concepts to help master these fundamental C++ concepts.  
  
# Chapter 4: Arrays and Strings  
  
## 4.1 Arrays: 1D, 2D, and Multi-dimensional  
  
Arrays are collections of elements of the same data type stored in contiguous memory locations. They provide a way to group related data under a single name, allowing for efficient data organization and access.  
  
### One-Dimensional Arrays  
  
A one-dimensional array is the simplest form of array, representing a linear collection of elements.  
  
#### Declaration and Initialization  
  
```cpp  
// Declaration syntax  
data\_type array\_name[size];  
  
// Examples  
int numbers[5]; // Declares an array of 5 integers (uninitialized)  
double prices[3] = {10.5, 20.75, 30.0}; // Declares and initializes  
char grades[] = {'A', 'B', 'C', 'D', 'F'}; // Size determined automatically

#### Accessing Array Elements

Array elements are accessed using zero-based indexing:

#include <iostream>  
using namespace std;  
  
int main() {  
 int scores[5] = {95, 88, 76, 92, 85};  
   
 // Accessing individual elements  
 cout << "First score: " << scores[0] << endl; // 95  
 cout << "Third score: " << scores[2] << endl; // 76  
   
 // Modifying an element  
 scores[1] = 90;  
 cout << "Updated second score: " << scores[1] << endl; // 90  
   
 return 0;  
}

#### Common Pitfalls with 1D Arrays

1. **Array Index Out of Bounds**

* int arr[5] = {1, 2, 3, 4, 5};  
  cout << arr[5]; // ERROR: Valid indices are 0-4
* This is undefined behavior and can cause crashes or unexpected results.

1. **Uninitialized Arrays**

* int values[5]; // Elements contain garbage values

1. **Array Size Must Be a Compile-Time Constant**

* int size = 10;  
  int arr[size]; // Error in standard C++ (though some compilers allow it)

### Two-Dimensional Arrays

Two-dimensional arrays can be visualized as tables with rows and columns.

#### Declaration and Initialization

// Declaration syntax  
data\_type array\_name[rows][columns];  
  
// Examples  
int matrix[3][4]; // 3 rows, 4 columns (uninitialized)  
  
// Initialization  
int grid[2][3] = {  
 {1, 2, 3}, // First row  
 {4, 5, 6} // Second row  
};  
  
// Alternative initialization syntax  
int grid2[2][3] = {1, 2, 3, 4, 5, 6}; // Filled row by row  
  
// Partial initialization  
int grid3[2][3] = {{1, 2}}; // Rest filled with zeros

#### Accessing 2D Array Elements

#include <iostream>  
using namespace std;  
  
int main() {  
 int matrix[3][3] = {  
 {1, 2, 3},  
 {4, 5, 6},  
 {7, 8, 9}  
 };  
   
 // Accessing individual elements  
 cout << "Element at (1,2): " << matrix[1][2] << endl; // 6  
   
 // Modifying an element  
 matrix[0][0] = 10;  
 cout << "Updated element at (0,0): " << matrix[0][0] << endl; // 10  
   
 // Iterating through a 2D array  
 cout << "All matrix elements:" << endl;  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 3; j++) {  
 cout << matrix[i][j] << " ";  
 }  
 cout << endl;  
 }  
   
 return 0;  
}

#### Memory Layout of 2D Arrays

In C++, 2D arrays are stored in row-major order, meaning all elements of the first row are stored first, then the second row, and so on.

For int arr[3][4], the memory layout is:  
arr[0][0], arr[0][1], arr[0][2], arr[0][3], arr[1][0], arr[1][1], ...

This is important to understand for performance reasons, especially when working with large arrays.

### Multi-dimensional Arrays

C++ supports arrays with more than two dimensions. The principles are the same as for 2D arrays, but with additional indexes.

// 3D array (cube)  
int cube[2][3][4]; // 2 layers, 3 rows, 4 columns  
  
// Initialization  
int cube[2][2][2] = {  
 { // First layer  
 {1, 2},  
 {3, 4}  
 },  
 { // Second layer  
 {5, 6},  
 {7, 8}  
 }  
};  
  
// Accessing elements  
cube[0][1][1] = 10; // First layer, second row, second column

#### Iterating Through Multi-dimensional Arrays

#include <iostream>  
using namespace std;  
  
int main() {  
 int cube[2][2][3] = {  
 {{1, 2, 3}, {4, 5, 6}},  
 {{7, 8, 9}, {10, 11, 12}}  
 };  
   
 // Iterating through a 3D array  
 for (int i = 0; i < 2; i++) {  
 cout << "Layer " << i << ":" << endl;  
 for (int j = 0; j < 2; j++) {  
 for (int k = 0; k < 3; k++) {  
 cout << cube[i][j][k] << " ";  
 }  
 cout << endl;  
 }  
 cout << endl;  
 }  
   
 return 0;  
}

### Arrays in Functions

When passing arrays to functions, C++ actually passes a pointer to the first element, not a copy of the array.

#include <iostream>  
using namespace std;  
  
// Function that takes a 1D array  
void printArray(int arr[], int size) {  
 for (int i = 0; i < size; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
   
 // Modifying the array affects the original  
 arr[0] = 100;  
}  
  
// Function that takes a 2D array  
// Column size must be specified  
void print2DArray(int arr[][3], int rows) {  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < 3; j++) {  
 cout << arr[i][j] << " ";  
 }  
 cout << endl;  
 }  
}  
  
int main() {  
 int numbers[] = {1, 2, 3, 4, 5};  
 int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};  
   
 printArray(numbers, 5);  
 cout << "After function call, numbers[0] = " << numbers[0] << endl; // 100  
   
 print2DArray(matrix, 2);  
   
 return 0;  
}

#### Alternative Ways to Pass Arrays

Using templates and references can make array handling safer and more flexible:

#include <iostream>  
using namespace std;  
  
// Using a template parameter for the array size  
template <size\_t N>  
void printArray(int (&arr)[N]) {  
 for (size\_t i = 0; i < N; i++) {  
 cout << arr[i] << " ";  
 }  
 cout << endl;  
}  
  
// For 2D arrays  
template <size\_t Rows, size\_t Cols>  
void print2DArray(int (&arr)[Rows][Cols]) {  
 for (size\_t i = 0; i < Rows; i++) {  
 for (size\_t j = 0; j < Cols; j++) {  
 cout << arr[i][j] << " ";  
 }  
 cout << endl;  
 }  
}  
  
int main() {  
 int numbers[] = {1, 2, 3, 4, 5};  
 int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};  
   
 printArray(numbers); // Size deduced automatically  
 print2DArray(matrix); // Dimensions deduced automatically  
   
 return 0;  
}

### Modern Array Alternatives

While traditional C-style arrays are important to understand, modern C++ provides safer alternatives:

1. **std::array** (fixed-size array, C++11)

#include <array>  
#include <iostream>  
using namespace std;  
  
int main() {  
 array<int, 5> numbers = {1, 2, 3, 4, 5};  
   
 cout << "Size: " << numbers.size() << endl;  
 cout << "Element at index 2: " << numbers[2] << endl;  
 cout << "Element at index 2 (with bounds checking): " << numbers.at(2) << endl;  
   
 return 0;  
}

1. **std::vector** (dynamic-size array)

#include <vector>  
#include <iostream>  
using namespace std;  
  
int main() {  
 vector<int> numbers = {1, 2, 3, 4, 5};  
   
 // Adding elements  
 numbers.push\_back(6);  
   
 // Accessing elements  
 cout << "Size: " << numbers.size() << endl;  
 cout << "Element at index 2: " << numbers[2] << endl;  
   
 return 0;  
}

## 4.2 Operations on Arrays (Insert, Delete, Search, Traverse)

Arrays support various operations, including traversal, insertion, deletion, and searching. However, unlike higher-level data structures, C-style arrays don’t provide built-in methods for these operations, so we need to implement them manually.

### Array Traversal

Traversal means visiting each element in the array.

#include <iostream>  
using namespace std;  
  
int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
   
 // Traversing using indexing  
 cout << "Using index-based loop:" << endl;  
 for (int i = 0; i < size; i++) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
   
 // Traversing using range-based for loop (C++11)  
 cout << "Using range-based for loop:" << endl;  
 for (int num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### Array Insertion

Adding elements to an array requires shifting existing elements to make space.

#include <iostream>  
using namespace std;  
  
// Function to insert an element at a specific position  
bool insertElement(int arr[], int& size, int maxSize, int element, int position) {  
 // Check if array is full or position is invalid  
 if (size >= maxSize || position < 0 || position > size) {  
 return false;  
 }  
   
 // Shift elements to the right  
 for (int i = size; i > position; i--) {  
 arr[i] = arr[i-1];  
 }  
   
 // Insert the element  
 arr[position] = element;  
 size++;  
   
 return true;  
}  
  
int main() {  
 const int maxSize = 10;  
 int numbers[maxSize] = {10, 20, 30, 40, 50};  
 int size = 5;  
   
 cout << "Before insertion: ";  
 for (int i = 0; i < size; i++) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
   
 // Insert 25 at position 2  
 if (insertElement(numbers, size, maxSize, 25, 2)) {  
 cout << "After insertion: ";  
 for (int i = 0; i < size; i++) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
 } else {  
 cout << "Insertion failed" << endl;  
 }  
   
 return 0;  
}

Output:

Before insertion: 10 20 30 40 50  
After insertion: 10 20 25 30 40 50

### Array Deletion

Removing elements requires shifting subsequent elements to fill the gap.

#include <iostream>  
using namespace std;  
  
// Function to delete an element at a specific position  
bool deleteElement(int arr[], int& size, int position) {  
 // Check if array is empty or position is invalid  
 if (size <= 0 || position < 0 || position >= size) {  
 return false;  
 }  
   
 // Shift elements to the left  
 for (int i = position; i < size - 1; i++) {  
 arr[i] = arr[i+1];  
 }  
   
 // Decrease size  
 size--;  
   
 return true;  
}  
  
int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
 int size = 5;  
   
 cout << "Before deletion: ";  
 for (int i = 0; i < size; i++) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
   
 // Delete element at position 2 (value 30)  
 if (deleteElement(numbers, size, 2)) {  
 cout << "After deletion: ";  
 for (int i = 0; i < size; i++) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
 } else {  
 cout << "Deletion failed" << endl;  
 }  
   
 return 0;  
}

Output:

Before deletion: 10 20 30 40 50  
After deletion: 10 20 40 50

### Linear Search

Linear search checks each element sequentially until the target is found or the end is reached.

#include <iostream>  
using namespace std;  
  
// Function to perform linear search  
int linearSearch(int arr[], int size, int target) {  
 for (int i = 0; i < size; i++) {  
 if (arr[i] == target) {  
 return i; // Return index if found  
 }  
 }  
 return -1; // Return -1 if not found  
}  
  
int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
 int size = 5;  
 int target = 30;  
   
 int result = linearSearch(numbers, size, target);  
   
 if (result != -1) {  
 cout << "Element " << target << " found at index " << result << endl;  
 } else {  
 cout << "Element " << target << " not found" << endl;  
 }  
   
 return 0;  
}

### Binary Search

Binary search is more efficient for sorted arrays. It repeatedly divides the search space in half.

#include <iostream>  
using namespace std;  
  
// Function to perform binary search on a sorted array  
int binarySearch(int arr[], int left, int right, int target) {  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
   
 // Check if target is at mid  
 if (arr[mid] == target) {  
 return mid;  
 }  
   
 // If target is greater, ignore left half  
 if (arr[mid] < target) {  
 left = mid + 1;  
 }  
 // If target is smaller, ignore right half  
 else {  
 right = mid - 1;  
 }  
 }  
   
 // Target not found  
 return -1;  
}  
  
int main() {  
 int sortedNumbers[] = {10, 20, 30, 40, 50, 60, 70};  
 int size = 7;  
 int target = 40;  
   
 int result = binarySearch(sortedNumbers, 0, size - 1, target);  
   
 if (result != -1) {  
 cout << "Element " << target << " found at index " << result << endl;  
 } else {  
 cout << "Element " << target << " not found" << endl;  
 }  
   
 return 0;  
}

### Sorting Arrays

Sorting is a fundamental operation. Here’s a simple implementation of bubble sort:

#include <iostream>  
using namespace std;  
  
// Function to perform bubble sort  
void bubbleSort(int 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]) {  
 // Swap elements  
 int temp = arr[j];  
 arr[j] = arr[j+1];  
 arr[j+1] = temp;  
 }  
 }  
 }  
}  
  
int main() {  
 int numbers[] = {64, 34, 25, 12, 22, 11, 90};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
   
 cout << "Before sorting: ";  
 for (int i = 0; i < size; i++) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
   
 bubbleSort(numbers, size);  
   
 cout << "After sorting: ";  
 for (int i = 0; i < size; i++) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

Output:

Before sorting: 64 34 25 12 22 11 90  
After sorting: 11 12 22 25 34 64 90

#### Using Standard Library Algorithms

The C++ standard library provides efficient implementations of many common array operations:

#include <iostream>  
#include <algorithm> // for sort(), find(), etc.  
using namespace std;  
  
int main() {  
 int numbers[] = {64, 34, 25, 12, 22, 11, 90};  
 int size = sizeof(numbers) / sizeof(numbers[0]);  
   
 // Sort  
 sort(numbers, numbers + size);  
   
 cout << "After sorting: ";  
 for (int i = 0; i < size; i++) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
   
 // Binary search (array must be sorted)  
 bool found = binary\_search(numbers, numbers + size, 25);  
 cout << "Is 25 in the array? " << (found ? "Yes" : "No") << endl;  
   
 // Find element  
 int\* position = find(numbers, numbers + size, 22);  
 if (position != numbers + size) {  
 cout << "Found 22 at index: " << (position - numbers) << endl;  
 }  
   
 // Count occurrences  
 int count = std::count(numbers, numbers + size, 11);  
 cout << "Number of occurrences of 11: " << count << endl;  
   
 return 0;  
}

## 4.3 Strings in C++

Strings represent sequences of characters. C++ provides two main ways to work with strings: 1. C-style strings (character arrays) 2. The std::string class from the standard library

### 4.3.1 Character Arrays vs std::string

#### C-style Strings

C-style strings are character arrays terminated by a null character '\0'.

#include <iostream>  
#include <cstring> // For C-string functions  
using namespace std;  
  
int main() {  
 // Declaration and initialization  
 char greeting1[] = "Hello"; // Compiler adds null terminator  
 char greeting2[] = {'H', 'e', 'l', 'l', 'o', '\0'}; // Explicit null terminator  
   
 // Size calculations  
 cout << "Size of greeting1: " << sizeof(greeting1) << endl; // 6 (includes '\0')  
 cout << "Length of greeting1: " << strlen(greeting1) << endl; // 5 (excludes '\0')  
   
 return 0;  
}

**Limitations of C-style strings**: - Fixed size determined at declaration - No bounds checking - Manual memory management required for dynamic strings - Limited built-in operations

#### std::string

The std::string class provides a safer, more flexible, and feature-rich alternative.

#include <iostream>  
#include <string>  
using namespace std;  
  
int main() {  
 // Declaration and initialization  
 string str1 = "Hello";  
 string str2("World");  
 string str3 = str1 + " " + str2; // String concatenation  
   
 // Size and capacity  
 cout << "str3: " << str3 << endl;  
 cout << "Length: " << str3.length() << endl;  
 cout << "Size: " << str3.size() << endl; // Same as length()  
 cout << "Capacity: " << str3.capacity() << endl; // Space allocated  
   
 return 0;  
}

**Advantages of std::string**: - Dynamically resizes as needed - Provides bounds checking with .at() method - Handles memory management automatically - Rich set of member functions for string manipulation

#### Comparison Between C-style Strings and std::string

| Feature | C-style Strings | std::string |
| --- | --- | --- |
| Memory Management | Manual | Automatic |
| Size | Fixed at compile time | Dynamic |
| Bounds Checking | None | Available with .at() |
| Concatenation | Manual (strcat) | Simple with + operator |
| Comparison | strcmp function | ==, !=, <, > operators |
| Standard Library Support | Limited (cstring) | Extensive |
| Performance | Sometimes faster for simple operations | Optimized for most use cases |

### 4.3.2 String Functions

#### C-style String Functions

The <cstring> header provides various functions for C-style strings:

#include <iostream>  
#include <cstring>  
using namespace std;  
  
int main() {  
 char str1[20] = "Hello";  
 char str2[20] = "World";  
 char result[40];  
   
 // String length  
 cout << "Length of str1: " << strlen(str1) << endl;  
   
 // String copy  
 strcpy(result, str1);  
 cout << "strcpy result: " << result << endl;  
   
 // String concatenation  
 strcat(result, " ");  
 strcat(result, str2);  
 cout << "strcat result: " << result << endl;  
   
 // String comparison  
 int comparison = strcmp(str1, str2);  
 cout << "strcmp result: " << comparison << endl;  
 // Negative: str1 < str2, 0: str1 == str2, Positive: str1 > str2  
   
 // Finding characters  
 char\* position = strchr(str1, 'l');  
 if (position) {  
 cout << "First 'l' found at position: " << (position - str1) << endl;  
 }  
   
 // Finding substrings  
 char\* subPosition = strstr(result, "World");  
 if (subPosition) {  
 cout << "Substring 'World' found at position: " << (subPosition - result) << endl;  
 }  
   
 return 0;  
}

#### std::string Member Functions

The std::string class provides numerous member functions for string manipulation:

#include <iostream>  
#include <string>  
using namespace std;  
  
int main() {  
 string str = "Hello, World!";  
   
 // Access operations  
 cout << "First character: " << str[0] << endl;  
 cout << "Safe access: " << str.at(1) << endl;  
 cout << "Substring: " << str.substr(7, 5) << endl; // "World"  
   
 // Modification  
 str.replace(7, 5, "C++");  
 cout << "After replace: " << str << endl; // "Hello, C++!"  
   
 str.insert(7, "Amazing ");  
 cout << "After insert: " << str << endl; // "Hello, Amazing C++!"  
   
 str.erase(7, 8); // Erase "Amazing "  
 cout << "After erase: " << str << endl; // "Hello, C++!"  
   
 // Finding  
 size\_t pos = str.find("C++");  
 if (pos != string::npos) {  
 cout << "Found 'C++' at position: " << pos << endl;  
 }  
   
 // Case conversion (C++17 or later with <algorithm>)  
 string lower = str;  
 transform(lower.begin(), lower.end(), lower.begin(), ::tolower);  
 cout << "Lowercase: " << lower << endl;  
   
 return 0;  
}

### 4.3.3 Input/Output of Strings

#### Reading and Writing C-style Strings

#include <iostream>  
#include <cstring>  
using namespace std;  
  
int main() {  
 char name[50];  
   
 // Reading a word (stops at whitespace)  
 cout << "Enter your name (first name only): ";  
 cin >> name;  
 cout << "Hello, " << name << "!" << endl;  
   
 // Clear input buffer  
 cin.ignore(numeric\_limits<streamsize>::max(), '\n');  
   
 // Reading a line (including spaces)  
 cout << "Enter your full name: ";  
 cin.getline(name, 50);  
 cout << "Hello, " << name << "!" << endl;  
   
 return 0;  
}

#### Input/Output with std::string

#include <iostream>  
#include <string>  
using namespace std;  
  
int main() {  
 string name;  
   
 // Reading a word  
 cout << "Enter your name (first name only): ";  
 cin >> name;  
 cout << "Hello, " << name << "!" << endl;  
   
 // Clear input buffer  
 cin.ignore(numeric\_limits<streamsize>::max(), '\n');  
   
 // Reading a line  
 cout << "Enter your full name: ";  
 getline(cin, name);  
 cout << "Hello, " << name << "!" << endl;  
   
 return 0;  
}

#### Reading Multiple Lines

#include <iostream>  
#include <string>  
#include <sstream>  
using namespace std;  
  
int main() {  
 string input;  
   
 cout << "Enter multiple lines of text (Ctrl+Z or Ctrl+D to end):" << endl;  
   
 // Read multiple lines until EOF  
 stringstream buffer;  
 while (getline(cin, input)) {  
 buffer << input << endl;  
 }  
   
 string allText = buffer.str();  
 cout << "\nYou entered:" << endl;  
 cout << allText;  
   
 return 0;  
}

#### String Streams

String streams provide a way to treat strings like streams, useful for parsing and formatting:

#include <iostream>  
#include <sstream>  
#include <string>  
using namespace std;  
  
int main() {  
 // String to numbers  
 string numStr = "123 45.67";  
 istringstream iss(numStr);  
   
 int intVal;  
 double doubleVal;  
 iss >> intVal >> doubleVal;  
   
 cout << "Parsed values: " << intVal << " and " << doubleVal << endl;  
   
 // Numbers to string  
 ostringstream oss;  
 oss << "Int: " << intVal << ", Double: " << doubleVal;  
 string result = oss.str();  
   
 cout << "Formatted string: " << result << endl;  
   
 return 0;  
}

### 4.3.4 String Conversion Functions

Converting between strings and other data types:

#include <iostream>  
#include <string>  
#include <cstdlib> // For C-style conversions  
using namespace std;  
  
int main() {  
 // C-style conversions  
 const char\* numStr = "123.456";  
 int intVal = atoi(numStr); // String to int  
 double doubleVal = atof(numStr); // String to double  
   
 cout << "C-style conversions: " << intVal << ", " << doubleVal << endl;  
   
 // C++ style conversions (C++11 and later)  
 string str = "42";  
 int num = stoi(str); // String to int  
 double dbl = stod("3.14159"); // String to double  
   
 cout << "C++ style conversions: " << num << ", " << dbl << endl;  
   
 // Converting numbers to strings  
 string numString = to\_string(12345);  
 string dblString = to\_string(67.89);  
   
 cout << "Numbers to strings: " << numString << ", " << dblString << endl;  
   
 return 0;  
}

## 4.4 String Manipulation Problems

Let’s look at common string manipulation problems and their solutions.

### 1. Reverse a String

**Problem**: Reverse a given string.

**Solution 1**: Using std::string

#include <iostream>  
#include <string>  
#include <algorithm>  
using namespace std;  
  
string reverseString(const string& str) {  
 string reversed = str;  
 reverse(reversed.begin(), reversed.end());  
 return reversed;  
}  
  
int main() {  
 string input = "Hello, World!";  
 string reversed = reverseString(input);  
   
 cout << "Original: " << input << endl;  
 cout << "Reversed: " << reversed << endl;  
   
 return 0;  
}

**Solution 2**: Manual reversal with C-style strings

#include <iostream>  
#include <cstring>  
using namespace std;  
  
void reverseString(char\* str) {  
 int length = strlen(str);  
 for (int i = 0; i < length / 2; i++) {  
 char temp = str[i];  
 str[i] = str[length - i - 1];  
 str[length - i - 1] = temp;  
 }  
}  
  
int main() {  
 char str[] = "Hello, World!";  
   
 cout << "Original: " << str << endl;  
 reverseString(str);  
 cout << "Reversed: " << str << endl;  
   
 return 0;  
}

### 2. Check if a String is a Palindrome

**Problem**: Determine if a string reads the same forward and backward.

#include <iostream>  
#include <string>  
#include <algorithm>  
using namespace std;  
  
bool isPalindrome(const string& str) {  
 string cleaned;  
   
 // Remove non-alphanumeric characters and convert to lowercase  
 for (char c : str) {  
 if (isalnum(c)) {  
 cleaned += tolower(c);  
 }  
 }  
   
 // Compare with reverse  
 string reversed = cleaned;  
 reverse(reversed.begin(), reversed.end());  
   
 return cleaned == reversed;  
}  
  
int main() {  
 string testCases[] = {  
 "A man, a plan, a canal: Panama",  
 "race a car",  
 "Was it a car or a cat I saw?",  
 "No lemon, no melon"  
 };  
   
 for (const string& test : testCases) {  
 cout << "\"" << test << "\" is "   
 << (isPalindrome(test) ? "a palindrome" : "not a palindrome") << endl;  
 }  
   
 return 0;  
}

### 3. Find All Occurrences of a Substring

**Problem**: Find all occurrences of a substring within a string.

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
vector<size\_t> findAllOccurrences(const string& str, const string& sub) {  
 vector<size\_t> positions;  
 size\_t pos = str.find(sub, 0);  
   
 while (pos != string::npos) {  
 positions.push\_back(pos);  
 pos = str.find(sub, pos + 1);  
 }  
   
 return positions;  
}  
  
int main() {  
 string text = "The rain in Spain falls mainly in the plain.";  
 string pattern = "in";  
   
 vector<size\_t> occurrences = findAllOccurrences(text, pattern);  
   
 cout << "Occurrences of \"" << pattern << "\" in:" << endl;  
 cout << "\"" << text << "\":" << endl;  
   
 for (size\_t pos : occurrences) {  
 cout << "Position " << pos << ": " << text.substr(pos, 10) << "..." << endl;  
 }  
   
 return 0;  
}

### 4. Capitalize First Letter of Each Word

**Problem**: Convert a string so that the first letter of each word is capitalized.

#include <iostream>  
#include <string>  
#include <cctype>  
using namespace std;  
  
string capitalizeWords(string str) {  
 bool newWord = true;  
   
 for (char& c : str) {  
 if (isspace(c)) {  
 newWord = true;  
 }  
 else if (newWord) {  
 c = toupper(c);  
 newWord = false;  
 }  
 else {  
 c = tolower(c);  
 }  
 }  
   
 return str;  
}  
  
int main() {  
 string sentence = "the quick brown fox jumps over the lazy dog";  
 string capitalized = capitalizeWords(sentence);  
   
 cout << "Original: " << sentence << endl;  
 cout << "Capitalized: " << capitalized << endl;  
   
 return 0;  
}

### 5. Count Words in a String

**Problem**: Count the number of words in a string.

#include <iostream>  
#include <string>  
#include <sstream>  
using namespace std;  
  
int countWords(const string& str) {  
 stringstream ss(str);  
 string word;  
 int count = 0;  
   
 while (ss >> word) {  
 count++;  
 }  
   
 return count;  
}  
  
int main() {  
 string text = "The quick brown fox jumps over the lazy dog";  
 int wordCount = countWords(text);  
   
 cout << "Text: \"" << text << "\"" << endl;  
 cout << "Word count: " << wordCount << endl;  
   
 return 0;  
}

### 6. String Pattern Matching (KMP Algorithm)

**Problem**: Efficiently find a pattern in a text (Knuth-Morris-Pratt algorithm).

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
// Compute the Longest Prefix Suffix (LPS) array  
vector<int> computeLPS(const string& pattern) {  
 int m = pattern.length();  
 vector<int> lps(m, 0);  
   
 int len = 0;  
 int i = 1;  
   
 while (i < m) {  
 if (pattern[i] == pattern[len]) {  
 len++;  
 lps[i] = len;  
 i++;  
 } else {  
 if (len != 0) {  
 len = lps[len - 1];  
 } else {  
 lps[i] = 0;  
 i++;  
 }  
 }  
 }  
   
 return lps;  
}  
  
// KMP search algorithm  
vector<int> KMPSearch(const string& text, const string& pattern) {  
 vector<int> matches;  
 int n = text.length();  
 int m = pattern.length();  
   
 if (m == 0) return matches;  
   
 vector<int> lps = computeLPS(pattern);  
   
 int i = 0; // For text  
 int j = 0; // For pattern  
   
 while (i < n) {  
 if (pattern[j] == text[i]) {  
 i++;  
 j++;  
 }  
   
 if (j == m) {  
 matches.push\_back(i - j);  
 j = lps[j - 1];  
 } else if (i < n && pattern[j] != text[i]) {  
 if (j != 0) {  
 j = lps[j - 1];  
 } else {  
 i++;  
 }  
 }  
 }  
   
 return matches;  
}  
  
int main() {  
 string text = "ABABDABACDABABCABAB";  
 string pattern = "ABABCABAB";  
   
 vector<int> matches = KMPSearch(text, pattern);  
   
 if (matches.empty()) {  
 cout << "Pattern not found in text." << endl;  
 } else {  
 cout << "Pattern found at positions: ";  
 for (int pos : matches) {  
 cout << pos << " ";  
 }  
 cout << endl;  
 }  
   
 return 0;  
}

### 7. Anagram Check

**Problem**: Determine if two strings are anagrams (contain the same characters in different order).

#include <iostream>  
#include <string>  
#include <algorithm>  
using namespace std;  
  
bool areAnagrams(string str1, string str2) {  
 // Remove spaces and convert to lowercase  
 str1.erase(remove\_if(str1.begin(), str1.end(), ::isspace), str1.end());  
 str2.erase(remove\_if(str2.begin(), str2.end(), ::isspace), str2.end());  
   
 transform(str1.begin(), str1.end(), str1.begin(), ::tolower);  
 transform(str2.begin(), str2.end(), str2.begin(), ::tolower);  
   
 if (str1.length() != str2.length()) {  
 return false;  
 }  
   
 // Sort both strings  
 sort(str1.begin(), str1.end());  
 sort(str2.begin(), str2.end());  
   
 // Compare sorted strings  
 return str1 == str2;  
}  
  
int main() {  
 string str1 = "Listen";  
 string str2 = "Silent";  
   
 cout << "\"" << str1 << "\" and \"" << str2 << "\" are "   
 << (areAnagrams(str1, str2) ? "anagrams" : "not anagrams") << endl;  
   
 str1 = "Triangle";  
 str2 = "Integral";  
   
 cout << "\"" << str1 << "\" and \"" << str2 << "\" are "   
 << (areAnagrams(str1, str2) ? "anagrams" : "not anagrams") << endl;  
   
 return 0;  
}

### 8. String Compression

**Problem**: Perform basic string compression using counts of repeated characters.

#include <iostream>  
#include <string>  
using namespace std;  
  
string compressString(const string& str) {  
 if (str.empty()) return "";  
   
 string compressed;  
 char currentChar = str[0];  
 int count = 1;  
   
 for (size\_t i = 1; i < str.length(); i++) {  
 if (str[i] == currentChar) {  
 count++;  
 } else {  
 compressed += currentChar + to\_string(count);  
 currentChar = str[i];  
 count = 1;  
 }  
 }  
   
 // Add the last character group  
 compressed += currentChar + to\_string(count);  
   
 // Return original if compression doesn't save space  
 return (compressed.length() < str.length()) ? compressed : str;  
}  
  
int main() {  
 string test = "aabcccccaaa";  
 string compressed = compressString(test);  
   
 cout << "Original: " << test << endl;  
 cout << "Compressed: " << compressed << endl;  
   
 return 0;  
}

### Best Practices for Array and String Manipulation

1. **Prefer std::string over C-style strings** for most applications
   * Easier memory management
   * Rich functionality
   * Safer operations
2. **Prefer std::vector or std::array over raw arrays**
   * Dynamic sizing (vector)
   * Bounds checking
   * STL algorithm compatibility
3. **Be aware of boundary conditions**
   * Empty arrays/strings
   * Single element cases
   * Maximum possible indices
4. **Use existing algorithms when possible**
   * The <algorithm> header provides many optimized functions
   * Don’t reinvent the wheel
5. **Check for performance bottlenecks**
   * String concatenation in loops can be expensive
   * Consider using StringStream or reserving capacity
6. **Handle special characters and localization**
   * Be careful with Unicode and multi-byte characters
   * Consider using libraries for internationalization
7. **Validate inputs**
   * Check array bounds before accessing
   * Verify string contents meet expected format
8. **Consider space and time complexity**
   * In-place algorithms save memory
   * Some algorithms trade space for speed
9. **Remember that strings are immutable in some languages**
   * In C++, string operations can modify the original string
   * This differs from languages like Java where strings are immutable

By mastering arrays and strings, you’ll build a solid foundation for more complex data structures and algorithms. These fundamental concepts are essential for solving a wide range of programming problems efficiently.

# Chapter 5: Pointers and Memory Management

## 5.1 Introduction to Pointers

Pointers are among the most powerful and challenging features in C++. At their core, pointers are variables that store memory addresses of other variables or objects, enabling direct memory manipulation.

### What is a Pointer?

A pointer is a variable that stores the memory address of another variable. Just as a regular variable gives a name to a value, a pointer gives a name to a memory location.

### Declaring Pointers

To declare a pointer, use the \* symbol after the data type:

data\_type\* pointer\_name;  
// or  
data\_type \*pointer\_name; // Both styles are correct

Examples:

int\* p\_integer; // Pointer to an integer  
double\* p\_double; // Pointer to a double  
char\* p\_character; // Pointer to a character

### Address-of Operator (&)

The address-of operator & returns the memory address of a variable:

#include <iostream>  
using namespace std;  
  
int main() {  
 int num = 42;  
 int\* ptr = &num; // ptr now holds the address of num  
   
 cout << "Value of num: " << num << endl;  
 cout << "Address of num: " << &num << endl;  
 cout << "Value stored in ptr: " << ptr << endl;  
   
 return 0;  
}

Output:

Value of num: 42  
Address of num: 0x7ffcb53e9d0c (this will differ on your system)  
Value stored in ptr: 0x7ffcb53e9d0c

### Dereferencing Operator (\*)

The dereferencing operator \* accesses the value at the memory address stored in a pointer:

#include <iostream>  
using namespace std;  
  
int main() {  
 int num = 42;  
 int\* ptr = &num;  
   
 cout << "Value of num: " << num << endl;  
 cout << "Value pointed to by ptr: " << \*ptr << endl;  
   
 // Modify the value through the pointer  
 \*ptr = 100;  
 cout << "New value of num: " << num << endl;  
   
 return 0;  
}

Output:

Value of num: 42  
Value pointed to by ptr: 42  
New value of num: 100

### Null Pointers

A null pointer doesn’t point to any valid memory location. Always initialize pointers to nullptr if you’re not immediately assigning a valid address:

#include <iostream>  
using namespace std;  
  
int main() {  
 // Modern C++ (C++11 and later)  
 int\* modern\_ptr = nullptr;  
   
 // Pre-C++11 style (avoid in modern code)  
 int\* old\_style\_ptr = NULL;  
 int\* c\_style\_ptr = 0; // Also works but less clear  
   
 // Check before dereferencing  
 if (modern\_ptr != nullptr) {  
 cout << "Value: " << \*modern\_ptr << endl;  
 } else {  
 cout << "Pointer is null, cannot dereference" << endl;  
 }  
   
 return 0;  
}

### Pointer Memory Requirements

Regardless of the data type they point to, pointers have a fixed size determined by the architecture: - 4 bytes on 32-bit systems - 8 bytes on 64-bit systems

#include <iostream>  
using namespace std;  
  
int main() {  
 int\* p\_int;  
 double\* p\_double;  
 char\* p\_char;  
 void\* p\_void;  
   
 cout << "Size of int pointer: " << sizeof(p\_int) << " bytes" << endl;  
 cout << "Size of double pointer: " << sizeof(p\_double) << " bytes" << endl;  
 cout << "Size of char pointer: " << sizeof(p\_char) << " bytes" << endl;  
 cout << "Size of void pointer: " << sizeof(p\_void) << " bytes" << endl;  
   
 return 0;  
}

Output on a 64-bit system:

Size of int pointer: 8 bytes  
Size of double pointer: 8 bytes  
Size of char pointer: 8 bytes  
Size of void pointer: 8 bytes

### Void Pointers

Void pointers can point to data of any type but must be cast before dereferencing:

#include <iostream>  
using namespace std;  
  
int main() {  
 int num = 42;  
 void\* generic\_ptr = &num; // Store address as void pointer  
   
 // Must cast before dereferencing  
 int\* int\_ptr = static\_cast<int\*>(generic\_ptr);  
 cout << "Value: " << \*int\_ptr << endl;  
   
 // Direct dereferencing of void pointer is not allowed  
 // cout << \*generic\_ptr << endl; // Error!  
   
 return 0;  
}

### Pointers to Pointers

You can create pointers to pointers, which are especially useful for dynamic multi-dimensional arrays:

#include <iostream>  
using namespace std;  
  
int main() {  
 int value = 42;  
 int\* ptr = &value; // Pointer to int  
 int\*\* ptr\_to\_ptr = &ptr; // Pointer to pointer to int  
   
 cout << "value: " << value << endl;  
 cout << "\*ptr: " << \*ptr << endl;  
 cout << "\*\*ptr\_to\_ptr: " << \*\*ptr\_to\_ptr << endl;  
   
 \*\*ptr\_to\_ptr = 100; // Change value through double pointer  
 cout << "After modification, value: " << value << endl;  
   
 return 0;  
}

Output:

value: 42  
\*ptr: 42  
\*\*ptr\_to\_ptr: 42  
After modification, value: 100

### Common Pointer Mistakes and Their Solutions

1. **Dereferencing Uninitialized Pointers**

* int\* ptr; // Uninitialized - contains garbage address  
  \*ptr = 10; // DANGER! Undefined behavior  
    
  // Solution:  
  int\* ptr = nullptr; // Initialize to nullptr  
  if (ptr) \*ptr = 10; // Check before dereferencing

1. **Dereferencing Null Pointers**

* int\* ptr = nullptr;  
  \*ptr = 10; // CRASH! Cannot dereference nullptr  
    
  // Solution:  
  if (ptr != nullptr) {  
   \*ptr = 10; // Safe - only deference if not null  
  }

1. **Dangling Pointers** (pointer to released memory)

* int\* createAndReturn() {  
   int local = 10;  
   return &local; // DANGER! Returns address of local variable  
  }  
    
  // Solution:  
  int\* createAndReturn() {  
   static int persistent = 10; // Static variable persists beyond function call  
   return &persistent; // Safe - memory still exists  
  }

1. **Memory Leaks**

* void leakMemory() {  
   int\* ptr = new int(42);  
   // Function ends without delete  
   // Memory leak!  
  }  
    
  // Solution:  
  void noLeak() {  
   int\* ptr = new int(42);  
   // Use ptr...  
   delete ptr; // Always clean up allocated memory  
  }

### Best Practices for Pointers

1. **Always initialize pointers**

* int\* ptr = nullptr; // Good practice

1. **Check for null before dereferencing**

* if (ptr != nullptr) {  
   \*ptr = 100; // Safe  
  }

1. **Delete dynamically allocated memory**

* int\* ptr = new int(42);  
  // Use ptr...  
  delete ptr; // Free memory  
  ptr = nullptr; // Prevent use after free

1. **Use smart pointers for automatic memory management** (covered later in this chapter)

* #include <memory>  
  std::unique\_ptr<int> ptr = std::make\_unique<int>(42);  
  // Memory automatically freed when ptr goes out of scope

1. **Prefer references over pointers when possible**

* void modify(int& num) { // Reference parameter  
   num \*= 2;  
  }

1. **Use const for pointers to read-only data**

* const int\* ptr; // Pointer to constant int (data cannot be modified)  
  int\* const ptr; // Constant pointer to int (pointer cannot be reassigned)  
  const int\* const ptr; // Constant pointer to constant int (neither can change)

## 5.2 Pointer Arithmetic

Pointer arithmetic allows you to navigate through memory by performing mathematical operations on pointers. This is particularly useful for array manipulation and memory management.

### Basic Arithmetic Operations

#### Incrementing and Decrementing

#include <iostream>  
using namespace std;  
  
int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
 int\* ptr = numbers; // Points to first element  
   
 cout << "Initial value: " << \*ptr << endl; // 10  
   
 ptr++; // Move to next element  
 cout << "After increment: " << \*ptr << endl; // 20  
   
 ptr--; // Move back to previous element  
 cout << "After decrement: " << \*ptr << endl; // 10  
   
 return 0;  
}

#### Adding and Subtracting Integers

#include <iostream>  
using namespace std;  
  
int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
 int\* ptr = numbers;  
   
 cout << "Value at index 0: " << \*ptr << endl; // 10  
 cout << "Value at index 2: " << \*(ptr + 2) << endl; // 30  
 cout << "Value at index 4: " << \*(ptr + 4) << endl; // 50  
   
 // Pointer arithmetic with array indexing  
 cout << "Using array notation: " << ptr[3] << endl; // 40  
   
 return 0;  
}

### How Pointer Arithmetic Works

When we add an integer n to a pointer, it actually advances the pointer by n \* sizeof(data\_type) bytes, not just by n bytes. This ensures we advance to the correct memory location based on the type’s size.

#include <iostream>  
using namespace std;  
  
int main() {  
 cout << "Size of char: " << sizeof(char) << " bytes" << endl;  
 cout << "Size of int: " << sizeof(int) << " bytes" << endl;  
 cout << "Size of double: " << sizeof(double) << " bytes" << endl;  
   
 char char\_array[5] = {'A', 'B', 'C', 'D', 'E'};  
 int int\_array[5] = {10, 20, 30, 40, 50};  
 double double\_array[5] = {1.1, 2.2, 3.3, 4.4, 5.5};  
   
 char\* char\_ptr = char\_array;  
 int\* int\_ptr = int\_array;  
 double\* double\_ptr = double\_array;  
   
 // Print initial and incremented addresses  
 cout << "\nChar pointer:" << endl;  
 cout << "Initial address: " << static\_cast<void\*>(char\_ptr) << endl;  
 char\_ptr += 1; // Advances by 1 byte  
 cout << "After ptr += 1: " << static\_cast<void\*>(char\_ptr) << endl;  
   
 cout << "\nInt pointer:" << endl;  
 cout << "Initial address: " << int\_ptr << endl;  
 int\_ptr += 1; // Advances by sizeof(int) bytes (typically 4)  
 cout << "After ptr += 1: " << int\_ptr << endl;  
   
 cout << "\nDouble pointer:" << endl;  
 cout << "Initial address: " << double\_ptr << endl;  
 double\_ptr += 1; // Advances by sizeof(double) bytes (typically 8)  
 cout << "After ptr += 1: " << double\_ptr << endl;  
   
 return 0;  
}

Note: We use static\_cast<void\*> for the char pointer because cout would interpret a char\* as a C-string and try to print the characters it points to.

### Pointer Subtraction

You can subtract pointers of the same type to find the number of elements between them:

#include <iostream>  
using namespace std;  
  
int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
 int\* start = numbers; // Points to first element  
 int\* end = &numbers[4]; // Points to last element  
   
 ptrdiff\_t elements = end - start;  
 cout << "Elements between pointers: " << elements << endl; // 4  
   
 return 0;  
}

### Pointer Comparison

Pointers can be compared using relational operators:

#include <iostream>  
using namespace std;  
  
int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
 int\* p1 = &numbers[1]; // Points to 20  
 int\* p2 = &numbers[3]; // Points to 40  
   
 if (p1 < p2) {  
 cout << "p1 points to an element before p2" << endl;  
 } else {  
 cout << "p1 points to an element after or equal to p2" << endl;  
 }  
   
 cout << "Elements between p1 and p2: " << (p2 - p1) << endl;  
   
 return 0;  
}

### Traversing Arrays with Pointers

Pointers provide an efficient way to traverse arrays:

#include <iostream>  
using namespace std;  
  
int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
 int\* end = numbers + 5; // Points past the last element  
   
 cout << "Method 1: Using pointer arithmetic in a for loop" << endl;  
 for (int\* ptr = numbers; ptr < end; ptr++) {  
 cout << \*ptr << " ";  
 }  
 cout << endl;  
   
 cout << "Method 2: Using array indexing with pointer arithmetic" << endl;  
 int\* base = numbers;  
 for (int i = 0; i < 5; i++) {  
 cout << \*(base + i) << " "; // Equivalent to base[i]  
 }  
 cout << endl;  
   
 cout << "Method 3: Using a while loop" << endl;  
 int\* ptr = numbers;  
 while (ptr < end) {  
 cout << \*ptr++ << " "; // Print current value, then increment pointer  
 }  
 cout << endl;  
   
 return 0;  
}

### Limitations and Safety Concerns

1. **Going Beyond Bounds**

If you increment or decrement a pointer beyond the bounds of the array it initially pointed to, you’ll be accessing invalid memory locations:

int arr[5] = {10, 20, 30, 40, 50};  
int\* ptr = arr;  
ptr += 10; // Points far beyond the array  
\*ptr = 100; // DANGER: Undefined behavior, potential crash

1. **Invalid Operations**

You cannot: - Add two pointers - Multiply or divide pointers - Perform bitwise operations on pointers

int\* p1 = &x;  
int\* p2 = &y;  
int\* sum = p1 + p2; // Error: Cannot add two pointers

1. **Arithmetic on void Pointers**

Pointer arithmetic on void pointers isn’t allowed without casting:

void\* ptr = &x;  
ptr++; // Error: Cannot increment void pointer  
void\* next = static\_cast<char\*>(ptr) + 1; // Cast first

### Real-World Applications

1. **Implementing Custom String Functions**

#include <iostream>  
using namespace std;  
  
// Custom strlen function  
size\_t custom\_strlen(const char\* str) {  
 const char\* start = str;  
 // Increment until null terminator is found  
 while (\*str != '\0') {  
 str++;  
 }  
 return str - start; // Distance is the length  
}  
  
int main() {  
 const char\* message = "Hello, World!";  
 cout << "Length of string: " << custom\_strlen(message) << endl;  
 return 0;  
}

1. **Efficient Array Processing**

#include <iostream>  
using namespace std;  
  
// Sum array elements using pointer arithmetic  
int sum\_array(const int\* arr, int size) {  
 const int\* end = arr + size;  
 int total = 0;  
   
 while (arr < end) {  
 total += \*arr++;  
 }  
   
 return total;  
}  
  
int main() {  
 int numbers[] = {10, 20, 30, 40, 50};  
 cout << "Sum of array: " << sum\_array(numbers, 5) << endl;  
 return 0;  
}

### Best Practices for Pointer Arithmetic

1. **Always check bounds**

* if (ptr >= array && ptr < array + size) {  
   // Safe to dereference  
  }

1. **Use modern C++ containers when possible**

* #include <vector>  
  vector<int> numbers = {10, 20, 30, 40, 50};  
  // No manual pointer manipulation needed

1. **Consider using iterators instead**

* for (auto it = numbers.begin(); it != numbers.end(); ++it) {  
   cout << \*it << " ";  
  }

1. **Prefer array indexing for better readability**

* int arr[5] = {10, 20, 30, 40, 50};  
  // Instead of \*(arr + i), prefer arr[i]

## 5.3 Pointers and Arrays

In C++, arrays and pointers are closely related. Understanding this relationship is fundamental to mastering C++ memory management.

### The Relationship Between Arrays and Pointers

When you use an array name in most expressions, it “decays” (is implicitly converted) into a pointer to its first element:

#include <iostream>  
using namespace std;  
  
int main() {  
 int numbers[5] = {10, 20, 30, 40, 50};  
 int\* ptr = numbers; // Same as &numbers[0]  
   
 cout << "Array name: " << numbers << endl;  
 cout << "Address of first element: " << &numbers[0] << endl;  
 cout << "Pointer value: " << ptr << endl;  
   
 // All three point to the same memory location  
   
 return 0;  
}

### Key Differences Between Arrays and Pointers

Despite this close relationship, arrays and pointers are not identical:

1. **Size Information**

#include <iostream>  
using namespace std;  
  
int main() {  
 int numbers[5] = {10, 20, 30, 40, 50};  
 int\* ptr = numbers;  
   
 cout << "sizeof(numbers): " << sizeof(numbers) << " bytes" << endl; // 5 \* sizeof(int)  
 cout << "sizeof(ptr): " << sizeof(ptr) << " bytes" << endl; // Size of pointer only  
   
 return 0;  
}

The array name holds the size information, while a pointer doesn’t.

1. **Reassignment**

int arr[5] = {10, 20, 30, 40, 50};  
int\* ptr = arr;  
  
ptr = nullptr; // OK - pointer can be reassigned  
// arr = nullptr; // ERROR - array name cannot be reassigned

1. **Behavior in sizeof**

int arr[5];  
int\* ptr = arr;  
  
sizeof(arr); // Returns 5 \* sizeof(int)  
sizeof(ptr); // Returns sizeof(pointer), typically 4 or 8 bytes

1. **Array Decay When Passing to Functions**

#include <iostream>  
using namespace std;  
  
void printSize(int arr[]) {  
 // arr is a pointer here, not an array  
 cout << "Size inside function: " << sizeof(arr) << " bytes" << endl; // Size of pointer  
}  
  
int main() {  
 int numbers[5] = {10, 20, 30, 40, 50};  
 cout << "Size in main: " << sizeof(numbers) << " bytes" << endl; // 5 \* sizeof(int)  
   
 printSize(numbers);  
   
 return 0;  
}

### Accessing Array Elements with Pointers

You can access array elements using pointers in several ways:

#include <iostream>  
using namespace std;  
  
int main() {  
 int numbers[5] = {10, 20, 30, 40, 50};  
 int\* ptr = numbers;  
   
 // Method 1: Array notation with array name  
 cout << "numbers[2]: " << numbers[2] << endl;  
   
 // Method 2: Pointer arithmetic with array name  
 cout << "\*(numbers + 2): " << \*(numbers + 2) << endl;  
   
 // Method 3: Array notation with pointer  
 cout << "ptr[2]: " << ptr[2] << endl;  
   
 // Method 4: Pointer arithmetic with pointer  
 cout << "\*(ptr + 2): " << \*(ptr + 2) << endl;  
   
 return 0;  
}

All four methods access the same element (the third element, value 30).

### Multi-dimensional Arrays and Pointers

Working with multi-dimensional arrays using pointers is more complex:

#include <iostream>  
using namespace std;  
  
int main() {  
 int matrix[3][4] = {  
 {11, 12, 13, 14},  
 {21, 22, 23, 24},  
 {31, 32, 33, 34}  
 };  
   
 // Pointer to the first row  
 int (\*rowPtr)[4] = matrix;  
 cout << "First element via row pointer: " << (\*rowPtr)[0] << endl; // 11  
   
 // Pointer to the first element  
 int\* elemPtr = &matrix[0][0];  
 cout << "First element via element pointer: " << \*elemPtr << endl; // 11  
   
 // Access elements  
 cout << "Element at row 1, col 2: " << \*(\*(rowPtr + 1) + 2) << endl; // 23  
 cout << "Same element using direct access: " << matrix[1][2] << endl; // 23  
   
 // Accessing matrix elements with a single pointer  
 cout << "Using linear indexing with element pointer:" << endl;  
 for (int i = 0; i < 3; i++) {  
 for (int j = 0; j < 4; j++) {  
 cout << \*(elemPtr + i\*4 + j) << " ";  
 }  
 cout << endl;  
 }  
   
 return 0;  
}

### Dynamic Arrays with Pointers

Pointers are essential for creating dynamically-sized arrays:

#include <iostream>  
using namespace std;  
  
int main() {  
 int size;  
 cout << "Enter size for dynamic array: ";  
 cin >> size;  
   
 // Allocate memory  
 int\* dynamicArray = new int[size];  
   
 // Initialize array  
 for (int i = 0; i < size; i++) {  
 dynamicArray[i] = i \* 10;  
 }  
   
 // Print values  
 cout << "Array values:" << endl;  
 for (int i = 0; i < size; i++) {  
 cout << dynamicArray[i] << " ";  
 }  
 cout << endl;  
   
 // Free memory  
 delete[] dynamicArray;  
   
 return 0;  
}

### Creating Dynamic 2D Arrays

There are several approaches to creating dynamic 2D arrays:

#include <iostream>  
using namespace std;  
  
int main() {  
 int rows = 3, cols = 4;  
   
 // Method 1: Array of pointers  
 int\*\* array1 = new int\*[rows];  
 for (int i = 0; i < rows; i++) {  
 array1[i] = new int[cols];  
 // Initialize  
 for (int j = 0; j < cols; j++) {  
 array1[i][j] = i \* cols + j;  
 }  
 }  
   
 // Method 2: Single block with manual indexing  
 int\* array2 = new int[rows \* cols];  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < cols; j++) {  
 array2[i \* cols + j] = i \* cols + j;  
 }  
 }  
   
 // Clean up Method 1  
 for (int i = 0; i < rows; i++) {  
 delete[] array1[i];  
 }  
 delete[] array1;  
   
 // Clean up Method 2  
 delete[] array2;  
   
 return 0;  
}

### Arrays of Pointers vs. Pointers to Arrays

#include <iostream>  
using namespace std;  
  
int main() {  
 // Array of pointers  
 int\* ptr\_array[3]; // Array of 3 int pointers  
   
 // Initialize with pointers to different variables  
 int a = 10, b = 20, c = 30;  
 ptr\_array[0] = &a;  
 ptr\_array[1] = &b;  
 ptr\_array[2] = &c;  
   
 cout << "Array of pointers values: ";  
 for (int i = 0; i < 3; i++) {  
 cout << \*ptr\_array[i] << " ";  
 }  
 cout << endl;  
   
 // Pointer to an array  
 int numbers[5] = {1, 2, 3, 4, 5};  
 int (\*array\_ptr)[5] = &numbers; // Pointer to an array of 5 ints  
   
 cout << "Values through array pointer: ";  
 for (int i = 0; i < 5; i++) {  
 cout << (\*array\_ptr)[i] << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### Best Practices for Arrays and Pointers

1. **Prefer Modern Alternatives**

* #include <vector>  
  #include <array>  
    
  // Instead of raw dynamic arrays  
  vector<int> dynamicArray(size);  
    
  // Instead of fixed-size arrays  
  array<int, 5> fixedArray = {1, 2, 3, 4, 5};

1. **Pass Array Size**

* void processArray(int\* arr, int size) {  
   // Always include size information  
  }

1. **Use Range-based For Loops When Possible**

* for (int value : fixedArray) {  
   cout << value << " ";  
  }

1. **Consider Templates for Size Safety**

* template<size\_t N>  
  void processArray(int (&arr)[N]) {  
   // Size is known at compile time  
   for (size\_t i = 0; i < N; i++) {  
   // Process arr[i]  
   }  
  }

1. **Avoid Pointer Arithmetic When Better Alternatives Exist**

* // Prefer this  
  for (int i = 0; i < size; i++) {  
   cout << arr[i];  
  }  
    
  // Over this  
  for (int\* p = arr; p < arr + size; p++) {  
   cout << \*p;  
  }

## 5.4 Pointers to Functions

Function pointers allow you to store and invoke functions dynamically. They’re essential for callbacks, plugins, and implementing function dispatch tables.

### Basic Function Pointer Syntax

The syntax for declaring a function pointer is:

return\_type (\*pointer\_name)(parameter\_types);

Example:

#include <iostream>  
using namespace std;  
  
// Function to be pointed to  
int add(int x, int y) {  
 return x + y;  
}  
  
int main() {  
 // Declare function pointer  
 int (\*operation)(int, int);  
   
 // Assign function address to the pointer  
 operation = add; // No & needed, function name decays to address  
   
 // Call the function via pointer  
 int result = operation(5, 3);  
 cout << "Result: " << result << endl; // 8  
   
 return 0;  
}

### Using typedef and using for Function Pointers

To simplify complicated function pointer syntax, use typedef or using:

#include <iostream>  
using namespace std;  
  
int add(int x, int y) { return x + y; }  
int subtract(int x, int y) { return x - y; }  
int multiply(int x, int y) { return x \* y; }  
int divide(int x, int y) { return y != 0 ? x / y : 0; }  
  
// Old style with typedef  
typedef int (\*MathFunction)(int, int);  
  
// Modern style with using (C++11)  
using Operation = int (\*)(int, int);  
  
int main() {  
 MathFunction op1 = add;  
 Operation op2 = multiply;  
   
 cout << "5 + 3 = " << op1(5, 3) << endl; // 8  
 cout << "5 \* 3 = " << op2(5, 3) << endl; // 15  
   
 return 0;  
}

### Arrays of Function Pointers

Function pointers can be organized into arrays to create function tables:

#include <iostream>  
using namespace std;  
  
int add(int x, int y) { return x + y; }  
int subtract(int x, int y) { return x - y; }  
int multiply(int x, int y) { return x \* y; }  
int divide(int x, int y) { return y != 0 ? x / y : 0; }  
  
int main() {  
 // Array of function pointers  
 int (\*operations[4])(int, int) = {add, subtract, multiply, divide};  
 string op\_names[4] = {"+", "-", "\*", "/"};  
   
 int a = 10, b = 5;  
   
 // Use each function in the array  
 for (int i = 0; i < 4; i++) {  
 cout << a << " " << op\_names[i] << " " << b << " = ";  
 cout << operations[i](a, b) << endl;  
 }  
   
 return 0;  
}

### Function Pointers as Arguments

Function pointers allow you to pass behavior to other functions:

#include <iostream>  
#include <vector>  
using namespace std;  
  
// Takes a function pointer as parameter  
void transform(vector<int>& values, int (\*func)(int)) {  
 for (size\_t i = 0; i < values.size(); i++) {  
 values[i] = func(values[i]);  
 }  
}  
  
// Functions to be passed  
int square(int x) { return x \* x; }  
int triple(int x) { return x \* 3; }  
  
int main() {  
 vector<int> numbers = {1, 2, 3, 4, 5};  
   
 cout << "Original values: ";  
 for (int num : numbers) cout << num << " ";  
 cout << endl;  
   
 // Apply square function  
 transform(numbers, square);  
 cout << "After square: ";  
 for (int num : numbers) cout << num << " ";  
 cout << endl;  
   
 // Apply triple function  
 transform(numbers, triple);  
 cout << "After triple: ";  
 for (int num : numbers) cout << num << " ";  
 cout << endl;  
   
 return 0;  
}

### Member Function Pointers

Member function pointers are used to point to class methods:

#include <iostream>  
#include <string>  
using namespace std;  
  
class Calculator {  
public:  
 int add(int x, int y) { return x + y; }  
 int subtract(int x, int y) { return x - y; }  
 int multiply(int x, int y) { return x \* y; }  
 int divide(int x, int y) { return y != 0 ? x / y : 0; }  
};  
  
int main() {  
 // Declare a member function pointer  
 int (Calculator::\*operation)(int, int) = &Calculator::add;  
   
 // Create an object  
 Calculator calc;  
   
 // Call the member function through the pointer  
 cout << "5 + 3 = " << (calc.\*operation)(5, 3) << endl;  
   
 // Change the function pointer  
 operation = &Calculator::multiply;  
 cout << "5 \* 3 = " << (calc.\*operation)(5, 3) << endl;  
   
 return 0;  
}

### Function Pointers vs. std::function (C++11)

Modern C++ provides std::function for more flexibility:

#include <iostream>  
#include <functional>  
#include <vector>  
using namespace std;  
  
// Traditional function  
int add(int x, int y) { return x + y; }  
  
// Function object (functor)  
struct Multiplier {  
 int operator()(int x, int y) const { return x \* y; }  
};  
  
int main() {  
 // Different ways to create std::function objects  
 function<int(int,int)> f1 = add;  
 function<int(int,int)> f2 = [](int x, int y) { return x - y; }; // Lambda  
 function<int(int,int)> f3 = Multiplier(); // Function object  
   
 cout << "10 + 5 = " << f1(10, 5) << endl;  
 cout << "10 - 5 = " << f2(10, 5) << endl;  
 cout << "10 \* 5 = " << f3(10, 5) << endl;  
   
 // Store in a container  
 vector<function<int(int,int)>> operations = {f1, f2, f3};  
   
 int x = 20, y = 4;  
 for (auto& op : operations) {  
 cout << "Result: " << op(x, y) << endl;  
 }  
   
 return 0;  
}

### Practical Applications

1. **Callbacks**

#include <iostream>  
#include <functional>  
using namespace std;  
  
class Button {  
private:  
 string label;  
 function<void()> clickHandler;  
   
public:  
 Button(const string& text, function<void()> handler)  
 : label(text), clickHandler(handler) {}  
   
 void click() {  
 cout << "Button '" << label << "' clicked!" << endl;  
 if (clickHandler) {  
 clickHandler();  
 }  
 }  
};  
  
void sayHello() {  
 cout << "Hello, World!" << endl;  
}  
  
int main() {  
 // Button with named function as callback  
 Button btn1("Greet", sayHello);  
   
 // Button with lambda as callback  
 Button btn2("Exit", []() {  
 cout << "Exiting application..." << endl;  
 });  
   
 btn1.click(); // Triggers sayHello  
 btn2.click(); // Triggers lambda  
   
 return 0;  
}

1. **Strategy Pattern**

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <functional>  
using namespace std;  
  
// Different sorting strategies  
bool ascending(int a, int b) {  
 return a < b;  
}  
  
bool descending(int a, int b) {  
 return a > b;  
}  
  
class Sorter {  
private:  
 function<bool(int,int)> compareStrategy;  
   
public:  
 Sorter(function<bool(int,int)> strategy) : compareStrategy(strategy) {}  
   
 void sort(vector<int>& data) {  
 std::sort(data.begin(), data.end(), compareStrategy);  
 }  
};  
  
int main() {  
 vector<int> numbers = {5, 2, 9, 1, 7, 3};  
   
 cout << "Original: ";  
 for (int n : numbers) cout << n << " ";  
 cout << endl;  
   
 // Sort with ascending strategy  
 Sorter ascendingSorter(ascending);  
 ascendingSorter.sort(numbers);  
   
 cout << "Ascending: ";  
 for (int n : numbers) cout << n << " ";  
 cout << endl;  
   
 // Sort with descending strategy  
 Sorter descendingSorter(descending);  
 descendingSorter.sort(numbers);  
   
 cout << "Descending: ";  
 for (int n : numbers) cout << n << " ";  
 cout << endl;  
   
 return 0;  
}

1. **Plugin Architecture**

#include <iostream>  
#include <string>  
#include <map>  
#include <functional>  
using namespace std;  
  
// Plugin interface  
using PluginFunction = function<void(const string&)>;  
  
class PluginManager {  
private:  
 map<string, PluginFunction> plugins;  
   
public:  
 void registerPlugin(const string& name, PluginFunction func) {  
 plugins[name] = func;  
 }  
   
 void executePlugin(const string& name, const string& data) {  
 auto it = plugins.find(name);  
 if (it != plugins.end()) {  
 it->second(data);  
 } else {  
 cout << "Plugin '" << name << "' not found!" << endl;  
 }  
 }  
};  
  
// Plugin implementations  
void textPlugin(const string& data) {  
 cout << "Text Plugin: Processing '" << data << "'" << endl;  
}  
  
void numberPlugin(const string& data) {  
 try {  
 int value = stoi(data);  
 cout << "Number Plugin: Received " << value << ", square is " << (value \* value) << endl;  
 } catch (...) {  
 cout << "Number Plugin: Invalid number format" << endl;  
 }  
}  
  
int main() {  
 PluginManager manager;  
   
 // Register plugins  
 manager.registerPlugin("text", textPlugin);  
 manager.registerPlugin("number", numberPlugin);  
 manager.registerPlugin("logger", [](const string& data) {  
 cout << "LOG: " << data << endl;  
 });  
   
 // Execute plugins  
 manager.executePlugin("text", "Hello, World!");  
 manager.executePlugin("number", "42");  
 manager.executePlugin("logger", "Application started");  
 manager.executePlugin("unknown", "test"); // Not found  
   
 return 0;  
}

### Best Practices for Function Pointers

1. **Prefer std::function for complex use cases**

* // Instead of:  
  void (\*callback)(int, const string&);  
    
  // Use:  
  function<void(int, const string&)> callback;

1. **Check for null before calling**

* if (callback != nullptr) {  
   callback(42, "test");  
  }

1. **Use meaningful typedefs or using statements**

* using ErrorHandler = void (\*)(const string&, int);

1. **Consider alternatives**
   * Templates for compile-time polymorphism
   * Virtual functions for object-oriented polymorphism
   * Lambda expressions for simple callbacks
2. **Document expected behavior**

* // Takes a comparison function that should return:  
  // - Negative value if a < b  
  // - Zero if a == b  
  // - Positive value if a > b  
  void sort(int\* array, int size, int (\*compare)(int a, int b));

# Chapter 5: Pointers and Memory Management (Part 2)

## 5.5 Dynamic Memory Allocation (new, delete)

Dynamic memory allocation allows you to request and release memory during program execution rather than at compile time. This flexibility is essential when you need to create data structures whose size is determined at runtime.

### The Stack vs. The Heap

Before diving into dynamic memory allocation, it’s important to understand the two main memory regions in C++:

1. **Stack**:
   * Automatically managed memory
   * Fast allocation and deallocation (simple pointer increment/decrement)
   * Limited in size (typically a few MB)
   * Stores local variables, function parameters, and return addresses
   * Memory is freed automatically when variables go out of scope
   * LIFO (Last In, First Out) structure
2. **Heap** (Free Store):
   * Manually managed memory in traditional C++
   * Slower allocation/deallocation compared to stack
   * Limited only by available virtual memory
   * Stores dynamically allocated objects
   * Must be explicitly freed to prevent memory leaks
   * No inherent structure (allocations/deallocations can happen in any order)

### Basic Dynamic Memory Allocation

In C++, you use the new operator to allocate memory on the heap and the delete operator to free that memory:

#include <iostream>  
using namespace std;  
  
int main() {  
 // Allocating a single integer  
 int\* ptr = new int; // Allocates memory for an int and returns its address  
   
 // Initializing the allocated memory  
 \*ptr = 42;  
 cout << "Value: " << \*ptr << endl;  
   
 // Freeing the memory when done  
 delete ptr; // Returns memory to the system  
   
 // After deletion, ptr is a dangling pointer  
 // Best practice: set to nullptr  
 ptr = nullptr;  
   
 return 0;  
}

### Allocation with Initialization

You can initialize memory as you allocate it:

#include <iostream>  
using namespace std;  
  
int main() {  
 // Allocate and initialize an integer  
 int\* p1 = new int(42); // Direct initialization  
 cout << "p1 value: " << \*p1 << endl;  
   
 // C++11 uniform initialization  
 double\* p2 = new double{3.14159};  
 cout << "p2 value: " << \*p2 << endl;  
   
 // Clean up  
 delete p1;  
 delete p2;  
   
 return 0;  
}

### Dynamic Arrays

To allocate arrays dynamically, use the array form of new and delete:

#include <iostream>  
using namespace std;  
  
int main() {  
 // Get size at runtime  
 int size;  
 cout << "Enter array size: ";  
 cin >> size;  
   
 // Allocate an array of the specified size  
 int\* array = new int[size];  
   
 // Initialize the array  
 for (int i = 0; i < size; i++) {  
 array[i] = i \* 10;  
 }  
   
 // Display the array  
 cout << "Array values: ";  
 for (int i = 0; i < size; i++) {  
 cout << array[i] << " ";  
 }  
 cout << endl;  
   
 // Free the array memory with array delete  
 delete[] array; // IMPORTANT: Use delete[] for arrays, not delete  
   
 return 0;  
}

### Allocation Failure Handling

By default, new throws a std::bad\_alloc exception if memory allocation fails. You can use the nothrow version to get a null pointer instead:

#include <iostream>  
#include <new> // For std::nothrow  
using namespace std;  
  
int main() {  
 // Default behavior: throws exception on failure  
 try {  
 int\* ptr = new int[1000000000000]; // Very large allocation likely to fail  
 // If we get here, allocation succeeded  
 delete[] ptr;  
 } catch (const bad\_alloc& e) {  
 cout << "Memory allocation failed: " << e.what() << endl;  
 }  
   
 // Alternative: nothrow version  
 int\* ptr = new (nothrow) int[1000000000000];  
   
 if (ptr == nullptr) {  
 cout << "Memory allocation failed with nothrow" << endl;  
 } else {  
 cout << "Allocation succeeded" << endl;  
 delete[] ptr;  
 }  
   
 return 0;  
}

### Dynamic Allocation of Objects

When you use new with classes, the constructor is automatically called, and delete calls the destructor:

#include <iostream>  
#include <string>  
using namespace std;  
  
class Person {  
private:  
 string name;  
 int age;  
   
public:  
 Person(const string& n, int a) : name(n), age(a) {  
 cout << "Constructing " << name << endl;  
 }  
   
 ~Person() {  
 cout << "Destructing " << name << endl;  
 }  
   
 void display() {  
 cout << name << " is " << age << " years old" << endl;  
 }  
};  
  
int main() {  
 // Allocate a single object  
 Person\* p1 = new Person("Alice", 30);  
 p1->display();  
   
 // Allocate an array of objects  
 Person\* group = new Person[3] {  
 {"Bob", 25},  
 {"Charlie", 32},  
 {"Diana", 28}  
 };  
   
 // Access array objects  
 for (int i = 0; i < 3; i++) {  
 group[i].display();  
 }  
   
 // Clean up  
 delete p1; // Calls destructor for a single object  
 delete[] group; // Calls destructor for each array element  
   
 return 0;  
}

### Placement new

Placement new allows you to construct an object at a specific, pre-allocated memory address:

#include <iostream>  
#include <new> // For placement new  
using namespace std;  
  
class Complex {  
public:  
 double real;  
 double imag;  
   
 Complex(double r, double i) : real(r), imag(i) {  
 cout << "Complex constructed at " << this << endl;  
 }  
   
 ~Complex() {  
 cout << "Complex destructed at " << this << endl;  
 }  
};  
  
int main() {  
 // Allocate raw memory (no constructor called)  
 char\* memory = new char[sizeof(Complex)];  
   
 // Construct object at the allocated memory  
 Complex\* obj = new (memory) Complex(3.0, 4.0);  
   
 cout << "Real: " << obj->real << ", Imaginary: " << obj->imag << endl;  
   
 // Must explicitly call destructor (placement new doesn't register with delete)  
 obj->~Complex();  
   
 // Free the raw memory  
 delete[] memory;  
   
 return 0;  
}

### Common Mistakes and Pitfalls

1. **Memory Leaks**: Forgetting to delete what you new

void leakMemory() {  
 int\* ptr = new int(42);  
 // No delete before function returns - memory leaked!  
}

1. **Double Deletion**: Deleting the same memory twice

int\* ptr = new int(42);  
delete ptr;  
// ... some code ...  
delete ptr; // ERROR: Double free, undefined behavior

1. **Using Deleted Memory**: Accessing memory after it’s been freed

int\* ptr = new int(42);  
delete ptr;  
\*ptr = 100; // ERROR: Accessing freed memory

1. **Mismatched new/delete**: Using wrong form of delete

int\* single = new int;  
int\* array = new int[10];  
  
delete[] single; // ERROR: Using array delete for single object  
delete array; // ERROR: Using single delete for array

### Advanced: Custom Memory Allocators

For performance-critical applications, you can create custom allocators:

#include <iostream>  
using namespace std;  
  
class PoolAllocator {  
private:  
 char\* memory\_pool;  
 size\_t used;  
 size\_t capacity;  
  
public:  
 PoolAllocator(size\_t size) : capacity(size), used(0) {  
 memory\_pool = new char[size];  
 }  
   
 ~PoolAllocator() {  
 delete[] memory\_pool;  
 }  
   
 void\* allocate(size\_t size) {  
 if (used + size > capacity) {  
 return nullptr; // Not enough space  
 }  
   
 void\* result = memory\_pool + used;  
 used += size;  
 return result;  
 }  
   
 // Simple allocator doesn't support individual deallocation  
};  
  
int main() {  
 // Create a 1KB memory pool  
 PoolAllocator pool(1024);  
   
 // Allocate some integers from the pool  
 int\* a = static\_cast<int\*>(pool.allocate(sizeof(int)));  
 int\* b = static\_cast<int\*>(pool.allocate(sizeof(int)));  
   
 \*a = 42;  
 \*b = 100;  
   
 cout << "a: " << \*a << endl;  
 cout << "b: " << \*b << endl;  
   
 // No need to delete individual allocations  
 // Pool is freed when it goes out of scope  
   
 return 0;  
}

### Best Practices

1. **Always match new with delete, and new[] with delete[]**
2. **Set pointers to nullptr after deleting them**

* delete ptr;  
  ptr = nullptr; // Prevents accidental use after delete

1. **Check for allocation failure with nothrow**

* int\* ptr = new(nothrow) int[1000000];  
  if (!ptr) {  
   // Handle allocation failure  
  }

1. **Prefer stack allocation when possible**

* // Better (stack allocation)  
  int values[10];  
    
  // Worse (heap allocation, manual management)  
  int\* values = new int[10];  
  // ... use values ...  
  delete[] values;

1. **Use containers and smart pointers for dynamic memory management** (covered later)

* #include <vector>  
  vector<int> values(10); // Dynamic array with automatic memory management

## 5.6 Dangling Pointers and Memory Leaks

Memory management errors are among the most common and problematic bugs in C++. Two particularly significant issues are dangling pointers and memory leaks.

### Dangling Pointers

A dangling pointer is a pointer that references a memory location that is no longer valid, typically because the memory has been freed or gone out of scope.

#### Common Causes of Dangling Pointers

1. **Using a Pointer After Deletion**

#include <iostream>  
using namespace std;  
  
int main() {  
 int\* ptr = new int(42);  
 delete ptr;  
   
 // Danger! ptr is now dangling  
 \*ptr = 100; // Undefined behavior  
 cout << \*ptr << endl; // Undefined behavior  
   
 return 0;  
}

1. **Returning Address of Local Variables**

#include <iostream>  
using namespace std;  
  
int\* createDanglingPointer() {  
 int local = 42; // Local variable on stack  
 return &local; // DANGER: Returns address of local variable  
}  
  
int main() {  
 int\* ptr = createDanglingPointer();  
 // ptr is dangling - points to memory that was on the stack  
 // and is no longer valid  
 cout << \*ptr << endl; // Undefined behavior  
   
 return 0;  
}

1. **Pointers to Freed Memory in Data Structures**

#include <iostream>  
using namespace std;  
  
struct Node {  
 int data;  
 Node\* next;  
   
 Node(int d) : data(d), next(nullptr) {}  
};  
  
int main() {  
 Node\* head = new Node(1);  
 Node\* second = new Node(2);  
 head->next = second;  
   
 // Delete the second node  
 delete second;  
   
 // head->next is now dangling  
 cout << head->next->data << endl; // DANGER: Undefined behavior  
   
 delete head;  
 return 0;  
}

### Detecting and Preventing Dangling Pointers

1. **Set Pointers to nullptr After Deletion**

#include <iostream>  
using namespace std;  
  
int main() {  
 int\* ptr = new int(42);  
   
 // Use the pointer  
 cout << "Value: " << \*ptr << endl;  
   
 // Clean up and nullify  
 delete ptr;  
 ptr = nullptr;  
   
 // Safe check before use  
 if (ptr != nullptr) {  
 cout << \*ptr << endl;  
 } else {  
 cout << "Pointer is null" << endl;  
 }  
   
 return 0;  
}

1. **Handle Owner Responsibility Clearly**

#include <iostream>  
using namespace std;  
  
class ResourceManager {  
private:  
 int\* resource;  
  
public:  
 ResourceManager() : resource(new int(42)) {}  
   
 ~ResourceManager() {  
 delete resource;  
 resource = nullptr;  
 }  
   
 // Prevent copying to avoid multiple owners  
 ResourceManager(const ResourceManager&) = delete;  
 ResourceManager& operator=(const ResourceManager&) = delete;  
   
 int getValue() const { return \*resource; }  
};  
  
int main() {  
 ResourceManager manager;  
 cout << "Value: " << manager.getValue() << endl;  
   
 // Resource automatically freed when manager goes out of scope  
 return 0;  
}

1. **Use Defensive Programming with External APIs**

#include <iostream>  
using namespace std;  
  
// Simulated external API function that might free a pointer  
void externalFunction(int\* ptr) {  
 // Might delete ptr internally  
 delete ptr;  
}  
  
int main() {  
 int\* ptr = new int(42);  
   
 // Call to external function that might delete ptr  
 externalFunction(ptr);  
   
 // Defensive check (imperfect but better than nothing)  
 // Note: This doesn't guarantee safety, as ptr might point to reallocated memory  
 if (ptr != nullptr) {  
 cout << "Assuming pointer is still valid: " << \*ptr << endl;  
 }  
   
 // Better approach: Don't use ptr after the external call  
 ptr = nullptr; // Explicitly indicate we no longer own it  
   
 return 0;  
}

### Memory Leaks

A memory leak occurs when dynamically allocated memory is not freed, causing the program to gradually consume more and more memory, potentially leading to degraded performance or crashes.

#### Common Causes of Memory Leaks

1. **Forgetting to Delete Allocated Memory**

#include <iostream>  
using namespace std;  
  
void leakMemory() {  
 int\* ptr = new int(42);  
 // Function ends without deleting ptr  
 // The memory at ptr is now leaked  
}  
  
int main() {  
 for (int i = 0; i < 1000; i++) {  
 leakMemory(); // Leaks 1000 integers worth of memory  
 }  
   
 cout << "Program continues with leaked memory" << endl;  
   
 return 0;  
}

1. **Losing the Pointer Before Deletion**

#include <iostream>  
using namespace std;  
  
int main() {  
 int\* ptr = new int(42);  
 ptr = new int(100); // Original memory is leaked  
   
 delete ptr; // Only frees the second allocation  
   
 return 0;  
}

1. **Memory Leaks in Exception Scenarios**

#include <iostream>  
#include <stdexcept>  
using namespace std;  
  
void processData() {  
 int\* data = new int[1000];  
   
 // If this throws, data won't be deleted  
 if (rand() % 2 == 0) {  
 throw runtime\_error("Random error occurred");  
 }  
   
 // Normal cleanup path  
 delete[] data;  
}  
  
int main() {  
 try {  
 processData();  
 } catch (const exception& e) {  
 cout << "Exception caught: " << e.what() << endl;  
 // Memory is leaked here  
 }  
   
 return 0;  
}

1. **Circular References in Data Structures**

#include <iostream>  
using namespace std;  
  
struct Node {  
 int data;  
 Node\* next;  
   
 Node(int d) : data(d), next(nullptr) {}  
};  
  
void createCycle() {  
 Node\* first = new Node(1);  
 Node\* second = new Node(2);  
   
 first->next = second;  
 second->next = first; // Creates a cycle  
   
 // No way to delete all nodes from here  
 // Both nodes are leaked  
}  
  
int main() {  
 createCycle();  
   
 cout << "Program continues with leaked memory" << endl;  
   
 return 0;  
}

### Detecting and Preventing Memory Leaks

1. **Always Delete What You Allocate**

#include <iostream>  
using namespace std;  
  
int main() {  
 // Proper memory management  
 int\* ptr = new int(42);  
   
 // Use the pointer...  
 cout << "Value: " << \*ptr << endl;  
   
 // Clean up  
 delete ptr;  
 ptr = nullptr;  
   
 return 0;  
}

1. **Use RAII (Resource Acquisition Is Initialization)**

#include <iostream>  
#include <fstream>  
using namespace std;  
  
class IntArray {  
private:  
 int\* data;  
 size\_t size;  
   
public:  
 IntArray(size\_t s) : size(s) {  
 data = new int[size];  
 cout << "Allocated array of size " << size << endl;  
 }  
   
 ~IntArray() {  
 delete[] data;  
 cout << "Freed array of size " << size << endl;  
 data = nullptr;  
 }  
   
 // Disable copying to prevent double deletion  
 IntArray(const IntArray&) = delete;  
 IntArray& operator=(const IntArray&) = delete;  
   
 int& operator[](size\_t index) {  
 return data[index];  
 }  
};  
  
void processData() {  
 IntArray arr(10); // Resource acquisition  
   
 // Use the array  
 arr[0] = 42;  
   
 // No need to manually free memory  
 // arr's destructor will be called automatically when it goes out of scope  
}  
  
int main() {  
 processData();  
 cout << "Function completed with no leaks" << endl;  
   
 return 0;  
}

1. **Exception-Safe Resource Management**

#include <iostream>  
#include <stdexcept>  
using namespace std;  
  
class DataProcessor {  
private:  
 int\* buffer;  
  
public:  
 DataProcessor(size\_t size) : buffer(new int[size]) {  
 cout << "Buffer allocated" << endl;  
 }  
   
 ~DataProcessor() {  
 cout << "Buffer deallocated" << endl;  
 delete[] buffer;  
 }  
   
 void process() {  
 // Potentially throws an exception  
 if (rand() % 2 == 0) {  
 throw runtime\_error("Processing error");  
 }  
   
 cout << "Processing completed successfully" << endl;  
 }  
};  
  
int main() {  
 try {  
 DataProcessor processor(1000);  
 processor.process();  
 } catch (const exception& e) {  
 cout << "Exception caught: " << e.what() << endl;  
 }  
   
 cout << "Program continues with no memory leaks" << endl;  
   
 return 0;  
}

### Tools for Detecting Memory Issues

1. **Valgrind** - A powerful tool for memory debugging, detecting leaks, and profiling

* # Compile with debugging information  
  g++ -g program.cpp -o program  
    
  # Run with Valgrind  
  valgrind --leak-check=full ./program

1. **Address Sanitizer** - Part of Clang/GCC compilers

* # Compile with AddressSanitizer  
  g++ -fsanitize=address -g program.cpp -o program  
    
  # Run normally, leaks will be reported  
  ./program

1. **Static Analysis Tools**
   * CppCheck
   * Clang Static Analyzer
   * PVS-Studio
2. **Custom Leak Detection** - For specific scenarios

#include <iostream>  
#include <map>  
using namespace std;  
  
// Simple memory leak detector  
class LeakDetector {  
private:  
 static map<void\*, size\_t> allocations;  
   
public:  
 static void\* allocate(size\_t size) {  
 void\* ptr = malloc(size);  
 allocations[ptr] = size;  
 return ptr;  
 }  
   
 static void deallocate(void\* ptr) {  
 auto it = allocations.find(ptr);  
 if (it != allocations.end()) {  
 allocations.erase(it);  
 free(ptr);  
 }  
 }  
   
 static void reportLeaks() {  
 if (allocations.empty()) {  
 cout << "No memory leaks detected" << endl;  
 return;  
 }  
   
 cout << "Memory leaks detected:" << endl;  
 size\_t total = 0;  
 for (const auto& pair : allocations) {  
 cout << " Address " << pair.first << ": " << pair.second << " bytes" << endl;  
 total += pair.second;  
 }  
 cout << "Total leaked: " << total << " bytes" << endl;  
 }  
};  
  
map<void\*, size\_t> LeakDetector::allocations;  
  
// Override global new/delete  
void\* operator new(size\_t size) {  
 return LeakDetector::allocate(size);  
}  
  
void operator delete(void\* ptr) noexcept {  
 LeakDetector::deallocate(ptr);  
}  
  
int main() {  
 int\* p1 = new int(42);  
 int\* p2 = new int(100);  
   
 // Deliberately leak p1  
 delete p2;  
   
 LeakDetector::reportLeaks();  
   
 return 0;  
}

## 5.7 Smart Pointers (unique\_ptr, shared\_ptr, weak\_ptr)

Smart pointers are class templates that provide automatic memory management for dynamically allocated objects. They ensure proper deletion of objects when they are no longer needed, helping to prevent memory leaks and other memory-related problems.

### Types of Smart Pointers

Modern C++ provides three main types of smart pointers in the <memory> header:

1. **std::unique\_ptr** - Exclusive ownership model
2. **std::shared\_ptr** - Shared ownership model
3. **std::weak\_ptr** - Non-owning reference to a shared\_ptr

### std::unique\_ptr

std::unique\_ptr implements exclusive ownership of a dynamically allocated object. Only one unique\_ptr can own the object at any time, and when that unique\_ptr is destroyed or reassigned, the object is automatically deleted.

#### Basic Usage

#include <iostream>  
#include <memory>  
using namespace std;  
  
class Resource {  
public:  
 Resource(int id) : id\_(id) {  
 cout << "Resource " << id\_ << " constructed" << endl;  
 }  
   
 ~Resource() {  
 cout << "Resource " << id\_ << " destroyed" << endl;  
 }  
   
 void use() {  
 cout << "Using resource " << id\_ << endl;  
 }  
   
private:  
 int id\_;  
};  
  
int main() {  
 // Create a unique\_ptr  
 unique\_ptr<Resource> res1(new Resource(1));  
   
 // Use the resource  
 res1->use();  
   
 // Automatic cleanup when res1 goes out of scope  
   
 return 0;  
}

#### Creating with make\_unique (C++14)

#include <iostream>  
#include <memory>  
using namespace std;  
  
int main() {  
 // C++14 way: Preferred for exception safety and efficiency  
 unique\_ptr<Resource> res = make\_unique<Resource>(1);  
   
 // Pre-C++14 way  
 unique\_ptr<Resource> res2(new Resource(2));  
   
 return 0;  
}

#### Moving Ownership

unique\_ptr can’t be copied, but ownership can be transferred using std::move:

#include <iostream>  
#include <memory>  
#include <utility> // For std::move  
using namespace std;  
  
int main() {  
 unique\_ptr<Resource> res1 = make\_unique<Resource>(1);  
   
 // Transfer ownership  
 unique\_ptr<Resource> res2 = move(res1);  
   
 // res1 is now empty (null)  
 if (res1 == nullptr) {  
 cout << "res1 is null after move" << endl;  
 }  
   
 // res2 now owns the resource  
 res2->use();  
   
 return 0;  
}

#### Array Support

unique\_ptr supports arrays with a specialized version:

#include <iostream>  
#include <memory>  
using namespace std;  
  
int main() {  
 // Array of integers  
 unique\_ptr<int[]> numbers = make\_unique<int[]>(5);  
   
 // Initialize array  
 for (int i = 0; i < 5; i++) {  
 numbers[i] = i \* 10;  
 }  
   
 // Access elements  
 for (int i = 0; i < 5; i++) {  
 cout << "numbers[" << i << "] = " << numbers[i] << endl;  
 }  
   
 // Automatically deletes the array when numbers goes out of scope  
   
 return 0;  
}

#### Custom Deleters

You can specify custom cleanup operations:

#include <iostream>  
#include <memory>  
#include <cstdio>  
using namespace std;  
  
int main() {  
 // Custom deleter for FILE\*  
 auto fileDeleter = [](FILE\* file) {  
 cout << "Closing file..." << endl;  
 if (file) {  
 fclose(file);  
 }  
 };  
   
 // Open a file  
 FILE\* rawFile = fopen("example.txt", "w");  
 if (!rawFile) {  
 cerr << "Could not open file" << endl;  
 return 1;  
 }  
   
 // Create unique\_ptr with custom deleter  
 unique\_ptr<FILE, decltype(fileDeleter)> file(rawFile, fileDeleter);  
   
 // Use the file  
 fprintf(file.get(), "Hello, Smart Pointers!");  
   
 // File automatically closed when file goes out of scope  
   
 return 0;  
}

### std::shared\_ptr

std::shared\_ptr implements shared ownership of a dynamically allocated object. Multiple shared\_ptr instances can own the same object, and the object is deleted only when the last owning shared\_ptr is destroyed or reassigned.

#### Basic Usage

#include <iostream>  
#include <memory>  
using namespace std;  
  
class Resource {  
public:  
 Resource(int id) : id\_(id) {  
 cout << "Resource " << id\_ << " constructed" << endl;  
 }  
   
 ~Resource() {  
 cout << "Resource " << id\_ << " destroyed" << endl;  
 }  
   
 void use() {  
 cout << "Using resource " << id\_ << endl;  
 }  
   
private:  
 int id\_;  
};  
  
int main() {  
 // Create a shared\_ptr  
 shared\_ptr<Resource> res1 = make\_shared<Resource>(1);  
   
 // Create another shared\_ptr pointing to the same resource  
 shared\_ptr<Resource> res2 = res1;  
   
 cout << "Resource count: " << res1.use\_count() << endl; // 2  
   
 // Both can use the resource  
 res1->use();  
 res2->use();  
   
 // Resource deleted when both res1 and res2 go out of scope  
   
 return 0;  
}

#### Creating with make\_shared

make\_shared is more efficient than using the constructor with new because it allocates memory for both the object and the control block in a single operation:

#include <iostream>  
#include <memory>  
using namespace std;  
  
int main() {  
 // Efficient way: One allocation for object and control block  
 shared\_ptr<Resource> res1 = make\_shared<Resource>(1);  
   
 // Less efficient way: Separate allocations  
 shared\_ptr<Resource> res2(new Resource(2));  
   
 return 0;  
}

#### Sharing and Use Count

Each shared\_ptr internally maintains a reference count:

#include <iostream>  
#include <memory>  
using namespace std;  
  
void useResource(shared\_ptr<Resource> res) {  
 cout << "In function, count: " << res.use\_count() << endl;  
 res->use();  
 // res is destroyed when function exits, decreasing count  
}  
  
int main() {  
 shared\_ptr<Resource> res = make\_shared<Resource>(1);  
 cout << "Initial count: " << res.use\_count() << endl; // 1  
   
 {  
 shared\_ptr<Resource> res2 = res;  
 cout << "Block scope count: " << res.use\_count() << endl; // 2  
 }  
   
 cout << "After block scope: " << res.use\_count() << endl; // 1  
   
 useResource(res); // Temporarily increases count  
   
 cout << "After function: " << res.use\_count() << endl; // 1  
   
 return 0;  
}

#### Custom Deleters

Similar to unique\_ptr, shared\_ptr supports custom deleters:

#include <iostream>  
#include <memory>  
#include <functional>  
using namespace std;  
  
class Socket {  
public:  
 Socket(int handle) : handle\_(handle) {  
 cout << "Socket " << handle\_ << " opened" << endl;  
 }  
   
 int getHandle() const { return handle\_; }  
   
private:  
 int handle\_;  
};  
  
void closeSocket(Socket\* socket) {  
 cout << "Socket " << socket->getHandle() << " closed" << endl;  
 delete socket;  
}  
  
int main() {  
 // Socket with custom deleter  
 shared\_ptr<Socket> socket(new Socket(12345), closeSocket);  
   
 cout << "Using socket " << socket->getHandle() << endl;  
   
 // Socket automatically closed when socket ptr goes out of scope  
   
 return 0;  
}

### std::weak\_ptr

std::weak\_ptr provides a non-owning reference to an object managed by a shared\_ptr. It doesn’t affect the reference count, and it can’t access the object directly without being converted to a shared\_ptr first.

#### Breaking Circular References

The primary use of weak\_ptr is to break circular references that would otherwise cause memory leaks with shared\_ptr:

#include <iostream>  
#include <memory>  
using namespace std;  
  
class Node;  
using NodePtr = shared\_ptr<Node>;  
using WeakNodePtr = weak\_ptr<Node>;  
  
class Node {  
public:  
 NodePtr next;  
 WeakNodePtr prev; // Using weak\_ptr to avoid circular reference  
   
 int data;  
   
 Node(int d) : data(d) {  
 cout << "Node " << d << " created" << endl;  
 }  
   
 ~Node() {  
 cout << "Node " << data << " destroyed" << endl;  
 }  
};  
  
int main() {  
 // Create nodes  
 NodePtr node1 = make\_shared<Node>(1);  
 NodePtr node2 = make\_shared<Node>(2);  
   
 // Create circular reference  
 node1->next = node2;  
 node2->prev = node1; // Weak reference doesn't increase count  
   
 cout << "Node1 use count: " << node1.use\_count() << endl; // 1  
 cout << "Node2 use count: " << node2.use\_count() << endl; // 2 (node1->next and node2)  
   
 // Check if weak\_ptr is still valid  
 if (auto p = node2->prev.lock()) {  
 cout << "Node2->prev points to node with data: " << p->data << endl;  
 }  
   
 // When we go out of scope, all nodes are properly deleted  
 // If we had used shared\_ptr for prev, nodes would never be deleted  
   
 return 0;  
}

#### Using weak\_ptr

weak\_ptr can’t be directly dereferenced; you must first convert it to a shared\_ptr using the lock() method:

#include <iostream>  
#include <memory>  
using namespace std;  
  
int main() {  
 // Create a shared\_ptr  
 shared\_ptr<int> shared = make\_shared<int>(42);  
   
 // Create a weak\_ptr from it  
 weak\_ptr<int> weak = shared;  
   
 cout << "shared count: " << shared.use\_count() << endl; // 1  
   
 // Convert weak\_ptr to shared\_ptr to use it  
 if (auto locked = weak.lock()) {  
 cout << "Value: " << \*locked << endl;  
 cout << "shared count during access: " << shared.use\_count() << endl; // 2  
 } else {  
 cout << "Original object no longer exists" << endl;  
 }  
   
 // Reset the original shared\_ptr  
 shared.reset();  
   
 // Try to access again  
 if (auto locked = weak.lock()) {  
 cout << "Value is still available" << endl;  
 } else {  
 cout << "Original object has been deleted" << endl;  
 }  
   
 return 0;  
}

#### Checking Expiration

You can check if a weak\_ptr refers to an object that still exists without converting it to a shared\_ptr:

#include <iostream>  
#include <memory>  
using namespace std;  
  
int main() {  
 weak\_ptr<int> weak;  
   
 {  
 shared\_ptr<int> shared = make\_shared<int>(42);  
 weak = shared;  
   
 cout << "weak.expired(): " << weak.expired() << endl; // 0 (false)  
 }  
   
 // After the shared\_ptr is destroyed  
 cout << "weak.expired(): " << weak.expired() << endl; // 1 (true)  
   
 return 0;  
}

### Choosing the Right Smart Pointer

1. **Use unique\_ptr when:**
   * You need exclusive ownership (only one owner)
   * You want to transfer ownership (via std::move)
   * You need custom deleters but don’t want reference counting overhead
   * For resource management within a single scope or class
2. **Use shared\_ptr when:**
   * Ownership is shared among multiple objects
   * You don’t know which owner will last longest
   * You need shared resources to be cleaned up only when all users are done
3. **Use weak\_ptr when:**
   * You need a non-owning reference to a resource managed by shared\_ptr
   * To break circular references between shared\_ptrs
   * When you need to track if an object still exists without affecting its lifetime
4. **Don’t use smart pointers when:**
   * Managing non-heap resources (use RAII wrappers instead)
   * Working with C-style APIs that expect raw pointers (use .get() method)
   * The overhead would be unacceptable (very rare in practice)

### Smart Pointer Best Practices

1. **Prefer make\_unique and make\_shared over direct constructor use**

* // Good  
  auto ptr = make\_unique<Resource>(args);  
    
  // Less optimal  
  unique\_ptr<Resource> ptr(new Resource(args));

1. **Don’t mix raw and smart pointers for ownership**

* Resource\* raw = new Resource();  
  shared\_ptr<Resource> smart(raw);  
  delete raw; // DANGER: Double deletion

1. **Use appropriate access patterns**

* // For unique\_ptr or shared\_ptr  
  resource->method();  
    
  // For weak\_ptr  
  if (auto locked = weak.lock()) {  
   locked->method();  
  }

1. **Pass by reference when you don’t want to transfer or share ownership**

* void useResource(const Resource& resource) {  
   // Function doesn't participate in ownership  
  }

1. **Consider const smart pointers for read-only access**

* const shared\_ptr<Resource> ptr = make\_shared<Resource>();  
  // ptr.reset() is not allowed

## 5.8 RAII and Scope-Based Resource Management

RAII (Resource Acquisition Is Initialization) is a programming idiom that ensures proper management of resources by tying their lifetimes to the lifetime of objects. It’s one of the most important idioms in C++ and forms the basis for many C++ standard library components.

### The RAII Principle

The core idea of RAII is:

1. Encapsulate each resource in a class
2. Acquire the resource in the constructor
3. Release the resource in the destructor
4. Use instances of the class to manage the resource

This ensures that resources are automatically cleaned up when objects go out of scope, even in the presence of exceptions or early returns.

### Basic RAII Example

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
class FileHandler {  
private:  
 ofstream file;  
 string filename;  
  
public:  
 FileHandler(const string& fname) : filename(fname) {  
 file.open(filename);  
 if (!file.is\_open()) {  
 throw runtime\_error("Could not open file: " + filename);  
 }  
 cout << "File opened: " << filename << endl;  
 }  
   
 ~FileHandler() {  
 if (file.is\_open()) {  
 file.close();  
 cout << "File closed: " << filename << endl;  
 }  
 }  
   
 void write(const string& data) {  
 if (!file.is\_open()) {  
 throw runtime\_error("File not open");  
 }  
 file << data << endl;  
 }  
};  
  
void processFile(const string& filename) {  
 try {  
 FileHandler handler(filename);  
 handler.write("Line 1");  
 handler.write("Line 2");  
   
 // File is automatically closed when handler goes out of scope,  
 // even if an exception is thrown  
   
 } catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
}  
  
int main() {  
 processFile("example.txt");  
 return 0;  
}

### Benefits of RAII

1. **Automatic Resource Management**
   * No need to manually track resource cleanup
   * Resources are released in the reverse order of acquisition
2. **Exception Safety**
   * Resources are properly cleaned up even if exceptions occur
   * No resource leaks in error paths
3. **Clear Ownership Semantics**
   * The object that owns a resource is responsible for its cleanup
   * Avoids ambiguity about who should clean up

### RAII for Different Resource Types

#### Memory Management

Smart pointers are the primary RAII mechanism for memory:

#include <iostream>  
#include <memory>  
using namespace std;  
  
class Resource {  
public:  
 Resource() { cout << "Resource acquired" << endl; }  
 ~Resource() { cout << "Resource released" << endl; }  
 void use() { cout << "Resource used" << endl; }  
};  
  
void useResource() {  
 unique\_ptr<Resource> res = make\_unique<Resource>();  
 res->use();  
   
 // Memory automatically freed when res goes out of scope  
}  
  
int main() {  
 cout << "Entering function" << endl;  
 useResource();  
 cout << "Function complete" << endl;  
   
 return 0;  
}

#### File Handling

The standard library’s file streams implement RAII:

#include <iostream>  
#include <fstream>  
using namespace std;  
  
void writeToFile(const string& filename, const string& data) {  
 ofstream file(filename); // Opens file  
   
 if (file.is\_open()) {  
 file << data << endl;  
 }  
   
 // File automatically closed when file goes out of scope  
}  
  
int main() {  
 writeToFile("example.txt", "Hello, RAII!");  
 cout << "File operation complete" << endl;  
   
 return 0;  
}

#### Mutex Locking

RAII for thread synchronization with std::lock\_guard:

#include <iostream>  
#include <mutex>  
#include <thread>  
#include <vector>  
using namespace std;  
  
class ThreadSafeCounter {  
private:  
 mutex mtx;  
 int counter = 0;  
  
public:  
 void increment() {  
 lock\_guard<mutex> lock(mtx); // RAII lock  
 counter++;  
 // lock automatically released when lock\_guard goes out of scope  
 }  
   
 int getValue() {  
 lock\_guard<mutex> lock(mtx); // RAII lock  
 return counter;  
 }  
};  
  
void incrementTask(ThreadSafeCounter& counter, int times) {  
 for (int i = 0; i < times; i++) {  
 counter.increment();  
 }  
}  
  
int main() {  
 ThreadSafeCounter counter;  
   
 vector<thread> threads;  
 for (int i = 0; i < 10; i++) {  
 threads.push\_back(thread(incrementTask, ref(counter), 1000));  
 }  
   
 for (auto& t : threads) {  
 t.join();  
 }  
   
 cout << "Final counter value: " << counter.getValue() << endl;  
   
 return 0;  
}

#### Database Connections

#include <iostream>  
#include <stdexcept>  
using namespace std;  
  
// Mock database connection  
class DBConnection {  
public:  
 DBConnection(const string& connString) {  
 cout << "Opening database connection to " << connString << endl;  
 // Imagine real connection logic here  
 if (connString.empty()) {  
 throw runtime\_error("Invalid connection string");  
 }  
 }  
   
 ~DBConnection() {  
 cout << "Closing database connection" << endl;  
 }  
   
 void executeQuery(const string& query) {  
 cout << "Executing query: " << query << endl;  
 }  
};  
  
// RAII wrapper  
class DBTransaction {  
private:  
 DBConnection& conn;  
 bool committed = false;  
  
public:  
 DBTransaction(DBConnection& connection) : conn(connection) {  
 cout << "Beginning transaction" << endl;  
 }  
   
 void commit() {  
 cout << "Committing transaction" << endl;  
 committed = true;  
 }  
   
 ~DBTransaction() {  
 if (!committed) {  
 cout << "Rolling back uncomitted transaction" << endl;  
 }  
 }  
};  
  
void performDatabaseOperation(const string& connString) {  
 DBConnection connection(connString);  
   
 try {  
 DBTransaction transaction(connection);  
   
 connection.executeQuery("SELECT \* FROM users");  
 connection.executeQuery("UPDATE users SET active = true");  
   
 transaction.commit();  
   
 } catch (const exception& e) {  
 cout << "Error: " << e.what() << endl;  
 // Transaction automatically rolled back  
 // Connection automatically closed  
 }  
   
 // Connection automatically closed when it goes out of scope  
}  
  
int main() {  
 try {  
 performDatabaseOperation("server=localhost;db=test");  
 } catch (const exception& e) {  
 cerr << "Database operation failed: " << e.what() << endl;  
 }  
   
 return 0;  
}

### Creating Custom RAII Classes

To create your own RAII wrappers:

#include <iostream>  
using namespace std;  
  
// Example of a custom resource  
struct RawResource {  
 void\* data;  
   
 static RawResource\* create() {  
 cout << "Creating raw resource" << endl;  
 return new RawResource{malloc(1024)};  
 }  
   
 static void destroy(RawResource\* res) {  
 cout << "Destroying raw resource" << endl;  
 free(res->data);  
 delete res;  
 }  
};  
  
// RAII wrapper  
class ManagedResource {  
private:  
 RawResource\* resource;  
  
public:  
 ManagedResource() : resource(RawResource::create()) {}  
   
 ~ManagedResource() {  
 RawResource::destroy(resource);  
 }  
   
 // Disable copying  
 ManagedResource(const ManagedResource&) = delete;  
 ManagedResource& operator=(const ManagedResource&) = delete;  
   
 // Enable moving  
 ManagedResource(ManagedResource&& other) noexcept : resource(other.resource) {  
 other.resource = nullptr;  
 }  
   
 ManagedResource& operator=(ManagedResource&& other) noexcept {  
 if (this != &other) {  
 if (resource) {  
 RawResource::destroy(resource);  
 }  
 resource = other.resource;  
 other.resource = nullptr;  
 }  
 return \*this;  
 }  
   
 void\* getData() const {  
 return resource ? resource->data : nullptr;  
 }  
};  
  
void useResource() {  
 ManagedResource res; // Resource created  
   
 // Use the resource  
 void\* data = res.getData();  
 cout << "Resource address: " << data << endl;  
   
 // Resource automatically cleaned up when res goes out of scope  
}  
  
int main() {  
 useResource();  
 cout << "Function completed" << endl;  
   
 return 0;  
}

### RAII and the Rule of Three/Five/Zero

The “Rule of Three” (expanded to “Rule of Five” in C++11, and “Rule of Zero” as a modern alternative) is closely related to RAII:

1. **Rule of Three**: If a class requires a custom destructor, copy constructor, or copy assignment operator, it probably needs all three.
2. **Rule of Five**: In C++11 and later, adds move constructor and move assignment operator to the list.
3. **Rule of Zero**: Design classes to avoid the need for custom resource management, delegating it to member objects.

#include <iostream>  
#include <memory>  
using namespace std;  
  
// Rule of Five example  
class Buffer {  
private:  
 size\_t size;  
 unsigned char\* data;  
  
public:  
 // Constructor  
 Buffer(size\_t size) : size(size), data(new unsigned char[size]) {  
 cout << "Constructing buffer of size " << size << endl;  
 fill(data, data + size, 0);  
 }  
   
 // Destructor  
 ~Buffer() {  
 cout << "Destroying buffer of size " << size << endl;  
 delete[] data;  
 }  
   
 // Copy constructor  
 Buffer(const Buffer& other) : size(other.size), data(new unsigned char[other.size]) {  
 cout << "Copy constructing buffer" << endl;  
 memcpy(data, other.data, size);  
 }  
   
 // Copy assignment operator  
 Buffer& operator=(const Buffer& other) {  
 cout << "Copy assigning buffer" << endl;  
 if (this != &other) {  
 delete[] data;  
 size = other.size;  
 data = new unsigned char[size];  
 memcpy(data, other.data, size);  
 }  
 return \*this;  
 }  
   
 // Move constructor  
 Buffer(Buffer&& other) noexcept : size(other.size), data(other.data) {  
 cout << "Move constructing buffer" << endl;  
 other.data = nullptr;  
 other.size = 0;  
 }  
   
 // Move assignment operator  
 Buffer& operator=(Buffer&& other) noexcept {  
 cout << "Move assigning buffer" << endl;  
 if (this != &other) {  
 delete[] data;  
 size = other.size;  
 data = other.data;  
 other.data = nullptr;  
 other.size = 0;  
 }  
 return \*this;  
 }  
   
 // Access methods  
 size\_t getSize() const { return size; }  
 unsigned char\* getData() { return data; }  
};  
  
// Rule of Zero example  
class Document {  
private:  
 string title;  
 unique\_ptr<Buffer> content; // Uses smart pointer for RAII  
  
public:  
 Document(string title, size\_t size)   
 : title(move(title)), content(make\_unique<Buffer>(size)) {}  
   
 // No explicit destructor, copy/move constructors or assignment operators needed  
 // The default ones will handle the resources properly  
};  
  
int main() {  
 // Test Rule of Five  
 Buffer b1(100);  
 Buffer b2 = b1; // Copy constructor  
 Buffer b3(200);  
 b3 = b1; // Copy assignment  
   
 Buffer b4 = move(b2); // Move constructor  
 b3 = move(b4); // Move assignment  
   
 // Test Rule of Zero  
 Document doc("Report", 1024);  
   
 return 0;  
}

### Advanced RAII Techniques

#### Scope Guards

Scope guards execute actions at scope exit, even when exceptions occur:

#include <iostream>  
#include <functional>  
using namespace std;  
  
class ScopeGuard {  
private:  
 function<void()> action;  
 bool dismissed = false;  
  
public:  
 explicit ScopeGuard(function<void()> action) : action(move(action)) {}  
   
 ~ScopeGuard() {  
 if (!dismissed) {  
 action();  
 }  
 }  
   
 void dismiss() {  
 dismissed = true;  
 }  
   
 // Prevent copying and moving  
 ScopeGuard(const ScopeGuard&) = delete;  
 ScopeGuard& operator=(const ScopeGuard&) = delete;  
 ScopeGuard(ScopeGuard&&) = delete;  
 ScopeGuard& operator=(ScopeGuard&&) = delete;  
};  
  
void someFunctionWithCleanup() {  
 cout << "Allocating temporary resources..." << endl;  
   
 // Setup cleanup  
 ScopeGuard cleanup([]() {  
 cout << "Cleanup action executed!" << endl;  
 });  
   
 cout << "Working with resources..." << endl;  
   
 // Simulate potential exit points  
 int choice = rand() % 3;  
   
 if (choice == 0) {  
 cout << "Early return, cleanup will still happen" << endl;  
 return; // Early return  
 }  
 else if (choice == 1) {  
 cout << "Exception thrown, cleanup will still happen" << endl;  
 throw runtime\_error("Something went wrong"); // Exception  
 }  
   
 cout << "Function completed normally" << endl;  
 // Optional: dismiss the guard if you don't want the action to run  
 // cleanup.dismiss();  
}  
  
int main() {  
 try {  
 someFunctionWithCleanup();  
 } catch (const exception& e) {  
 cerr << "Exception caught: " << e.what() << endl;  
 }  
   
 return 0;  
}

#### Deferred Actions with Destructors

#include <iostream>  
#include <string>  
using namespace std;  
  
class Timer {  
private:  
 string name;  
 clock\_t start;  
  
public:  
 Timer(const string& operationName)   
 : name(operationName), start(clock()) {  
 cout << "Starting operation: " << name << endl;  
 }  
   
 ~Timer() {  
 clock\_t end = clock();  
 double elapsed = double(end - start) / CLOCKS\_PER\_SEC;  
 cout << "Operation " << name << " took " << elapsed << " seconds" << endl;  
 }  
};  
  
void performSlowOperation() {  
 Timer t("SlowOperation");  
   
 // Simulate work  
 for (int i = 0; i < 100000000; i++) {  
 int x = i \* i;  
 (void)x; // Prevent optimization  
 }  
   
 // Timer automatically reports duration on function exit  
}  
  
int main() {  
 performSlowOperation();  
 return 0;  
}

### RAII and Concurrency

RAII is particularly valuable in concurrent code:

#include <iostream>  
#include <mutex>  
#include <thread>  
#include <vector>  
using namespace std;  
  
class SharedResource {  
private:  
 mutex resourceMutex;  
 vector<int> data;  
  
public:  
 void addData(int value) {  
 // lock\_guard is an RAII wrapper for mutex  
 lock\_guard<mutex> lock(resourceMutex);  
   
 cout << "Thread " << this\_thread::get\_id()   
 << " adding value: " << value << endl;  
 data.push\_back(value);  
   
 // mutex automatically unlocked when lock goes out of scope  
 }  
   
 void processData() {  
 // unique\_lock is more flexible than lock\_guard  
 unique\_lock<mutex> lock(resourceMutex);  
   
 if (data.empty()) {  
 cout << "No data to process" << endl;  
 return; // mutex unlocked here  
 }  
   
 cout << "Processing " << data.size() << " values..." << endl;  
 int sum = 0;  
 for (int value : data) {  
 sum += value;  
 }  
 cout << "Sum: " << sum << endl;  
 data.clear();  
   
 // mutex unlocked when lock goes out of scope  
 }  
};  
  
void workerThread(SharedResource& resource, int startVal, int count) {  
 for (int i = 0; i < count; i++) {  
 resource.addData(startVal + i);  
 this\_thread::sleep\_for(chrono::milliseconds(10));  
 }  
}  
  
int main() {  
 SharedResource resource;  
   
 // Create threads  
 thread t1(workerThread, ref(resource), 100, 5);  
 thread t2(workerThread, ref(resource), 200, 5);  
   
 // Wait for threads to complete  
 t1.join();  
 t2.join();  
   
 // Process the collected data  
 resource.processData();  
   
 return 0;  
}

### Best Practices for RAII

1. **Design resource handling classes to follow RAII principles**
   * Acquire resources in constructors
   * Release resources in destructors
   * Don’t throw exceptions from destructors
2. **Use existing RAII wrappers when available**
   * Smart pointers for dynamic memory
   * Standard containers for collections
   * Stream objects for I/O
   * lock\_guard and unique\_lock for mutexes
3. **Follow the Rule of Zero when possible**
   * Delegate resource management to member objects
   * Use standard library containers and smart pointers
4. **Make resource ownership clear**
   * Use move semantics to transfer ownership
   * Use smart pointers to express ownership semantics
5. **Properly handle self-assignment and exceptions**
   * Implement copy-and-swap idiom for assignment operators
   * Use the “acquire resources, and then modify state” idiom
6. **Control object copying and moving**
   * Delete copy operations for exclusive ownership
   * Implement deep copying for shared resources
   * Use move semantics for transferable resources
7. **Keep destructors simple and noexcept**
   * Avoid throwing exceptions from destructors
   * Handle cleanup failures gracefully

RAII is one of the most powerful idioms in C++, enabling robust resource management with clean, exception-safe code. Mastering RAII is essential for writing reliable C++ programs.

# Chapter 6: Object-Oriented Programming (Part 1)

## 6.1 Introduction to OOP

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around “objects” rather than functions and logic. An object is a data structure that contains both data (attributes or properties) and code (methods or functions). OOP was developed to make complex software systems more manageable, maintainable, and reusable.

### Core Concepts of Object-Oriented Programming

#### 1. Classes and Objects

A **class** is a blueprint or template that defines the characteristics and behaviors of a specific type of object. An **object** is an instance of a class, representing a concrete entity with its own unique state.

// Class definition  
class Car {  
 // Attributes and methods will go here  
};  
  
// Creating objects  
Car sedan; // First object of type Car  
Car suv; // Second object of type Car

#### 2. Encapsulation

**Encapsulation** is the bundling of data (attributes) and the methods that operate on that data into a single unit (class), and restricting direct access to some of the object’s components. This helps to:

* Hide implementation details
* Control access to data
* Protect data integrity
* Reduce system complexity

class BankAccount {  
private:  
 double balance; // Hidden from outside access  
   
public:  
 void deposit(double amount) {  
 if (amount > 0) {  
 balance += amount;  
 }  
 }  
   
 bool withdraw(double amount) {  
 if (amount <= balance && amount > 0) {  
 balance -= amount;  
 return true;  
 }  
 return false;  
 }  
   
 double getBalance() {  
 return balance;  
 }  
};

#### 3. Inheritance

**Inheritance** allows a class (subclass/derived class) to inherit attributes and methods from another class (superclass/base class). This promotes code reuse and establishes an “is-a” relationship between classes.

class Vehicle {  
public:  
 void startEngine() {  
 // Code to start engine  
 }  
};  
  
class Car : public Vehicle { // Car inherits from Vehicle  
public:  
 void accelerate() {  
 // Car-specific acceleration code  
 }  
};

#### 4. Polymorphism

**Polymorphism** allows objects of different classes to be treated as objects of a common superclass. It enables one interface to be used for a general class of actions, with the specific action determined by the type of data.

class Shape {  
public:  
 virtual double area() = 0; // Pure virtual function  
};  
  
class Circle : public Shape {  
private:  
 double radius;  
   
public:  
 Circle(double r) : radius(r) {}  
   
 double area() override {  
 return 3.14159 \* radius \* radius;  
 }  
};  
  
class Rectangle : public Shape {  
private:  
 double width;  
 double height;  
   
public:  
 Rectangle(double w, double h) : width(w), height(h) {}  
   
 double area() override {  
 return width \* height;  
 }  
};

#### 5. Abstraction

**Abstraction** is the concept of exposing only the necessary features of an object while hiding the unnecessary details from the user. It helps manage complexity by hiding implementation details behind simple interfaces.

class Television {  
private:  
 // Complex internal components and mechanisms  
 bool powerState;  
 int channel;  
 int volume;  
 void initializeScreen() { /\* Complex initialization code \*/ }  
 void tuneChannel() { /\* Complex tuning logic \*/ }  
   
public:  
 // Simple interface  
 void turnOn() {   
 powerState = true;   
 initializeScreen();  
 }  
   
 void turnOff() {   
 powerState = false;   
 }  
   
 void setChannel(int newChannel) {   
 channel = newChannel;   
 tuneChannel();  
 }  
   
 void adjustVolume(int amount) {   
 volume += amount;   
 if (volume < 0) volume = 0;  
 if (volume > 100) volume = 100;  
 }  
};

### Benefits of OOP

1. **Modularity**: Systems can be divided into smaller, manageable parts
2. **Reusability**: Code can be reused through inheritance
3. **Flexibility and scalability**: Systems can be adapted and extended more easily
4. **Maintainability**: Changes to one part of the system have minimal impact on others
5. **Problem-solving approach**: Models real-world entities and their interactions

### OOP vs. Procedural Programming

| Aspect | OOP | Procedural |
| --- | --- | --- |
| Focus | Objects (data + behaviors) | Procedures (algorithms) |
| Data Access | Limited through encapsulation | Often globally accessible |
| Data & Function | Combined in objects | Separate |
| Code Reuse | Through inheritance | Through functions |
| Data Structure | Complex (objects) | Simple |
| Flexibility | High | Lower |
| Complexity | Higher initial complexity | Lower initial complexity |

### Practical Example: OOP in Action

Let’s see how a simple library system might be modeled using OOP principles:

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
class Book {  
private:  
 string title;  
 string author;  
 string isbn;  
 bool checkedOut;  
   
public:  
 // Constructor  
 Book(string t, string a, string i)  
 : title(t), author(a), isbn(i), checkedOut(false) {}  
   
 // Methods  
 bool isAvailable() const {  
 return !checkedOut;  
 }  
   
 void checkOut() {  
 if (!checkedOut) {  
 checkedOut = true;  
 cout << title << " has been checked out." << endl;  
 } else {  
 cout << title << " is already checked out." << endl;  
 }  
 }  
   
 void returnBook() {  
 if (checkedOut) {  
 checkedOut = false;  
 cout << title << " has been returned." << endl;  
 } else {  
 cout << title << " is already in the library." << endl;  
 }  
 }  
   
 string getTitle() const { return title; }  
 string getAuthor() const { return author; }  
 string getISBN() const { return isbn; }  
};  
  
class Library {  
private:  
 vector<Book> books;  
   
public:  
 void addBook(const Book& book) {  
 books.push\_back(book);  
 }  
   
 void displayAvailableBooks() const {  
 cout << "Available Books:" << endl;  
 bool foundAvailable = false;  
   
 for (const auto& book : books) {  
 if (book.isAvailable()) {  
 cout << book.getTitle() << " by " << book.getAuthor() << endl;  
 foundAvailable = true;  
 }  
 }  
   
 if (!foundAvailable) {  
 cout << "No books are currently available." << endl;  
 }  
 }  
   
 Book\* findBookByTitle(const string& title) {  
 for (auto& book : books) {  
 if (book.getTitle() == title) {  
 return &book;  
 }  
 }  
 return nullptr;  
 }  
};  
  
int main() {  
 // Create a library and add books  
 Library myLibrary;  
   
 myLibrary.addBook(Book("1984", "George Orwell", "9780451524935"));  
 myLibrary.addBook(Book("To Kill a Mockingbird", "Harper Lee", "9780060935467"));  
 myLibrary.addBook(Book("The Great Gatsby", "F. Scott Fitzgerald", "9780743273565"));  
   
 // Display available books  
 myLibrary.displayAvailableBooks();  
   
 // Check out a book  
 Book\* book = myLibrary.findBookByTitle("1984");  
 if (book != nullptr) {  
 book->checkOut();  
 }  
   
 // Try to check out the same book again  
 if (book != nullptr) {  
 book->checkOut();  
 }  
   
 // Display available books after checkout  
 myLibrary.displayAvailableBooks();  
   
 // Return the book  
 if (book != nullptr) {  
 book->returnBook();  
 }  
   
 // Display available books after return  
 myLibrary.displayAvailableBooks();  
   
 return 0;  
}

This example demonstrates many OOP principles: - **Classes and Objects**: Book and Library classes with their respective objects - **Encapsulation**: Private data members with public methods to access/modify them - **Abstraction**: Complex operations simplified through methods like checkOut() and returnBook() - **Code organization**: Related data and behavior bundled together

## 6.2 Classes and Objects

Classes and objects are the fundamental building blocks of object-oriented programming in C++. Let’s dive deeper into how they work.

### Defining Classes in C++

A class in C++ is defined using the class keyword, followed by the class name and a block containing the class members:

class ClassName {  
 // Members (data and functions)  
};

#### Class Members

A class can contain:

1. **Data members** (attributes): Variables that hold the state of the object
2. **Member functions** (methods): Functions that define the behavior of the object
3. **Access specifiers**: Keywords that control the visibility of class members (public, private, protected)

class Person {  
private:  
 // Data members (attributes)  
 std::string name;  
 int age;  
 double height;  
   
public:  
 // Member functions (methods)  
 void setName(const std::string& n);  
 std::string getName() const;  
 void setAge(int a);  
 int getAge() const;  
 void setHeight(double h);  
 double getHeight() const;  
 void celebrateBirthday();  
};

#### Class vs Struct

In C++, a struct is essentially the same as a class, with one key difference: members in a struct are public by default, while members in a class are private by default.

struct Point {  
 int x; // public by default  
 int y; // public by default  
};  
  
class Circle {  
private: // private by default  
 double radius;  
 Point center;  
};

### Implementing Member Functions

Member functions can be defined: 1. Inside the class definition (implicitly inline) 2. Outside the class definition using the scope resolution operator (::):

// Inside class definition  
class Rectangle {  
private:  
 double width;  
 double height;  
   
public:  
 // Method defined inside class  
 double area() {  
 return width \* height;  
 }  
   
 void setDimensions(double w, double h);  
 double perimeter();  
};  
  
// Outside class definition  
void Rectangle::setDimensions(double w, double h) {  
 width = w;  
 height = h;  
}  
  
double Rectangle::perimeter() {  
 return 2 \* (width + height);  
}

### Creating and Using Objects

Once a class is defined, objects (instances) can be created in several ways:

#include <iostream>  
using namespace std;  
  
class Rectangle {  
private:  
 double width;  
 double height;  
   
public:  
 // Constructors will be covered in section 6.3  
 Rectangle() : width(0), height(0) {}  
 Rectangle(double w, double h) : width(w), height(h) {}  
   
 double area() { return width \* height; }  
 double perimeter() { return 2 \* (width + height); }  
 void setDimensions(double w, double h) { width = w; height = h; }  
 double getWidth() { return width; }  
 double getHeight() { return height; }  
};  
  
int main() {  
 // Method 1: Default initialization  
 Rectangle rect1;  
 rect1.setDimensions(5, 3);  
   
 // Method 2: Parameterized constructor  
 Rectangle rect2(4, 6);  
   
 // Method 3: Uniform initialization (C++11)  
 Rectangle rect3{7, 2};  
   
 // Method 4: Dynamic allocation  
 Rectangle\* rect4 = new Rectangle(8, 9);  
   
 // Accessing members with dot operator  
 cout << "Rectangle 1 area: " << rect1.area() << endl;  
 cout << "Rectangle 2 perimeter: " << rect2.perimeter() << endl;  
   
 // Accessing members with arrow operator (for pointer)  
 cout << "Rectangle 4 area: " << rect4->area() << endl;  
   
 // Don't forget to delete dynamically allocated objects  
 delete rect4;  
   
 return 0;  
}

### The this Pointer

Every object in C++ has access to its own address through the this pointer. It is an implicit parameter to all member functions and can be used to:

1. Disambiguate between member variables and parameters with the same name
2. Return the current object to enable method chaining
3. Pass the current object to another function

class Counter {  
private:  
 int count;  
   
public:  
 Counter(int count) : count(count) {} // Parameter has same name as member  
   
 void increment() {  
 count++;  
 }  
   
 // Using this to disambiguate  
 void reset(int count) {  
 this->count = count; // this->count refers to the member variable  
 }  
   
 // Using this for method chaining  
 Counter& add(int value) {  
 count += value;  
 return \*this; // Return reference to current object  
 }  
   
 int getCount() const {  
 return count;  
 }  
};  
  
int main() {  
 Counter c(5);  
 c.increment();  
 cout << c.getCount() << endl; // 6  
   
 c.reset(10);  
 cout << c.getCount() << endl; // 10  
   
 // Method chaining  
 c.add(5).add(3).add(2);  
 cout << c.getCount() << endl; // 20  
   
 return 0;  
}

### Static Class Members

A class can have static members that belong to the class itself rather than any specific instance:

1. **Static data members**: Shared among all objects of the class
2. **Static member functions**: Can access only static data members and don’t have a this pointer

#include <iostream>  
using namespace std;  
  
class BankAccount {  
private:  
 static double interestRate; // Static data member  
 string accountHolder;  
 double balance;  
   
public:  
 BankAccount(const string& name, double initialBalance)  
 : accountHolder(name), balance(initialBalance) {}  
   
 void addInterest() {  
 balance += balance \* interestRate;  
 }  
   
 double getBalance() const {  
 return balance;  
 }  
   
 // Static member function  
 static void setInterestRate(double rate) {  
 // Can only access static members  
 interestRate = rate;  
 // balance = 0; // ERROR: Can't access non-static members  
 }  
   
 static double getInterestRate() {  
 return interestRate;  
 }  
};  
  
// Static members must be defined outside the class  
double BankAccount::interestRate = 0.05; // 5% default interest rate  
  
int main() {  
 // Access static member through class name  
 cout << "Current interest rate: " << BankAccount::getInterestRate() << endl;  
   
 // Create bank accounts  
 BankAccount acc1("Alice", 1000);  
 BankAccount acc2("Bob", 2000);  
   
 // Apply interest  
 acc1.addInterest();  
 acc2.addInterest();  
   
 cout << "Alice's balance: " << acc1.getBalance() << endl; // 1050  
 cout << "Bob's balance: " << acc2.getBalance() << endl; // 2100  
   
 // Change interest rate for all accounts  
 BankAccount::setInterestRate(0.06); // 6% interest  
   
 // Apply new interest rate  
 acc1.addInterest();  
 acc2.addInterest();  
   
 cout << "Alice's new balance: " << acc1.getBalance() << endl; // 1113  
 cout << "Bob's new balance: " << acc2.getBalance() << endl; // 2226  
   
 return 0;  
}

### Const Member Functions

Member functions that don’t modify the object’s state should be declared as const. This enables them to be called on const objects and communicates intent:

class Circle {  
private:  
 double radius;  
   
public:  
 Circle(double r) : radius(r) {}  
   
 // const member function - doesn't modify the object  
 double area() const {  
 return 3.14159 \* radius \* radius;  
 }  
   
 // non-const member function - modifies the object  
 void setRadius(double r) {  
 radius = r;  
 }  
};  
  
int main() {  
 const Circle unitCircle(1.0);  
   
 // OK: area() is a const member function  
 cout << "Area: " << unitCircle.area() << endl;  
   
 // ERROR: setRadius() is not a const member function  
 // unitCircle.setRadius(2.0);  
   
 return 0;  
}

### Nested Classes

A class can contain the definition of another class:

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
class University {  
public:  
 // Nested class  
 class Student {  
 private:  
 string name;  
 int id;  
   
 public:  
 Student(const string& n, int i) : name(n), id(i) {}  
   
 string getName() const { return name; }  
 int getId() const { return id; }  
 };  
   
private:  
 string name;  
 vector<Student> students;  
   
public:  
 University(const string& n) : name(n) {}  
   
 void addStudent(const Student& s) {  
 students.push\_back(s);  
 }  
   
 void printStudents() const {  
 cout << "Students at " << name << ":" << endl;  
 for (const auto& student : students) {  
 cout << "ID: " << student.getId()   
 << ", Name: " << student.getName() << endl;  
 }  
 }  
};  
  
int main() {  
 University mit("MIT");  
   
 // Create student objects  
 University::Student s1("John", 12345);  
 University::Student s2("Lisa", 67890);  
   
 // Add students to university  
 mit.addStudent(s1);  
 mit.addStudent(s2);  
   
 // Print student list  
 mit.printStudents();  
   
 return 0;  
}

### Best Practices for Classes and Objects

1. **Keep classes focused**: A class should represent a single concept and have a clear responsibility
2. **Encapsulate data**: Make data members private and provide access through methods when needed
3. **Use meaningful names**: Choose descriptive names for classes, methods, and attributes
4. **Design for users**: Design the public interface with users of the class in mind
5. **Design for extension**: Consider how the class might be extended or inherited
6. **Follow the Single Responsibility Principle**: A class should have only one reason to change
7. **Minimize method size**: Keep methods small and focused on a single task
8. **Use const correctly**: Mark methods that don’t modify the object as const
9. **Limit dependencies**: Minimize dependencies between classes
10. **Document public interfaces**: Use comments to explain how to use the class

## 6.3 Constructors and Destructors

Constructors and destructors are special member functions that handle object initialization and cleanup in C++. They play a crucial role in implementing RAII (Resource Acquisition Is Initialization), one of the most important idioms in C++.

### Constructors

A constructor is a special member function that is automatically called when an object is created. Its purpose is to initialize the object’s data members and acquire necessary resources.

#### Types of Constructors

1. **Default Constructor**: Takes no parameters or has default values for all parameters
2. **Parameterized Constructor**: Takes one or more parameters
3. **Copy Constructor**: Creates a new object as a copy of an existing object
4. **Move Constructor (C++11)**: Creates a new object by transferring resources from a temporary object
5. **Delegating Constructor (C++11)**: Calls another constructor in the same class
6. **Converting Constructor**: Takes a single parameter and can be used for implicit type conversion

Let’s examine each type:

#### Default Constructor

class Box {  
private:  
 double length;  
 double width;  
 double height;  
   
public:  
 // Default constructor  
 Box() {  
 length = 1.0;  
 width = 1.0;  
 height = 1.0;  
 }  
   
 // Method to display dimensions  
 void displayDimensions() {  
 cout << "Length: " << length   
 << ", Width: " << width   
 << ", Height: " << height << endl;  
 }  
};  
  
int main() {  
 Box defaultBox; // Default constructor called  
 defaultBox.displayDimensions();  
   
 return 0;  
}

#### Parameterized Constructor

class Box {  
private:  
 double length;  
 double width;  
 double height;  
   
public:  
 // Default constructor  
 Box() {  
 length = 1.0;  
 width = 1.0;  
 height = 1.0;  
 }  
   
 // Parameterized constructor  
 Box(double l, double w, double h) {  
 length = l;  
 width = w;  
 height = h;  
 }  
   
 // Method to display dimensions  
 void displayDimensions() {  
 cout << "Length: " << length   
 << ", Width: " << width   
 << ", Height: " << height << endl;  
 }  
};  
  
int main() {  
 Box defaultBox; // Default constructor called  
 Box customBox(3.0, 2.0, 1.5); // Parameterized constructor called  
   
 defaultBox.displayDimensions();  
 customBox.displayDimensions();  
   
 return 0;  
}

#### Constructor with Default Parameters

class Box {  
private:  
 double length;  
 double width;  
 double height;  
   
public:  
 // Constructor with default parameters  
 Box(double l = 1.0, double w = 1.0, double h = 1.0) {  
 length = l;  
 width = w;  
 height = h;  
 }  
   
 // Method to display dimensions  
 void displayDimensions() {  
 cout << "Length: " << length   
 << ", Width: " << width   
 << ", Height: " << height << endl;  
 }  
};  
  
int main() {  
 Box box1; // Uses all default values: 1.0, 1.0, 1.0  
 Box box2(2.0); // Uses custom length, default width/height: 2.0, 1.0, 1.0  
 Box box3(3.0, 4.0); // Uses custom length/width, default height: 3.0, 4.0, 1.0  
 Box box4(5.0, 6.0, 7.0); // Uses all custom values: 5.0, 6.0, 7.0  
   
 box1.displayDimensions();  
 box2.displayDimensions();  
 box3.displayDimensions();  
 box4.displayDimensions();  
   
 return 0;  
}

#### Member Initialization List

A more efficient way to initialize members is using an initialization list:

class Person {  
private:  
 string name;  
 int age;  
 const string id; // const member must be initialized in the init list  
   
public:  
 // Constructor with initialization list  
 Person(const string& n, int a, const string& i)  
 : name(n), age(a), id(i) {  
 // Constructor body (can contain additional code)  
 cout << "Person created: " << name << endl;  
 }  
   
 void display() {  
 cout << "Name: " << name << ", Age: " << age << ", ID: " << id << endl;  
 }  
};  
  
int main() {  
 Person person("John Doe", 30, "A12345");  
 person.display();  
   
 return 0;  
}

Benefits of initialization lists: - More efficient than assigning values in the constructor body - Required for const members and reference members - Required for members that don’t have a default constructor - Initializes members in declaration order (not in list order)

#### Copy Constructor

A copy constructor creates a new object as a copy of an existing object:

class Array {  
private:  
 int\* data;  
 size\_t size;  
   
public:  
 // Regular constructor  
 Array(size\_t s) : size(s) {  
 data = new int[size](); // Allocate and zero-initialize  
 cout << "Regular constructor called" << endl;  
 }  
   
 // Copy constructor  
 Array(const Array& other) : size(other.size) {  
 data = new int[size];  
 // Deep copy the data  
 for (size\_t i = 0; i < size; i++) {  
 data[i] = other.data[i];  
 }  
 cout << "Copy constructor called" << endl;  
 }  
   
 // Destructor  
 ~Array() {  
 delete[] data;  
 cout << "Destructor called" << endl;  
 }  
   
 void setElement(size\_t index, int value) {  
 if (index < size) {  
 data[index] = value;  
 }  
 }  
   
 int getElement(size\_t index) const {  
 return (index < size) ? data[index] : 0;  
 }  
   
 size\_t getSize() const {  
 return size;  
 }  
};  
  
// Function that takes an Array by value (triggers copy)  
void processArray(Array arr) {  
 cout << "Processing array of size " << arr.getSize() << endl;  
}  
  
int main() {  
 Array arr1(5);  
 arr1.setElement(0, 10);  
 arr1.setElement(1, 20);  
   
 // Copy constructor called when creating arr2  
 Array arr2 = arr1; // Same as: Array arr2(arr1);  
   
 // Modify arr2 (doesn't affect arr1)  
 arr2.setElement(0, 30);  
   
 cout << "arr1[0] = " << arr1.getElement(0) << endl; // Still 10  
 cout << "arr2[0] = " << arr2.getElement(0) << endl; // 30  
   
 // Copy constructor called again when passing to function  
 processArray(arr1);  
   
 return 0;  
}

The copy constructor is called in these situations: - When initializing a new object from an existing one - When passing an object by value - When returning an object by value from a function

If you don’t provide a copy constructor, the compiler generates one that performs a member-wise copy (shallow copy), which is problematic for classes that manage resources like dynamically allocated memory.

#### Move Constructor (C++11)

A move constructor efficiently transfers resources from a temporary (rvalue) object to a newly created object:

#include <iostream>  
#include <utility> // For std::move  
using namespace std;  
  
class Array {  
private:  
 int\* data;  
 size\_t size;  
   
public:  
 // Regular constructor  
 Array(size\_t s) : size(s) {  
 data = new int[size]();  
 cout << "Regular constructor called" << endl;  
 }  
   
 // Copy constructor  
 Array(const Array& other) : size(other.size) {  
 data = new int[size];  
 for (size\_t i = 0; i < size; i++) {  
 data[i] = other.data[i];  
 }  
 cout << "Copy constructor called" << endl;  
 }  
   
 // Move constructor  
 Array(Array&& other) noexcept : data(other.data), size(other.size) {  
 // Transfer ownership (steal resources)  
 other.data = nullptr;  
 other.size = 0;  
 cout << "Move constructor called" << endl;  
 }  
   
 // Destructor  
 ~Array() {  
 delete[] data;  
 cout << "Destructor called for array of size " << size << endl;  
 }  
   
 // Other methods as before...  
 size\_t getSize() const { return size; }  
};  
  
// Function returning an Array by value  
Array createArray(size\_t size) {  
 Array temp(size);  
 return temp; // Return value optimization may apply  
}  
  
int main() {  
 cout << "Creating arr1:" << endl;  
 Array arr1(5);  
   
 cout << "\nCreating arr2 from arr1:" << endl;  
 Array arr2 = arr1; // Copy constructor  
   
 cout << "\nCreating arr3 from temporary:" << endl;  
 Array arr3 = createArray(3); // Move constructor may be used  
   
 cout << "\nCreating arr4 with std::move:" << endl;  
 Array arr4 = std::move(arr1); // Move constructor  
   
 cout << "\nProgram ending:" << endl;  
 return 0;  
}

The move constructor “steals” resources from a temporary object that’s about to be destroyed, avoiding unnecessary copying of resources. This is especially useful for classes that manage expensive resources like large memory buffers.

#### Delegating Constructors (C++11)

A constructor can call another constructor in the same class:

class Rectangle {  
private:  
 double width;  
 double height;  
 string color;  
   
public:  
 // Primary constructor  
 Rectangle(double w, double h, const string& c)  
 : width(w), height(h), color(c) {  
 cout << "Primary constructor called" << endl;  
 }  
   
 // Delegating constructors  
 Rectangle() : Rectangle(1.0, 1.0, "white") {  
 cout << "Default constructor called" << endl;  
 }  
   
 Rectangle(double size) : Rectangle(size, size, "white") {  
 cout << "Square constructor called" << endl;  
 }  
   
 Rectangle(double w, double h) : Rectangle(w, h, "white") {  
 cout << "Two-parameter constructor called" << endl;  
 }  
   
 void display() const {  
 cout << "Rectangle: " << width << " x " << height   
 << ", Color: " << color << endl;  
 }  
};  
  
int main() {  
 Rectangle r1; // Default constructor  
 Rectangle r2(5.0); // Square constructor  
 Rectangle r3(3.0, 4.0); // Two-parameter constructor  
 Rectangle r4(2.0, 3.0, "blue"); // Primary constructor  
   
 r1.display();  
 r2.display();  
 r3.display();  
 r4.display();  
   
 return 0;  
}

Delegating constructors avoid code duplication and ensure consistent initialization.

#### Explicit Constructors

By default, constructors that take a single parameter can be used for implicit type conversion. The explicit keyword prevents this:

class Integer {  
private:  
 int value;  
   
public:  
 // Implicit conversion constructor  
 Integer(int v) : value(v) {  
 cout << "Integer constructor called with " << v << endl;  
 }  
   
 int getValue() const { return value; }  
};  
  
class ExplicitInteger {  
private:  
 int value;  
   
public:  
 // Explicit constructor - prevents implicit conversion  
 explicit ExplicitInteger(int v) : value(v) {  
 cout << "ExplicitInteger constructor called with " << v << endl;  
 }  
   
 int getValue() const { return value; }  
};  
  
void processInteger(Integer i) {  
 cout << "Processing integer: " << i.getValue() << endl;  
}  
  
void processExplicit(ExplicitInteger e) {  
 cout << "Processing explicit integer: " << e.getValue() << endl;  
}  
  
int main() {  
 // Implicit conversion works  
 Integer i1 = 42; // Implicit conversion from int to Integer  
 processInteger(99); // Implicit conversion in function call  
   
 // Explicit conversion required  
 ExplicitInteger e1(42); // OK - direct initialization  
   
 // ExplicitInteger e2 = 42; // ERROR - no implicit conversion  
 processExplicit(ExplicitInteger(99)); // OK - explicit conversion  
   
 return 0;  
}

Using explicit for single-parameter constructors is generally a good practice as it prevents unintended implicit conversions.

### Destructors

A destructor is a special member function that is called when an object is destroyed. Its purpose is to release resources acquired by the object.

#### Basic Destructor Syntax

class Resource {  
private:  
 string name;  
 int\* data;  
   
public:  
 Resource(const string& n, int size) : name(n) {  
 cout << "Constructor: Creating " << name << endl;  
 data = new int[size];  
 }  
   
 ~Resource() {  
 cout << "Destructor: Cleaning up " << name << endl;  
 delete[] data;  
 }  
};  
  
void useResource() {  
 Resource r1("LocalResource", 100);  
 // r1 automatically destroyed when function exits  
}  
  
int main() {  
 cout << "Program starting" << endl;  
   
 {  
 Resource r2("BlockResource", 200);  
 // r2 destroyed at end of block  
 }  
   
 useResource();  
   
 Resource\* r3 = new Resource("DynamicResource", 300);  
 // r3 not automatically destroyed  
   
 delete r3; // Manual destruction needed for dynamically allocated objects  
   
 cout << "Program ending" << endl;  
 return 0;  
}

#### Key Points about Destructors

1. **Naming**: A destructor’s name is the class name preceded by a tilde (~)
2. **No Parameters**: Destructors don’t take parameters
3. **No Return Type**: Destructors don’t return a value (not even void)
4. **No Overloading**: A class can have only one destructor
5. **Automatic Invocation**: Destructors are called automatically when objects go out of scope
6. **Order**: Destructors are called in the reverse order of constructors
7. **Virtual Destructors**: Base class destructors should be virtual when inheritance is used (covered later)

#### When You Need to Write a Destructor

You need to write a custom destructor when your class manages resources that aren’t automatically cleaned up:

1. **Dynamically allocated memory** (new/delete)
2. **File handles**
3. **Network connections**
4. **Database connections**
5. **Other system resources**

If your class doesn’t manage any resources directly (it only contains built-in types or objects that handle their own cleanup), you typically don’t need to write a destructor.

#### The Rule of Three/Five/Zero

When you need to manage resources explicitly:

1. **Rule of Three**: If a class needs a custom destructor, copy constructor, or copy assignment operator, it likely needs all three
2. **Rule of Five (C++11)**: Adds move constructor and move assignment operator to the list
3. **Rule of Zero**: Design classes to avoid the need for custom resource management functions

Here’s a class implementing the Rule of Five:

#include <iostream>  
#include <cstring> // For strlen, strcpy  
using namespace std;  
  
class String {  
private:  
 char\* data;  
   
 // Helper to calculate string length including null terminator  
 static size\_t length(const char\* s) {  
 return s ? strlen(s) + 1 : 1;  
 }  
   
public:  
 // Constructor  
 String(const char\* s = "") {  
 size\_t len = length(s);  
 data = new char[len];  
 strcpy(data, s ? s : "");  
 cout << "Constructor: " << data << endl;  
 }  
   
 // Destructor  
 ~String() {  
 cout << "Destructor: " << data << endl;  
 delete[] data;  
 }  
   
 // Copy constructor  
 String(const String& other) {  
 size\_t len = length(other.data);  
 data = new char[len];  
 strcpy(data, other.data);  
 cout << "Copy constructor: " << data << endl;  
 }  
   
 // Copy assignment operator  
 String& operator=(const String& other) {  
 cout << "Copy assignment operator" << endl;  
 if (this != &other) { // Prevent self-assignment  
 char\* newData = new char[length(other.data)];  
 strcpy(newData, other.data);  
 delete[] data; // Delete old data  
 data = newData;  
 }  
 return \*this;  
 }  
   
 // Move constructor (C++11)  
 String(String&& other) noexcept {  
 cout << "Move constructor" << endl;  
 data = other.data; // Take ownership  
 other.data = nullptr; // Prevent destruction of data  
 }  
   
 // Move assignment operator (C++11)  
 String& operator=(String&& other) noexcept {  
 cout << "Move assignment operator" << endl;  
 if (this != &other) {  
 delete[] data; // Delete our data  
 data = other.data; // Take ownership  
 other.data = nullptr; // Prevent destruction of data  
 }  
 return \*this;  
 }  
   
 // Accessor  
 const char\* c\_str() const {  
 return data;  
 }  
};  
  
int main() {  
 String s1("Hello"); // Constructor  
 String s2 = s1; // Copy constructor  
 String s3; // Constructor with default value  
 s3 = s1; // Copy assignment operator  
   
 String s4 = std::move(s1); // Move constructor  
 String s5; // Constructor with default value  
 s5 = std::move(s2); // Move assignment operator  
   
 cout << "s3: " << s3.c\_str() << endl;  
 cout << "s4: " << s4.c\_str() << endl;  
 cout << "s5: " << s5.c\_str() << endl;  
 // s1 and s2 are in valid but unspecified states  
   
 return 0;  
}

### Best Practices for Constructors and Destructors

1. **Initialize all members**: Make sure all member variables are initialized in constructors
2. **Use member initialization lists**: More efficient than assignments in the constructor body
3. **Use delegating constructors**: Avoid code duplication in constructors
4. **Make single-parameter constructors explicit**: Prevent unintended implicit conversions
5. **Follow the Rule of Three/Five/Zero**: If you need one resource-managing function, you likely need all of them
6. **Make base class destructors virtual**: When using inheritance
7. **Keep destructors exception-safe**: Destructors should never throw exceptions
8. **Use RAII**: Acquire resources in constructors and release them in destructors
9. **Consider smart pointers**: Often better than manual memory management
10. **Don’t call virtual functions in constructors or destructors**: The behavior can be surprising

## 6.4 Access Specifiers (Public, Private, Protected)

Access specifiers control the visibility and accessibility of class members. C++ provides three access specifiers:

1. **public**: Members are accessible from anywhere
2. **private**: Members are accessible only from within the class
3. **protected**: Members are accessible from within the class and derived classes

### Public Access Specifier

Public members form the interface of the class. They can be accessed from anywhere the class is visible:

class Account {  
public: // Public access specifier  
 string accountNumber; // Public data member  
   
 void deposit(double amount) { // Public method  
 balance += amount;  
 }  
   
 double getBalance() { // Public method  
 return balance;  
 }  
   
private:  
 double balance = 0.0; // Private data member  
};  
  
int main() {  
 Account acc;  
 acc.accountNumber = "123456"; // OK - public member  
 acc.deposit(1000); // OK - public method  
 cout << acc.getBalance(); // OK - public method  
 // acc.balance = 5000; // ERROR - private member  
   
 return 0;  
}

### Private Access Specifier

Private members are accessible only from within the class itself. They cannot be accessed from outside the class, including derived classes:

class BankAccount {  
private: // Private access specifier  
 double balance;  
 string accountNumber;  
 double interestRate;  
   
 // Private helper method  
 bool isValidAmount(double amount) {  
 return amount > 0;  
 }  
   
public:  
 BankAccount(const string& accNum, double initialBalance)   
 : accountNumber(accNum), balance(initialBalance), interestRate(0.05) {}  
   
 bool withdraw(double amount) {  
 // Can access private members and methods  
 if (!isValidAmount(amount) || amount > balance) {  
 return false;  
 }  
 balance -= amount;  
 return true;  
 }  
   
 void deposit(double amount) {  
 if (isValidAmount(amount)) {  
 balance += amount;  
 }  
 }  
   
 double getBalance() const {  
 return balance;  
 }  
   
 string getAccountNumber() const {  
 return accountNumber;  
 }  
};  
  
int main() {  
 BankAccount acc("12345", 1000);  
   
 // acc.balance = 2000; // ERROR - private member  
 // acc.interestRate = 0.1; // ERROR - private member  
 // bool valid = acc.isValidAmount(500); // ERROR - private method  
   
 acc.deposit(500); // OK - public method  
 acc.withdraw(200); // OK - public method  
 cout << acc.getBalance() << endl; // OK - public method  
   
 return 0;  
}

### Protected Access Specifier

Protected members are accessible from within the class and from derived classes but not from outside:

class Shape {  
protected: // Protected access specifier  
 double x, y; // Position  
 string color;  
   
 // Protected helper method  
 void validate() {  
 if (color.empty()) {  
 color = "black";  
 }  
 }  
   
public:  
 Shape(double x, double y, const string& c)  
 : x(x), y(y), color(c) {  
 validate();  
 }  
   
 void move(double newX, double newY) {  
 x = newX;  
 y = newY;  
 }  
   
 string getColor() const {  
 return color;  
 }  
};  
  
class Circle : public Shape {  
private:  
 double radius;  
   
public:  
 Circle(double x, double y, double r, const string& c)  
 : Shape(x, y, c), radius(r) {}  
   
 void setRadius(double r) {  
 radius = r;  
 }  
   
 double area() const {  
 return 3.14159 \* radius \* radius;  
 }  
   
 // Can access protected members of base class  
 void changeColor(const string& newColor) {  
 color = newColor; // OK - protected member  
 validate(); // OK - protected method  
 }  
   
 void display() const {  
 cout << "Circle at (" << x << ", " << y << "), " // OK - protected  
 << "radius " << radius << ", color " << color << endl; // OK - protected  
 }  
};  
  
int main() {  
 Shape shape(10, 20, "red");  
 Circle circle(15, 25, 5, "blue");  
   
 // shape.x = 30; // ERROR - protected member  
 // shape.color = "green"; // ERROR - protected member  
 // shape.validate(); // ERROR - protected method  
   
 shape.move(30, 40); // OK - public method  
 circle.setRadius(10); // OK - public method  
 circle.changeColor("green"); // OK - public method  
 circle.display(); // OK - public method  
   
 return 0;  
}

### Friend Functions and Classes

C++ allows you to declare specific functions or classes as “friends,” granting them access to private and protected members:

class BankAccount {  
private:  
 string accountNumber;  
 double balance;  
   
public:  
 BankAccount(const string& accNum, double initialBalance)  
 : accountNumber(accNum), balance(initialBalance) {}  
   
 // Declare friends  
 friend void displayAccountDetails(const BankAccount& acc);  
 friend class BankSystem;  
   
 // Regular public methods  
 void deposit(double amount) {  
 if (amount > 0) {  
 balance += amount;  
 }  
 }  
};  
  
// Friend function can access private members  
void displayAccountDetails(const BankAccount& acc) {  
 cout << "Account: " << acc.accountNumber // OK - can access private member  
 << ", Balance: $" << acc.balance << endl; // OK - can access private member  
}  
  
// Friend class can access private members  
class BankSystem {  
public:  
 void transferFunds(BankAccount& from, BankAccount& to, double amount) {  
 if (amount > 0 && from.balance >= amount) {  
 from.balance -= amount; // OK - can access private member  
 to.balance += amount; // OK - can access private member  
 cout << "Transferred $" << amount << " from "   
 << from.accountNumber << " to " << to.accountNumber << endl;  
 } else {  
 cout << "Transfer failed" << endl;  
 }  
 }  
   
 void auditAccounts(const vector<BankAccount>& accounts) {  
 double totalFunds = 0;  
 for (const auto& acc : accounts) {  
 totalFunds += acc.balance; // OK - can access private member  
 }  
 cout << "Total funds: $" << totalFunds << endl;  
 }  
};  
  
int main() {  
 BankAccount acc1("12345", 1000);  
 BankAccount acc2("67890", 500);  
   
 displayAccountDetails(acc1); // Friend function  
   
 BankSystem system;  
 system.transferFunds(acc1, acc2, 300); // Friend class  
   
 vector<BankAccount> accounts = {acc1, acc2};  
 system.auditAccounts(accounts); // Friend class  
   
 return 0;  
}

#### Important Points about Friends

1. Friendship is not symmetric (if A is a friend of B, B is not automatically a friend of A)
2. Friendship is not transitive (if A is a friend of B, and B is a friend of C, A is not automatically a friend of C)
3. Friendship is not inherited (if a base class has a friend, derived classes don’t inherit that friendship)
4. Friends violate encapsulation to some extent, so use them judiciously

### Default Access in Classes and Structs

The default access specifier is different in classes and structs:

class MyClass {  
 int x; // Private by default in classes  
public:  
 int y; // Explicitly public  
};  
  
struct MyStruct {  
 int x; // Public by default in structs  
private:  
 int y; // Explicitly private  
};  
  
int main() {  
 MyClass c;  
 // c.x = 10; // ERROR - private by default  
 c.y = 20; // OK - public  
   
 MyStruct s;  
 s.x = 10; // OK - public by default  
 // s.y = 20; // ERROR - explicitly private  
   
 return 0;  
}

### Access Control and Inheritance

Access specifiers also control how members are inherited:

class Base {  
public:  
 int publicVar;  
protected:  
 int protectedVar;  
private:  
 int privateVar;  
};  
  
class PublicDerived : public Base {  
 // publicVar remains public  
 // protectedVar remains protected  
 // privateVar is not accessible  
   
 void accessTest() {  
 publicVar = 1; // OK - public in base  
 protectedVar = 2; // OK - protected in base  
 // privateVar = 3; // ERROR - private in base, not accessible  
 }  
};  
  
class ProtectedDerived : protected Base {  
 // publicVar becomes protected  
 // protectedVar remains protected  
 // privateVar is not accessible  
   
 void accessTest() {  
 publicVar = 1; // OK - protected in derived  
 protectedVar = 2; // OK - protected in derived  
 // privateVar = 3; // ERROR - private in base, not accessible  
 }  
};  
  
class PrivateDerived : private Base {  
 // publicVar becomes private  
 // protectedVar becomes private  
 // privateVar is not accessible  
   
 void accessTest() {  
 publicVar = 1; // OK - private in derived  
 protectedVar = 2; // OK - private in derived  
 // privateVar = 3; // ERROR - private in base, not accessible  
 }  
};  
  
int main() {  
 PublicDerived d1;  
 d1.publicVar = 1; // OK - public in derived  
 // d1.protectedVar = 2; // ERROR - protected in derived  
 // d1.privateVar = 3; // ERROR - not accessible  
   
 ProtectedDerived d2;  
 // d2.publicVar = 1; // ERROR - protected in derived  
 // d2.protectedVar = 2; // ERROR - protected in derived  
 // d2.privateVar = 3; // ERROR - not accessible  
   
 PrivateDerived d3;  
 // d3.publicVar = 1; // ERROR - private in derived  
 // d3.protectedVar = 2; // ERROR - private in derived  
 // d3.privateVar = 3; // ERROR - not accessible  
   
 return 0;  
}

### Best Practices for Access Control

1. **Public Interface**: Expose only what’s necessary in the public interface
2. **Private Implementation**: Keep implementation details private
3. **Protected for Inheritance**: Use protected for members that derived classes need to access
4. **Minimize Friend Declarations**: Use friends sparingly
5. **Accessors and Mutators**: Provide controlled access to private data through public methods
6. **Class Invariants**: Use private access to maintain class invariants (conditions that must always be true)
7. **Favor Encapsulation**: Restrict access to implementation details

// Example of good access control design  
class Temperature {  
private:  
 double celsius;  
   
 // Private helper method  
 double validateTemperature(double temp) {  
 const double absoluteZero = -273.15;  
 return (temp < absoluteZero) ? absoluteZero : temp;  
 }  
   
public:  
 // Constructor ensures valid initialization  
 Temperature(double c) : celsius(validateTemperature(c)) {}  
   
 // Public accessors  
 double getCelsius() const { return celsius; }  
 double getFahrenheit() const { return celsius \* 9.0/5.0 + 32; }  
 double getKelvin() const { return celsius + 273.15; }  
   
 // Public mutators with validation  
 void setCelsius(double c) { celsius = validateTemperature(c); }  
   
 void setFahrenheit(double f) {  
 setCelsius((f - 32) \* 5.0/9.0);  
 }  
   
 void setKelvin(double k) {  
 setCelsius(k - 273.15);  
 }  
};  
  
int main() {  
 Temperature temp(25.0);  
 cout << "Celsius: " << temp.getCelsius() << endl;  
 cout << "Fahrenheit: " << temp.getFahrenheit() << endl;  
 cout << "Kelvin: " << temp.getKelvin() << endl;  
   
 temp.setFahrenheit(98.6);  
 cout << "Body temperature in C: " << temp.getCelsius() << endl;  
   
 // Invalid temperature is corrected automatically  
 temp.setCelsius(-300);  
 cout << "After setting to -300C, actual value: " << temp.getCelsius() << endl;  
   
 return 0;  
}

In this example: - Private data (celsius) is protected from invalid modifications - Public methods provide a controlled interface for using the class - Class invariants (temperature cannot be below absolute zero) are enforced - Implementation details (validation logic) are hidden

Access control is a fundamental aspect of encapsulation in object-oriented programming. By carefully designing what is accessible and what is hidden, you create classes that are easier to use correctly and harder to use incorrectly.

# Chapter 6: Object-Oriented Programming (Part 2)

## 6.5 Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows you to create new classes (derived/child classes) from existing ones (base/parent classes). The derived class inherits attributes and behaviors from the base class, and can also extend or modify them.

### Core Concepts of Inheritance

1. **Base Class (Parent Class)**: The original class that shares its features
2. **Derived Class (Child Class)**: The new class that inherits features
3. **“is-a” Relationship**: Inheritance establishes an “is-a” relationship between classes
4. **Code Reuse**: Inheritance promotes code reuse and organization
5. **Specialization**: Derived classes can specialize behavior of base classes

### Basic Syntax

class BaseClass {  
 // Base class members  
};  
  
class DerivedClass : access-specifier BaseClass {  
 // Derived class members  
 // Inherits members from BaseClass  
};

The access-specifier can be: - public: Public and protected members of the base class remain public and protected in the derived class - protected: Public and protected members of the base class become protected in the derived class - private: Public and protected members of the base class become private in the derived class

### 6.5.1 Single Inheritance

Single inheritance is the simplest form where a class inherits from only one base class.

#include <iostream>  
#include <string>  
using namespace std;  
  
// Base class  
class Vehicle {  
private:  
 string make;  
 string model;  
 int year;  
   
protected:  
 double price;  
   
public:  
 Vehicle(const string& make, const string& model, int year, double price)  
 : make(make), model(model), year(year), price(price) {  
 cout << "Vehicle constructor called" << endl;  
 }  
   
 ~Vehicle() {  
 cout << "Vehicle destructor called" << endl;  
 }  
   
 void displayInfo() const {  
 cout << year << " " << make << " " << model   
 << ", Price: $" << price << endl;  
 }  
   
 void start() const {  
 cout << "Vehicle started" << endl;  
 }  
   
 void stop() const {  
 cout << "Vehicle stopped" << endl;  
 }  
   
 string getMake() const { return make; }  
 string getModel() const { return model; }  
 int getYear() const { return year; }  
};  
  
// Derived class  
class Car : public Vehicle {  
private:  
 int numDoors;  
 double engineSize; // in liters  
   
public:  
 Car(const string& make, const string& model, int year, double price,  
 int doors, double engine)  
 : Vehicle(make, model, year, price), // Call base class constructor  
 numDoors(doors), engineSize(engine) {  
 cout << "Car constructor called" << endl;  
 }  
   
 ~Car() {  
 cout << "Car destructor called" << endl;  
 }  
   
 void displayCarInfo() const {  
 // Access base class public methods  
 displayInfo();  
 cout << "Doors: " << numDoors << ", Engine: " << engineSize << "L" << endl;  
 }  
   
 void applyDiscount(double percentage) {  
 // Can access protected members from base class  
 double discountAmount = price \* percentage / 100.0;  
 price -= discountAmount;  
 cout << "Discount of " << percentage << "% applied. New price: $" << price << endl;  
 }  
};  
  
int main() {  
 Car myCar("Toyota", "Corolla", 2023, 25000, 4, 1.8);  
   
 myCar.displayCarInfo();  
   
 // Accessing base class methods  
 myCar.start();  
 myCar.stop();  
   
 // Using derived class method  
 myCar.applyDiscount(10);  
   
 return 0;  
}

#### Key Aspects of Single Inheritance

1. **Constructor and Destructor Order**
   * Base class constructor is called before derived class constructor
   * Derived class destructor is called before base class destructor
   * This ensures proper initialization and cleanup
2. **Access Control**
   * Public inheritance: “is-a” relationship (Car is-a Vehicle)
   * Private members of base class are never directly accessible in derived class
   * Public members of base class become public in derived class (with public inheritance)
   * Protected members of base class are accessible in derived class
3. **Method Overriding**
   * Derived class can override (redefine) methods from the base class
   * The most specific version of the method is called

#include <iostream>  
using namespace std;  
  
class Shape {  
public:  
 void draw() const {  
 cout << "Drawing a shape" << endl;  
 }  
   
 double area() const {  
 cout << "Shape::area() called" << endl;  
 return 0.0;  
 }  
};  
  
class Circle : public Shape {  
private:  
 double radius;  
   
public:  
 Circle(double r) : radius(r) {}  
   
 // Override the base class method  
 double area() const {  
 cout << "Circle::area() called" << endl;  
 return 3.14159 \* radius \* radius;  
 }  
   
 // Add new functionality  
 double circumference() const {  
 return 2 \* 3.14159 \* radius;  
 }  
};  
  
int main() {  
 Shape genericShape;  
 Circle circle(5.0);  
   
 genericShape.draw(); // Drawing a shape  
 circle.draw(); // Drawing a shape (inherited from Shape)  
   
 cout << "Generic shape area: " << genericShape.area() << endl;  
 cout << "Circle area: " << circle.area() << endl;  
 cout << "Circle circumference: " << circle.circumference() << endl;  
   
 return 0;  
}

### 6.5.2 Multiple Inheritance

Multiple inheritance allows a class to inherit from more than one base class.

#include <iostream>  
#include <string>  
using namespace std;  
  
class Person {  
private:  
 string name;  
 int age;  
   
public:  
 Person(const string& n, int a) : name(n), age(a) {  
 cout << "Person constructor called" << endl;  
 }  
   
 ~Person() {  
 cout << "Person destructor called" << endl;  
 }  
   
 void displayInfo() const {  
 cout << "Name: " << name << ", Age: " << age << endl;  
 }  
   
 void sleep() const {  
 cout << name << " is sleeping" << endl;  
 }  
};  
  
class Employee {  
private:  
 int employeeId;  
 double salary;  
   
public:  
 Employee(int id, double s) : employeeId(id), salary(s) {  
 cout << "Employee constructor called" << endl;  
 }  
   
 ~Employee() {  
 cout << "Employee destructor called" << endl;  
 }  
   
 void displayEmployeeInfo() const {  
 cout << "Employee ID: " << employeeId << ", Salary: $" << salary << endl;  
 }  
   
 void work() const {  
 cout << "Employee is working" << endl;  
 }  
};  
  
// Multiple inheritance  
class Manager : public Person, public Employee {  
private:  
 string department;  
   
public:  
 Manager(const string& name, int age, int id, double salary, const string& dept)  
 : Person(name, age), Employee(id, salary), department(dept) {  
 cout << "Manager constructor called" << endl;  
 }  
   
 ~Manager() {  
 cout << "Manager destructor called" << endl;  
 }  
   
 void displayManagerInfo() const {  
 displayInfo(); // From Person  
 displayEmployeeInfo(); // From Employee  
 cout << "Department: " << department << endl;  
 }  
   
 void manage() const {  
 cout << "Manager is managing the " << department << " department" << endl;  
 }  
};  
  
int main() {  
 Manager manager("John Smith", 35, 12345, 75000, "Engineering");  
   
 // Access methods from both base classes  
 manager.displayManagerInfo();  
 manager.sleep(); // From Person  
 manager.work(); // From Employee  
 manager.manage(); // From Manager  
   
 return 0;  
}

#### Challenges of Multiple Inheritance: The Diamond Problem

One of the main issues with multiple inheritance is the “Diamond Problem,” which occurs when a class inherits from two classes that have a common base class.

#include <iostream>  
using namespace std;  
  
class PoweredDevice {  
private:  
 int powerUsage;  
   
public:  
 PoweredDevice(int power) : powerUsage(power) {  
 cout << "PoweredDevice constructor called with power: " << power << endl;  
 }  
   
 void usePower() const {  
 cout << "Using " << powerUsage << " watts of power" << endl;  
 }  
};  
  
class Scanner : public PoweredDevice {  
public:  
 Scanner(int power) : PoweredDevice(power) {  
 cout << "Scanner constructor called" << endl;  
 }  
   
 void scan() const {  
 cout << "Scanning document" << endl;  
 usePower();  
 }  
};  
  
class Printer : public PoweredDevice {  
public:  
 Printer(int power) : PoweredDevice(power) {  
 cout << "Printer constructor called" << endl;  
 }  
   
 void print() const {  
 cout << "Printing document" << endl;  
 usePower();  
 }  
};  
  
// Diamond problem occurs here  
class Copier : public Scanner, public Printer {  
public:  
 Copier(int scannerPower, int printerPower)   
 : Scanner(scannerPower), Printer(printerPower) {  
 cout << "Copier constructor called" << endl;  
 }  
   
 void copy() const {  
 scan();  
 print();  
 }  
};  
  
int main() {  
 Copier copier(100, 200);  
   
 // Problem: Which usePower() method to call?  
 // This would be ambiguous:  
 // copier.usePower(); // Ambiguous - from Scanner or Printer?  
   
 // We have to specify:  
 copier.Scanner::usePower();  
 copier.Printer::usePower();  
   
 // But methods unique to each class work fine  
 copier.scan();  
 copier.print();  
 copier.copy();  
   
 return 0;  
}

#### Solving the Diamond Problem with Virtual Inheritance

Virtual inheritance solves the diamond problem by ensuring only one instance of the common base class:

#include <iostream>  
using namespace std;  
  
class PoweredDevice {  
private:  
 int powerUsage;  
   
public:  
 PoweredDevice(int power) : powerUsage(power) {  
 cout << "PoweredDevice constructor called with power: " << power << endl;  
 }  
   
 void usePower() const {  
 cout << "Using " << powerUsage << " watts of power" << endl;  
 }  
};  
  
// Use virtual inheritance  
class Scanner : virtual public PoweredDevice {  
public:  
 Scanner(int power) : PoweredDevice(power) {  
 cout << "Scanner constructor called" << endl;  
 }  
   
 void scan() const {  
 cout << "Scanning document" << endl;  
 usePower();  
 }  
};  
  
// Use virtual inheritance  
class Printer : virtual public PoweredDevice {  
public:  
 Printer(int power) : PoweredDevice(power) {  
 cout << "Printer constructor called" << endl;  
 }  
   
 void print() const {  
 cout << "Printing document" << endl;  
 usePower();  
 }  
};  
  
class Copier : public Scanner, public Printer {  
public:  
 // With virtual inheritance, the most derived class is responsible  
 // for calling the constructor of the virtual base class  
 Copier(int power)   
 : PoweredDevice(power), Scanner(power), Printer(power) {  
 cout << "Copier constructor called" << endl;  
 }  
   
 void copy() const {  
 scan();  
 print();  
 }  
};  
  
int main() {  
 Copier copier(150);  
   
 // No ambiguity - only one PoweredDevice instance exists  
 copier.usePower();  
   
 copier.copy();  
   
 return 0;  
}

#### Best Practices for Multiple Inheritance

1. **Use it sparingly**: Multiple inheritance often introduces complexity
2. **Prefer composition**: Consider using composition over multiple inheritance
3. **Use virtual inheritance** when inheriting from classes with a common base
4. **Keep interface and implementation separate**: Use multiple inheritance primarily for interface inheritance
5. **Avoid name collisions**: Prevent ambiguity by using different names or explicit qualification

### 6.5.3 Multilevel Inheritance

Multilevel inheritance creates a chain of inheritance where a derived class becomes the base class for another derived class.

#include <iostream>  
#include <string>  
using namespace std;  
  
// Base class  
class Animal {  
protected:  
 string name;  
   
public:  
 Animal(const string& n) : name(n) {  
 cout << "Animal constructor called" << endl;  
 }  
   
 ~Animal() {  
 cout << "Animal destructor called" << endl;  
 }  
   
 void eat() const {  
 cout << name << " is eating" << endl;  
 }  
   
 void sleep() const {  
 cout << name << " is sleeping" << endl;  
 }  
};  
  
// First level of inheritance  
class Mammal : public Animal {  
private:  
 int pregnancyDuration; // in months  
   
public:  
 Mammal(const string& n, int duration)   
 : Animal(n), pregnancyDuration(duration) {  
 cout << "Mammal constructor called" << endl;  
 }  
   
 ~Mammal() {  
 cout << "Mammal destructor called" << endl;  
 }  
   
 void giveBirth() const {  
 cout << name << " gives birth after " << pregnancyDuration   
 << " months of pregnancy" << endl;  
 }  
};  
  
// Second level of inheritance  
class Cat : public Mammal {  
private:  
 bool isIndoor;  
   
public:  
 Cat(const string& n, bool indoor)   
 : Mammal(n, 2), isIndoor(indoor) { // Cats have a 2-month pregnancy  
 cout << "Cat constructor called" << endl;  
 }  
   
 ~Cat() {  
 cout << "Cat destructor called" << endl;  
 }  
   
 void purr() const {  
 cout << name << " is purring" << endl;  
 }  
   
 void displayInfo() const {  
 cout << name << " is a " << (isIndoor ? "indoor" : "outdoor") << " cat" << endl;  
 }  
};  
  
int main() {  
 Cat myCat("Whiskers", true);  
   
 // Methods from Animal (base class)  
 myCat.eat();  
 myCat.sleep();  
   
 // Method from Mammal (intermediate class)  
 myCat.giveBirth();  
   
 // Methods from Cat (derived class)  
 myCat.purr();  
 myCat.displayInfo();  
   
 return 0;  
}

#### Key Aspects of Multilevel Inheritance

1. **Constructor/Destructor Chain**:
   * Constructors are called in order from base to most derived
   * Destructors are called in reverse order, from most derived to base
2. **Method Resolution**:
   * Calls to methods are resolved by searching first in the most derived class
   * If not found, the search continues up the inheritance hierarchy
3. **Access Control Propagation**:
   * Access specifiers apply at each level of inheritance
   * Private members of Animal are not directly accessible in Cat

#### Potential Issues with Deep Inheritance Hierarchies

1. **Complexity**: Deep hierarchies can be hard to understand and maintain
2. **Fragility**: Changes in base classes impact all derived classes
3. **Performance**: Long chains of virtual function calls may affect performance
4. **Inheritance leakage**: Implementation details may leak through the hierarchy

### 6.5.4 Hierarchical Inheritance

Hierarchical inheritance occurs when multiple classes inherit from a single base class.

#include <iostream>  
#include <string>  
using namespace std;  
  
// Base class  
class Employee {  
protected:  
 string name;  
 int id;  
 double baseSalary;  
   
public:  
 Employee(const string& n, int i, double salary)  
 : name(n), id(i), baseSalary(salary) {  
 cout << "Employee constructor called" << endl;  
 }  
   
 ~Employee() {  
 cout << "Employee destructor called" << endl;  
 }  
   
 void displayInfo() const {  
 cout << "Name: " << name << ", ID: " << id  
 << ", Base Salary: $" << baseSalary << endl;  
 }  
   
 virtual double calculateSalary() const {  
 return baseSalary;  
 }  
};  
  
// Derived class 1  
class Developer : public Employee {  
private:  
 string programmingLanguage;  
 int linesOfCode;  
   
public:  
 Developer(const string& n, int i, double salary,  
 const string& lang, int loc)  
 : Employee(n, i, salary),   
 programmingLanguage(lang),  
 linesOfCode(loc) {  
 cout << "Developer constructor called" << endl;  
 }  
   
 ~Developer() {  
 cout << "Developer destructor called" << endl;  
 }  
   
 void writeCode() const {  
 cout << name << " is writing code in " << programmingLanguage << endl;  
 }  
   
 double calculateSalary() const override {  
 // Bonus based on lines of code  
 return baseSalary + (linesOfCode / 1000) \* 500;  
 }  
};  
  
// Derived class 2  
class Manager : public Employee {  
private:  
 int teamSize;  
   
public:  
 Manager(const string& n, int i, double salary, int size)  
 : Employee(n, i, salary), teamSize(size) {  
 cout << "Manager constructor called" << endl;  
 }  
   
 ~Manager() {  
 cout << "Manager destructor called" << endl;  
 }  
   
 void manageMeeting() const {  
 cout << name << " is conducting a meeting with "   
 << teamSize << " team members" << endl;  
 }  
   
 double calculateSalary() const override {  
 // Managers get bonus based on team size  
 return baseSalary + teamSize \* 1000;  
 }  
};  
  
// Derived class 3  
class Designer : public Employee {  
private:  
 string designTool;  
 int projects;  
   
public:  
 Designer(const string& n, int i, double salary,  
 const string& tool, int p)  
 : Employee(n, i, salary), designTool(tool), projects(p) {  
 cout << "Designer constructor called" << endl;  
 }  
   
 ~Designer() {  
 cout << "Designer destructor called" << endl;  
 }  
   
 void createDesign() const {  
 cout << name << " is creating designs using " << designTool << endl;  
 }  
   
 double calculateSalary() const override {  
 // Designers get bonus based on projects  
 return baseSalary + projects \* 750;  
 }  
};  
  
int main() {  
 Developer dev("Alice", 1001, 70000, "C++", 5000);  
 Manager mgr("Bob", 2001, 85000, 8);  
 Designer des("Charlie", 3001, 65000, "Photoshop", 12);  
   
 cout << "\nEmployee Information:" << endl;  
 cout << "-------------------" << endl;  
   
 dev.displayInfo();  
 cout << "Total Salary: $" << dev.calculateSalary() << endl;  
 dev.writeCode();  
   
 cout << endl;  
   
 mgr.displayInfo();  
 cout << "Total Salary: $" << mgr.calculateSalary() << endl;  
 mgr.manageMeeting();  
   
 cout << endl;  
   
 des.displayInfo();  
 cout << "Total Salary: $" << des.calculateSalary() << endl;  
 des.createDesign();  
   
 return 0;  
}

#### Benefits of Hierarchical Inheritance

1. **Code Organization**: Common attributes and behaviors are defined in the base class
2. **Code Reuse**: Derived classes reuse code from the base class
3. **Specialization**: Each derived class can add its own specialized functionality
4. **Polymorphism**: Base class pointers can refer to objects of any derived class (discussed in section 6.6)

#### Potential Issues

1. **Base Class Changes**: Changes to the base class affect all derived classes
2. **Design Constraints**: The base class design influences all derived classes
3. **Feature Bloat**: The base class may accumulate features to support all derived classes

### 6.5.5 Hybrid Inheritance

Hybrid inheritance is a combination of multiple inheritance types (like multilevel and multiple inheritance).

#include <iostream>  
#include <string>  
using namespace std;  
  
// Base class  
class Person {  
protected:  
 string name;  
 int age;  
   
public:  
 Person(const string& n, int a) : name(n), age(a) {  
 cout << "Person constructor called" << endl;  
 }  
   
 void displayPerson() const {  
 cout << "Name: " << name << ", Age: " << age << endl;  
 }  
};  
  
// First level derived class  
class Student : virtual public Person {  
protected:  
 int studentId;  
   
public:  
 Student(const string& n, int a, int id)  
 : Person(n, a), studentId(id) {  
 cout << "Student constructor called" << endl;  
 }  
   
 void displayStudent() const {  
 cout << "Student ID: " << studentId << endl;  
 }  
};  
  
// First level derived class  
class Employee : virtual public Person {  
protected:  
 int employeeId;  
 double salary;  
   
public:  
 Employee(const string& n, int a, int id, double s)  
 : Person(n, a), employeeId(id), salary(s) {  
 cout << "Employee constructor called" << endl;  
 }  
   
 void displayEmployee() const {  
 cout << "Employee ID: " << employeeId << ", Salary: $" << salary << endl;  
 }  
};  
  
// Second level derived class (multiple + multilevel)  
class TeachingAssistant : public Student, public Employee {  
private:  
 string department;  
 int hoursPerWeek;  
   
public:  
 TeachingAssistant(const string& n, int a, int studentId, int employeeId,  
 double salary, const string& dept, int hours)  
 : Person(n, a), // Must initialize virtual base class  
 Student(n, a, studentId),  
 Employee(n, a, employeeId, salary),  
 department(dept), hoursPerWeek(hours) {  
 cout << "TeachingAssistant constructor called" << endl;  
 }  
   
 void displayTA() const {  
 // Base class display methods  
 displayPerson();  
 displayStudent();  
 displayEmployee();  
   
 // Additional TA information  
 cout << "Department: " << department << endl;  
 cout << "Hours per week: " << hoursPerWeek << endl;  
 }  
};  
  
int main() {  
 TeachingAssistant ta("David", 25, 10045, 5001, 20000, "Computer Science", 20);  
   
 cout << "\nTeaching Assistant Information:" << endl;  
 cout << "-----------------------------" << endl;  
 ta.displayTA();  
   
 return 0;  
}

#### Key Aspects of Hybrid Inheritance

1. **Complexity Management**: Virtual inheritance is often necessary to avoid the diamond problem
2. **Constructor Initialization**: The most derived class must initialize virtual base classes directly
3. **Multiple Inheritance Paths**: Methods can be inherited through multiple paths
4. **Name Resolution**: May require explicit qualification to resolve ambiguities

### Inheritance Best Practices

1. **Use “is-a” relationships**: Inheritance should represent an “is-a” relationship (Car is-a Vehicle)
2. **Prefer composition for “has-a” relationships**: Use composition instead of inheritance for “has-a” relationships
3. **Keep inheritance hierarchies shallow**: Avoid deep inheritance chains
4. **Design for inheritance or prohibit it**:
   * Design classes explicitly for inheritance, or
   * Make them final/sealed to prevent inheritance
5. **Use virtual destructors** in base classes when using polymorphism
6. **Follow the Liskov Substitution Principle**: Derived classes should be substitutable for their base classes
7. **Don’t override non-virtual methods**: Override only virtual methods from base classes
8. **Access base class methods explicitly** when needed using scope resolution operator (::)
9. **Initialize base classes properly** in constructors
10. **Consider alternatives to multiple inheritance** where possible

## 6.6 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class. The word polymorphism means “many forms.” In C++, polymorphism can be:

1. **Compile-time polymorphism** (static binding): Function overloading, operator overloading
2. **Runtime polymorphism** (dynamic binding): Virtual functions, function overriding

### 6.6.1 Compile-time Polymorphism (Function/Operator Overloading)

Compile-time polymorphism is resolved at compile time and doesn’t require any runtime type checking.

#### Function Overloading

Function overloading allows multiple functions with the same name but different parameters:

#include <iostream>  
#include <string>  
using namespace std;  
  
class Calculator {  
public:  
 // Overloaded functions with different parameter types  
 int add(int a, int b) {  
 cout << "Adding two integers" << endl;  
 return a + b;  
 }  
   
 double add(double a, double b) {  
 cout << "Adding two doubles" << endl;  
 return a + b;  
 }  
   
 string add(const string& a, const string& b) {  
 cout << "Concatenating two strings" << endl;  
 return a + b;  
 }  
   
 // Overloaded functions with different number of parameters  
 int add(int a, int b, int c) {  
 cout << "Adding three integers" << endl;  
 return a + b + c;  
 }  
   
 // Overloaded functions with different parameter order  
 double add(int a, double b) {  
 cout << "Adding integer and double" << endl;  
 return a + b;  
 }  
   
 double add(double a, int b) {  
 cout << "Adding double and integer" << endl;  
 return a + b;  
 }  
};  
  
int main() {  
 Calculator calc;  
   
 cout << "Result: " << calc.add(5, 3) << endl;  
 cout << "Result: " << calc.add(3.5, 2.7) << endl;  
 cout << "Result: " << calc.add("Hello, ", "World!") << endl;  
 cout << "Result: " << calc.add(1, 2, 3) << endl;  
 cout << "Result: " << calc.add(10, 3.5) << endl;  
 cout << "Result: " << calc.add(2.5, 10) << endl;  
   
 return 0;  
}

The compiler determines which function to call based on: 1. Number of arguments 2. Types of arguments 3. Const-ness of arguments 4. Reference qualifiers (lvalue/rvalue)

#### Function Overloading Resolution

The compiler follows these steps to resolve function calls: 1. Find exact matches (considering trivial conversions) 2. Try standard conversions (like int to double) 3. Try user-defined conversions 4. Try ellipsis matches 5. If multiple matches are found at the same level, the call is ambiguous

#include <iostream>  
using namespace std;  
  
void display(int x) {  
 cout << "Integer: " << x << endl;  
}  
  
void display(double x) {  
 cout << "Double: " << x << endl;  
}  
  
void display(char\* x) {  
 cout << "String: " << x << endl;  
}  
  
int main() {  
 display(10); // Calls display(int)  
 display(10.5); // Calls display(double)  
 display("Hello"); // Calls display(char\*)  
   
 // What happens with this?  
 // display('A'); // 'A' is a char, not int or double  
 // The char 'A' is implicitly converted to int, calling display(int)  
   
 return 0;  
}

#### Function Overloading vs Function Overriding

* **Overloading**: Multiple functions with the same name but different parameters
* **Overriding**: Redefining a function from a base class in a derived class (same name, same parameters)

#### Operator Overloading

Operator overloading allows you to redefine how operators work for user-defined types:

#include <iostream>  
using namespace std;  
  
class Complex {  
private:  
 double real;  
 double imag;  
   
public:  
 Complex(double r = 0, double i = 0) : real(r), imag(i) {}  
   
 // Overload the + operator  
 Complex operator+(const Complex& other) const {  
 return Complex(real + other.real, imag + other.imag);  
 }  
   
 // Overload the \* operator  
 Complex operator\*(const Complex& other) const {  
 return Complex(  
 real \* other.real - imag \* other.imag,  
 real \* other.imag + imag \* other.real  
 );  
 }  
   
 // Overload the == operator  
 bool operator==(const Complex& other) const {  
 return (real == other.real && imag == other.imag);  
 }  
   
 // Overload the != operator  
 bool operator!=(const Complex& other) const {  
 return !(\*this == other);  
 }  
   
 // Overload the << operator as a friend function  
 friend ostream& operator<<(ostream& os, const Complex& c);  
   
 // Overload the >> operator as a friend function  
 friend istream& operator>>(istream& is, Complex& c);  
};  
  
// Implementation of friend functions outside the class  
ostream& operator<<(ostream& os, const Complex& c) {  
 os << c.real;  
 if (c.imag >= 0)  
 os << "+" << c.imag << "i";  
 else  
 os << c.imag << "i";  
 return os;  
}  
  
istream& operator>>(istream& is, Complex& c) {  
 cout << "Enter real part: ";  
 is >> c.real;  
 cout << "Enter imaginary part: ";  
 is >> c.imag;  
 return is;  
}  
  
int main() {  
 Complex a(3, 4);  
 Complex b(1, 2);  
   
 // Using overloaded operators  
 Complex c = a + b;  
 Complex d = a \* b;  
   
 cout << "a = " << a << endl;  
 cout << "b = " << b << endl;  
 cout << "a + b = " << c << endl;  
 cout << "a \* b = " << d << endl;  
   
 if (a == b)  
 cout << "a equals b" << endl;  
 else  
 cout << "a does not equal b" << endl;  
   
 // Using overloaded input operator  
 Complex userInput;  
 cin >> userInput;  
 cout << "You entered: " << userInput << endl;  
   
 return 0;  
}

#### Commonly Overloaded Operators

| Category | Operators |
| --- | --- |
| Arithmetic | +, -, \*, /, %, ++, -- |
| Comparison | ==, !=, <, >, <=, >= |
| Logical | !, &&, \|\| |
| Bitwise | &, \|, ^, ~, <<, >> |
| Assignment | =, +=, -=, \*=, /=, etc. |
| Memory | new, delete, new[], delete[] |
| Other | (), [], ->, , |

#### Rules and Best Practices for Operator Overloading

1. **Can’t change precedence**: The precedence of operators remains the same
2. **Can’t change arity**: Unary operators remain unary, binary operators remain binary
3. **Can’t create new operators**: You can only overload existing operators
4. **Can’t overload operators for built-in types**: Only for user-defined types
5. **Some operators can’t be overloaded**: ., .\*, ::, ? :, sizeof
6. **Member vs. Non-member**:
   * Binary operators with left-side object: Usually member functions
   * Binary operators with modified left operand: Non-member functions
   * Assignment, subscript, function call: Must be members
7. **Return appropriate types**:
   * Arithmetic operators usually return by value
   * Assignment operators return a reference to \*this

### 6.6.2 Run-time Polymorphism (Virtual Functions, vtables)

Runtime polymorphism is achieved through virtual functions and is resolved at runtime based on the actual type of the object.

#### Virtual Functions

A virtual function is a member function that can be redefined in derived classes. The compiler determines which function to call at runtime:

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
class Shape {  
protected:  
 string name;  
   
public:  
 Shape(const string& n) : name(n) {}  
   
 // Virtual function  
 virtual double area() const {  
 return 0.0;  
 }  
   
 // Virtual function  
 virtual double perimeter() const {  
 return 0.0;  
 }  
   
 // Non-virtual function  
 string getName() const {  
 return name;  
 }  
   
 // Virtual destructor - important!  
 virtual ~Shape() {  
 cout << "Shape destructor: " << name << endl;  
 }  
};  
  
class Circle : public Shape {  
private:  
 double radius;  
   
public:  
 Circle(const string& n, double r) : Shape(n), radius(r) {}  
   
 // Override virtual function  
 double area() const override {  
 return 3.14159 \* radius \* radius;  
 }  
   
 // Override virtual function  
 double perimeter() const override {  
 return 2 \* 3.14159 \* radius;  
 }  
   
 ~Circle() {  
 cout << "Circle destructor: " << name << endl;  
 }  
};  
  
class Rectangle : public Shape {  
private:  
 double width;  
 double height;  
   
public:  
 Rectangle(const string& n, double w, double h)  
 : Shape(n), width(w), height(h) {}  
   
 // Override virtual function  
 double area() const override {  
 return width \* height;  
 }  
   
 // Override virtual function  
 double perimeter() const override {  
 return 2 \* (width + height);  
 }  
   
 ~Rectangle() {  
 cout << "Rectangle destructor: " << name << endl;  
 }  
};  
  
int main() {  
 // Create objects  
 Circle circle("Circle1", 5.0);  
 Rectangle rectangle("Rectangle1", 4.0, 6.0);  
   
 // Direct calls - no polymorphism  
 cout << "Direct calls:" << endl;  
 cout << circle.getName() << " area: " << circle.area() << endl;  
 cout << rectangle.getName() << " area: " << rectangle.area() << endl;  
   
 // Polymorphic calls through base class pointers  
 cout << "\nPolymorphic calls:" << endl;  
 Shape\* shapes[] = { &circle, &rectangle };  
   
 for (Shape\* shape : shapes) {  
 cout << shape->getName() << " area: " << shape->area() << endl;  
 cout << shape->getName() << " perimeter: " << shape->perimeter() << endl;  
 }  
   
 // Dynamic allocation  
 Shape\* dynamicCircle = new Circle("DynamicCircle", 3.0);  
 Shape\* dynamicRectangle = new Rectangle("DynamicRectangle", 2.0, 7.0);  
   
 cout << "\nDynamically allocated objects:" << endl;  
 cout << dynamicCircle->getName() << " area: " << dynamicCircle->area() << endl;  
 cout << dynamicRectangle->getName() << " area: " << dynamicRectangle->area() << endl;  
   
 // Clean up dynamic objects - virtual destructor ensures proper cleanup  
 delete dynamicCircle;  
 delete dynamicRectangle;  
   
 return 0;  
}

#### How Virtual Functions Work: The vtable

Virtual functions are implemented using a mechanism called the vtable (virtual function table):

1. **vtable**: A table of function pointers, one for each virtual function in the class
2. **vptr**: A hidden pointer added to objects of classes with virtual functions
3. **Working**:
   * Each class with virtual functions has its own vtable
   * Each object has a vptr pointing to its class’s vtable
   * When a virtual function is called through a pointer or reference, the correct function is found via the vtable

Memory layout:  
  
Base class object:  
+----------------+  
| vptr | --> Base class vtable:  
+----------------+ +-----------------+  
| Base members | | Base::func1() |  
+----------------+ | Base::func2() |  
 +-----------------+  
  
Derived class object:  
+----------------+  
| vptr | --> Derived class vtable:  
+----------------+ +-------------------+  
| Base members | | Derived::func1() | (overridden)  
+----------------+ | Base::func2() | (inherited)  
| Derived members| +-------------------+  
+----------------+

#### Virtual Destructors

When using polymorphism, base classes should have virtual destructors to ensure proper cleanup of derived classes:

#include <iostream>  
using namespace std;  
  
class Base {  
public:  
 Base() { cout << "Base constructor" << endl; }  
   
 // Case 1: Non-virtual destructor  
 ~Base() { cout << "Base destructor" << endl; }  
   
 // Case 2: Virtual destructor  
 // virtual ~Base() { cout << "Base virtual destructor" << endl; }  
};  
  
class Derived : public Base {  
private:  
 int\* data;  
   
public:  
 Derived() : Base() {  
 cout << "Derived constructor" << endl;  
 data = new int[100];  
 }  
   
 ~Derived() {  
 cout << "Derived destructor" << endl;  
 delete[] data;  
 }  
};  
  
int main() {  
 // Case 1: Direct object - always works correctly  
 cout << "Creating and destroying Derived object directly:" << endl;  
 {  
 Derived d;  
 } // Both destructors called  
   
 cout << "\nCreating and destroying via Base pointer:" << endl;  
 {  
 Base\* b = new Derived();  
 delete b; // Without virtual destructor, only Base destructor is called!  
 // This leads to a memory leak as Derived's destructor is not called  
 }  
   
 return 0;  
}

#### Pure Virtual Functions and Abstract Classes

A pure virtual function is declared with = 0 and has no implementation in the base class. A class with at least one pure virtual function becomes an abstract class:

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
// Abstract class (has at least one pure virtual function)  
class Shape {  
protected:  
 string name;  
   
public:  
 Shape(const string& n) : name(n) {}  
   
 // Pure virtual function - must be implemented by derived classes  
 virtual double area() const = 0;  
   
 // Pure virtual function  
 virtual double perimeter() const = 0;  
   
 // Regular virtual function with default implementation  
 virtual void display() const {  
 cout << "Shape: " << name << endl;  
 cout << "Area: " << area() << endl;  
 cout << "Perimeter: " << perimeter() << endl;  
 }  
   
 string getName() const {  
 return name;  
 }  
   
 virtual ~Shape() {  
 cout << "Shape destructor" << endl;  
 }  
};  
  
class Circle : public Shape {  
private:  
 double radius;  
   
public:  
 Circle(const string& n, double r) : Shape(n), radius(r) {}  
   
 // Implementation of pure virtual function  
 double area() const override {  
 return 3.14159 \* radius \* radius;  
 }  
   
 // Implementation of pure virtual function  
 double perimeter() const override {  
 return 2 \* 3.14159 \* radius;  
 }  
   
 // Override virtual function  
 void display() const override {  
 cout << "Circle: " << name << endl;  
 cout << "Radius: " << radius << endl;  
 cout << "Area: " << area() << endl;  
 cout << "Circumference: " << perimeter() << endl;  
 }  
};  
  
class Rectangle : public Shape {  
private:  
 double width;  
 double height;  
   
public:  
 Rectangle(const string& n, double w, double h)  
 : Shape(n), width(w), height(h) {}  
   
 // Implementation of pure virtual function  
 double area() const override {  
 return width \* height;  
 }  
   
 // Implementation of pure virtual function  
 double perimeter() const override {  
 return 2 \* (width + height);  
 }  
   
 // Using default display implementation from Shape  
};  
  
int main() {  
 // Shape shape("GenericShape"); // Error - cannot instantiate abstract class  
   
 Circle circle("Circle1", 5.0);  
 Rectangle rectangle("Rectangle1", 4.0, 6.0);  
   
 // Polymorphic container  
 vector<Shape\*> shapes;  
 shapes.push\_back(&circle);  
 shapes.push\_back(&rectangle);  
   
 // Polymorphic behavior  
 for (const Shape\* shape : shapes) {  
 shape->display();  
 cout << endl;  
 }  
   
 return 0;  
}

#### Key Points About Abstract Classes

1. **Cannot be instantiated** directly
2. **Can contain both pure virtual and regular functions**
3. **Derived classes must implement all pure virtual functions**
4. **Useful for defining interfaces** in C++

#### Virtual Function Table (vtable) in Detail

The vtable mechanism is how C++ implements runtime polymorphism:

#include <iostream>  
using namespace std;  
  
class Base {  
public:  
 virtual void func1() { cout << "Base::func1()" << endl; }  
 virtual void func2() { cout << "Base::func2()" << endl; }  
 void func3() { cout << "Base::func3()" << endl; } // Non-virtual  
 virtual ~Base() { cout << "Base destructor" << endl; }  
};  
  
class Derived : public Base {  
public:  
 // Override func1  
 void func1() override { cout << "Derived::func1()" << endl; }  
 // func2 is inherited from Base  
 // Override non-virtual func3 (not polymorphic)  
 void func3() { cout << "Derived::func3()" << endl; }  
 ~Derived() { cout << "Derived destructor" << endl; }  
};  
  
int main() {  
 cout << "Calling through Base pointer to Base object:" << endl;  
 Base\* b1 = new Base();  
 b1->func1(); // Base::func1()  
 b1->func2(); // Base::func2()  
 b1->func3(); // Base::func3()  
 delete b1;  
   
 cout << "\nCalling through Base pointer to Derived object:" << endl;  
 Base\* b2 = new Derived();  
 b2->func1(); // Derived::func1() - virtual, uses vtable  
 b2->func2(); // Base::func2() - virtual, uses vtable  
 b2->func3(); // Base::func3() - non-virtual, determined by pointer type  
 delete b2; // Calls both destructors due to virtual destructor  
   
 cout << "\nCalling through Derived pointer to Derived object:" << endl;  
 Derived\* d = new Derived();  
 d->func1(); // Derived::func1()  
 d->func2(); // Base::func2()  
 d->func3(); // Derived::func3() - non-virtual, determined by pointer type  
 delete d;  
   
 return 0;  
}

#### Performance Considerations for Virtual Functions

1. **Memory Overhead**: Each object with virtual functions has a vptr
2. **Speed Overhead**: Virtual function calls require an extra indirection
3. **Cache Misses**: Virtual function calls can lead to more cache misses
4. **Compiler Optimization**: Virtual functions can limit optimizations like inlining

### Polymorphism Best Practices

1. **Use virtual destructors** in base classes
2. **Use the override keyword** (C++11) for derived class functions
3. **Use the final keyword** (C++11) to prevent further overriding
4. **Consider using pure virtual functions** for interfaces
5. **Be cautious with multiple inheritance** and virtual functions
6. **Avoid calling virtual functions in constructors/destructors**
7. **Understand the performance implications** of virtual functions
8. **Use runtime polymorphism only when needed**

// Using override and final keywords  
class Base {  
public:  
 virtual void foo() { cout << "Base::foo()" << endl; }  
 virtual void bar() { cout << "Base::bar()" << endl; }  
 virtual ~Base() {}  
};  
  
class Derived : public Base {  
public:  
 void foo() override { cout << "Derived::foo()" << endl; }  
 void bar() override final { cout << "Derived::bar()" << endl; }  
};  
  
class Further : public Derived {  
public:  
 void foo() override { cout << "Further::foo()" << endl; }  
 // void bar() override { } // Error: cannot override final function  
};

#### Object Slicing

Object slicing occurs when a derived class object is assigned to a base class object (not a pointer or reference):

#include <iostream>  
using namespace std;  
  
class Base {  
protected:  
 int baseValue;  
   
public:  
 Base(int value) : baseValue(value) {}  
   
 virtual void display() const {  
 cout << "Base class with value: " << baseValue << endl;  
 }  
};  
  
class Derived : public Base {  
private:  
 int derivedValue;  
   
public:  
 Derived(int baseVal, int derivedVal)   
 : Base(baseVal), derivedValue(derivedVal) {}  
   
 void display() const override {  
 cout << "Derived class with base value: " << baseValue   
 << " and derived value: " << derivedValue << endl;  
 }  
};  
  
void processReference(const Base& obj) {  
 // No slicing here - virtual function call works correctly  
 obj.display();  
}  
  
void processObject(Base obj) {  
 // Object slicing! Only Base part is copied  
 obj.display(); // Always calls Base::display  
}  
  
int main() {  
 Derived derivedObj(10, 20);  
   
 cout << "Original object:" << endl;  
 derivedObj.display(); // Calls Derived::display  
   
 cout << "\nThrough base reference (no slicing):" << endl;  
 processReference(derivedObj); // Calls Derived::display  
   
 cout << "\nThrough base object (slicing occurs):" << endl;  
 processObject(derivedObj); // Calls Base::display  
   
 cout << "\nAssignment to base (slicing):" << endl;  
 Base baseObj = derivedObj; // Object slicing!  
 baseObj.display(); // Calls Base::display  
   
 return 0;  
}

To avoid object slicing: 1. Use pointers or references when working with polymorphic objects 2. Make base classes abstract to prevent direct instantiation 3. Delete the copy constructor and assignment operator if appropriate 4. Consider using shared ownership (shared\_ptr) if ownership transfer is needed

Polymorphism is one of the most powerful features of C++ and object-oriented programming in general. It allows you to write code that works with objects of different types through a common interface, making your code more flexible and extensible.

# Chapter 6: Object-Oriented Programming (Part 3)

## 6.7 Encapsulation & Abstraction

Encapsulation and abstraction are two fundamental principles of object-oriented programming that work together to create maintainable, robust software systems.

### Encapsulation

Encapsulation is the bundling of data and methods that operate on that data into a single unit (class) and restricting access to the internal representation of the object.

#### Key Aspects of Encapsulation:

1. **Data Hiding**: Making data members private so they can’t be accessed directly from outside the class
2. **Access Control**: Using access specifiers (private, protected, public) to control visibility
3. **Interface Definition**: Providing a clean public interface for interacting with the object
4. **Implementation Protection**: Preventing external code from depending on implementation details

#### Benefits of Encapsulation:

1. **Improved Maintainability**: Implementation details can be changed without affecting client code
2. **Reduced Complexity**: Users only need to understand the public interface, not the internal implementation
3. **Better Control**: Validate inputs and maintain invariants through controlled access
4. **Modularity**: Classes become self-contained units that can be developed and tested independently

#### Example of Encapsulation:

#include <iostream>  
#include <string>  
using namespace std;  
  
class BankAccount {  
private:  
 // Encapsulated data (hidden from outside)  
 string accountNumber;  
 double balance;  
 string ownerName;  
 double minimumBalance;  
   
 // Private helper method  
 bool isValidAmount(double amount) const {  
 return amount > 0;  
 }  
   
public:  
 // Constructor  
 BankAccount(const string& accNo, const string& name, double initialBalance = 0.0)  
 : accountNumber(accNo), ownerName(name), balance(initialBalance), minimumBalance(100.0) {  
 }  
   
 // Public interface methods  
 void deposit(double amount) {  
 if (isValidAmount(amount)) {  
 balance += amount;  
 cout << "Deposit successful. New balance: $" << balance << endl;  
 } else {  
 cout << "Invalid amount for deposit." << endl;  
 }  
 }  
   
 bool withdraw(double amount) {  
 if (!isValidAmount(amount)) {  
 cout << "Invalid amount for withdrawal." << endl;  
 return false;  
 }  
   
 if (balance - amount < minimumBalance) {  
 cout << "Withdrawal denied. Minimum balance would not be maintained." << endl;  
 return false;  
 }  
   
 balance -= amount;  
 cout << "Withdrawal successful. New balance: $" << balance << endl;  
 return true;  
 }  
   
 // Getters (controlled access to private data)  
 double getBalance() const { return balance; }  
 string getAccountNumber() const { return accountNumber; }  
 string getOwnerName() const { return ownerName; }  
   
 // Setters with validation  
 void setOwnerName(const string& name) {  
 if (!name.empty()) {  
 ownerName = name;  
 }  
 }  
};  
  
int main() {  
 BankAccount account("123456789", "John Doe", 500.0);  
   
 // Using the public interface  
 account.deposit(200.0);  
 account.withdraw(100.0);  
   
 // Access data through getters  
 cout << "Account: " << account.getAccountNumber() << endl;  
 cout << "Owner: " << account.getOwnerName() << endl;  
 cout << "Current Balance: $" << account.getBalance() << endl;  
   
 // Cannot access private members directly  
 // account.balance = 1000000.0; // Error: 'balance' is private  
 // account.minimumBalance = 0.0; // Error: 'minimumBalance' is private  
   
 return 0;  
}

In this example: - Data members are private, preventing direct external access - Public methods provide a controlled interface for interacting with the account - Business rules (minimum balance, valid amounts) are enforced within the class - Internal implementation details (like the helper method) are hidden

### Abstraction

Abstraction is the concept of exposing only the essential features of an object while hiding the unnecessary details. It’s about creating simple, high-level interfaces that hide complex implementations.

#### Key Aspects of Abstraction:

1. **Simplification**: Represent complex reality in a simplified model
2. **Selective Visibility**: Show only what’s necessary for outside users
3. **Implementation Hiding**: Hide the “how” while exposing the “what”
4. **Focusing on Behavior**: Emphasize what an object does rather than how it does it

#### Mechanisms for Abstraction in C++:

1. **Abstract Classes**: Classes with at least one pure virtual function
2. **Interfaces**: Abstract classes with only pure virtual functions
3. **Encapsulation**: Hiding implementation details through access control

#### Example of Abstraction:

#include <iostream>  
#include <vector>  
#include <string>  
#include <cmath>  
using namespace std;  
  
// Abstract base class (interface)  
class Shape {  
public:  
 // Pure virtual functions define the interface  
 virtual double area() const = 0;  
 virtual double perimeter() const = 0;  
 virtual void draw() const = 0;  
 virtual string getName() const = 0;  
   
 // Virtual destructor  
 virtual ~Shape() {}  
};  
  
// Concrete implementation  
class Circle : public Shape {  
private:  
 double radius;  
 double centerX, centerY;  
   
public:  
 Circle(double r, double x = 0, double y = 0)  
 : radius(r), centerX(x), centerY(y) {}  
   
 double area() const override {  
 return M\_PI \* radius \* radius;  
 }  
   
 double perimeter() const override {  
 return 2 \* M\_PI \* radius;  
 }  
   
 void draw() const override {  
 cout << "Drawing a circle at (" << centerX << ", " << centerY   
 << ") with radius " << radius << endl;  
 }  
   
 string getName() const override {  
 return "Circle";  
 }  
};  
  
// Another concrete implementation  
class Rectangle : public Shape {  
private:  
 double width, height;  
 double posX, posY;  
   
public:  
 Rectangle(double w, double h, double x = 0, double y = 0)  
 : width(w), height(h), posX(x), posY(y) {}  
   
 double area() const override {  
 return width \* height;  
 }  
   
 double perimeter() const override {  
 return 2 \* (width + height);  
 }  
   
 void draw() const override {  
 cout << "Drawing a rectangle at (" << posX << ", " << posY   
 << ") with width " << width << " and height " << height << endl;  
 }  
   
 string getName() const override {  
 return "Rectangle";  
 }  
};  
  
// Client code using abstraction  
void printShapeInfo(const Shape& shape) {  
 cout << "Shape: " << shape.getName() << endl;  
 cout << "Area: " << shape.area() << endl;  
 cout << "Perimeter: " << shape.perimeter() << endl;  
 shape.draw();  
 cout << endl;  
}  
  
int main() {  
 Circle circle(5, 10, 20);  
 Rectangle rectangle(4, 6, 15, 25);  
   
 // Using shapes through their abstract interface  
 printShapeInfo(circle);  
 printShapeInfo(rectangle);  
   
 // Collection of different shapes through common interface  
 vector<Shape\*> shapes;  
 shapes.push\_back(new Circle(3, 5, 5));  
 shapes.push\_back(new Rectangle(8, 2, 0, 0));  
 shapes.push\_back(new Circle(7, -10, 8));  
   
 for (const auto& shape : shapes) {  
 printShapeInfo(\*shape);  
 delete shape;  
 }  
   
 return 0;  
}

In this example: - Shape is an abstract class defining operations all shapes must support - Circle and Rectangle provide concrete implementations of those operations - Client code (printShapeInfo and main) uses shapes through the abstract interface - The details of how each shape calculates its area or draws itself are abstracted away

### Difference Between Encapsulation and Abstraction

While closely related, encapsulation and abstraction have distinct focuses:

| Aspect | Encapsulation | Abstraction |
| --- | --- | --- |
| **Focus** | Information hiding | Complexity reduction |
| **Purpose** | Bundle data and methods, control access | Simplify complex reality, focus on essentials |
| **Implementation** | Access specifiers (private, protected) | Abstract classes, interfaces |
| **Benefit** | Security, maintainability | Simplicity, modularity |
| **Level** | Primarily implementation level | Design level |

### Best Practices for Encapsulation and Abstraction

1. **Keep data members private**: Only expose them through controlled methods
2. **Validate inputs** in public methods to maintain object integrity
3. **Design stable public interfaces**: Avoid frequent changes to public APIs
4. **Use abstract classes** to define common behavior for related classes
5. **Program to interfaces**, not implementations
6. **Hide complexity** behind simple interfaces
7. **Separate interface from implementation** to allow independent evolution
8. **Don’t expose implementation details** in public interfaces

## 6.8 this Pointer

The this pointer is a hidden parameter that is automatically passed to non-static member functions. It points to the object that invoked the member function.

### Key Characteristics of this:

1. **Implicit Parameter**: Automatically available in member functions
2. **Points to Current Object**: Refers to the instance being operated on
3. **Type**: For a class X, this has type X\* const (const X\* for const member functions)
4. **Not Available**: Cannot be used in static member functions (as they don’t operate on specific objects)

### Common Uses of this Pointer:

#### 1. Disambiguating Member Variables and Parameters

#include <iostream>  
#include <string>  
using namespace std;  
  
class Person {  
private:  
 string name;  
 int age;  
   
public:  
 // Parameter names same as member names  
 Person(string name, int age) {  
 // Use this-> to disambiguate  
 this->name = name;  
 this->age = age;  
 }  
   
 void display() const {  
 cout << "Name: " << this->name << ", Age: " << this->age << endl;  
 }  
};  
  
int main() {  
 Person person("Alice", 25);  
 person.display();  
   
 return 0;  
}

#### 2. Method Chaining (Fluent Interface)

#include <iostream>  
#include <string>  
using namespace std;  
  
class StringBuilder {  
private:  
 string data;  
   
public:  
 StringBuilder() : data("") {}  
   
 // Return \*this for method chaining  
 StringBuilder& append(const string& str) {  
 data += str;  
 return \*this;  
 }  
   
 StringBuilder& appendLine(const string& str) {  
 data += str + "\n";  
 return \*this;  
 }  
   
 StringBuilder& clear() {  
 data.clear();  
 return \*this;  
 }  
   
 string toString() const {  
 return data;  
 }  
};  
  
int main() {  
 StringBuilder builder;  
   
 // Method chaining using 'this' pointer  
 string result = builder.append("Hello, ")  
 .append("World!")  
 .appendLine("")  
 .append("How are ")  
 .append("you?")  
 .toString();  
   
 cout << result << endl;  
   
 return 0;  
}

#### 3. Passing the Current Object to Other Functions

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
class Event; // Forward declaration  
  
class EventListener {  
public:  
 virtual void onEvent(const Event& event) = 0;  
 virtual ~EventListener() {}  
};  
  
class Event {  
private:  
 string name;  
 string data;  
 vector<EventListener\*> listeners;  
   
public:  
 Event(const string& name, const string& data)   
 : name(name), data(data) {}  
   
 void addListener(EventListener\* listener) {  
 listeners.push\_back(listener);  
 }  
   
 void trigger() {  
 cout << "Triggering event: " << name << endl;  
 for (auto listener : listeners) {  
 listener->onEvent(\*this); // Passing the current object  
 }  
 }  
   
 string getName() const { return name; }  
 string getData() const { return data; }  
};  
  
class Logger : public EventListener {  
public:  
 void onEvent(const Event& event) override {  
 cout << "Logging: Event " << event.getName()   
 << " triggered with data: " << event.getData() << endl;  
 }  
};  
  
class Notifier : public EventListener {  
public:  
 void onEvent(const Event& event) override {  
 cout << "Notification: " << event.getName() << " occurred!" << endl;  
 }  
};  
  
int main() {  
 Logger logger;  
 Notifier notifier;  
   
 Event userLogin("UserLogin", "user\_id=12345");  
   
 userLogin.addListener(&logger);  
 userLogin.addListener(&notifier);  
   
 userLogin.trigger();  
   
 return 0;  
}

#### 4. Comparing Objects (Self-Reference Check)

#include <iostream>  
using namespace std;  
  
class MyClass {  
private:  
 int value;  
   
public:  
 MyClass(int val) : value(val) {}  
   
 int getValue() const { return value; }  
   
 bool isEqualTo(const MyClass& other) const {  
 if (this == &other) { // Compare addresses using 'this'  
 cout << "Self-comparison detected" << endl;  
 return true; // An object is always equal to itself  
 }  
 return this->value == other.value;  
 }  
   
 MyClass& operator=(const MyClass& other) {  
 // Check for self-assignment  
 if (this != &other) {  
 // Perform the assignment only if not self-assignment  
 this->value = other.value;  
 }  
 return \*this;  
 }  
};  
  
int main() {  
 MyClass obj1(42);  
 MyClass obj2(42);  
 MyClass obj3(100);  
   
 cout << "obj1 equal to itself: " << obj1.isEqualTo(obj1) << endl;  
 cout << "obj1 equal to obj2: " << obj1.isEqualTo(obj2) << endl;  
 cout << "obj1 equal to obj3: " << obj1.isEqualTo(obj3) << endl;  
   
 obj1 = obj3; // Assignment operator uses 'this' to prevent self-assignment issues  
 cout << "After assignment, obj1 value: " << obj1.getValue() << endl;  
   
 return 0;  
}

#### 5. Implementing Wrapper Classes (Delegation)

#include <iostream>  
#include <memory>  
#include <string>  
using namespace std;  
  
// The actual implementation class  
class DatabaseConnection {  
public:  
 void connect(const string& connectionString) {  
 cout << "Connecting to: " << connectionString << endl;  
 }  
   
 void executeQuery(const string& query) {  
 cout << "Executing: " << query << endl;  
 }  
   
 void disconnect() {  
 cout << "Disconnecting from database" << endl;  
 }  
};  
  
// Wrapper class that delegates to the implementation  
class Database {  
private:  
 unique\_ptr<DatabaseConnection> connection;  
   
public:  
 Database() : connection(make\_unique<DatabaseConnection>()) {}  
   
 Database\* connect(const string& connectionString) {  
 connection->connect(connectionString);  
 return this; // Return this for method chaining  
 }  
   
 Database\* query(const string& sql) {  
 connection->executeQuery(sql);  
 return this;  
 }  
   
 void close() {  
 connection->disconnect();  
 }  
};  
  
int main() {  
 // Using the wrapper with method chaining  
 Database\* db = new Database();  
 db->connect("server=localhost;database=testdb;")  
 ->query("SELECT \* FROM users")  
 ->query("UPDATE users SET active = 1");  
   
 db->close();  
 delete db;  
   
 return 0;  
}

### Understanding this Under the Hood

When you call a member function on an object, the compiler effectively transforms it:

// Your code:  
obj.function(arg1, arg2);  
  
// What effectively happens:  
Class::function(&obj, arg1, arg2);

The compiler passes the address of the object (&obj) as a hidden first parameter, which becomes the this pointer inside the function.

### Best Practices for Using this:

1. **Use it for disambiguation** when parameter names conflict with member names
2. **Return \*this for method chaining** where it makes the interface more fluent
3. **Check for self-reference** in assignment operators and comparison methods
4. **Make it explicit** when it improves readability
5. **Don’t use this unnecessarily** when there’s no ambiguity
6. **Remember it’s implicit** in all non-static member function calls

## 6.9 Friend Functions and Classes

C++’s access control normally prevents external functions and classes from accessing private or protected members. Friend declarations allow you to grant specific external functions or classes access to these private details.

### Friend Functions

A friend function is a non-member function that has access to private and protected members of a class.

#include <iostream>  
using namespace std;  
  
class Box {  
private:  
 double length;  
 double width;  
 double height;  
   
public:  
 Box(double l, double w, double h) : length(l), width(w), height(h) {}  
   
 // Regular member function  
 double volume() const {  
 return length \* width \* height;  
 }  
   
 // Friend function declaration  
 friend void displayBoxDimensions(const Box& box);  
   
 // Friend function to add two boxes  
 friend Box addBoxes(const Box& b1, const Box& b2);  
};  
  
// Friend function definition - can access private members  
void displayBoxDimensions(const Box& box) {  
 cout << "Box dimensions: " << box.length << " x "  
 << box.width << " x " << box.height << endl;  
}  
  
// Another friend function  
Box addBoxes(const Box& b1, const Box& b2) {  
 // Can access private members of both boxes  
 Box result(  
 b1.length + b2.length,  
 b1.width + b2.width,  
 b1.height + b2.height  
 );  
 return result;  
}  
  
int main() {  
 Box smallBox(1, 2, 3);  
 Box bigBox(4, 5, 6);  
   
 // Call member function  
 cout << "Small box volume: " << smallBox.volume() << endl;  
   
 // Call friend functions  
 displayBoxDimensions(smallBox);  
   
 Box combinedBox = addBoxes(smallBox, bigBox);  
 displayBoxDimensions(combinedBox);  
   
 return 0;  
}

### Friend Classes

A friend class can access private and protected members of the class that declares it as a friend.

#include <iostream>  
#include <string>  
using namespace std;  
  
class Student; // Forward declaration  
  
class Course {  
private:  
 string name;  
 int maxCapacity;  
 int enrolledCount;  
   
public:  
 Course(const string& name, int capacity)  
 : name(name), maxCapacity(capacity), enrolledCount(0) {}  
   
 // This method needs access to Student's private members  
 void enrollStudent(Student& student);  
   
 string getName() const { return name; }  
 int getCapacity() const { return maxCapacity; }  
 int getEnrolledCount() const { return enrolledCount; }  
};  
  
class Student {  
private:  
 string name;  
 int id;  
 int courseCount;  
 static const int MAX\_COURSES = 5;  
   
 // Friend class declaration  
 friend class Course;  
   
public:  
 Student(const string& name, int id)  
 : name(name), id(id), courseCount(0) {}  
   
 string getName() const { return name; }  
 int getId() const { return id; }  
 int getCourseCount() const { return courseCount; }  
};  
  
// Now we can define this method, which accesses Student's private members  
void Course::enrollStudent(Student& student) {  
 if (enrolledCount < maxCapacity && student.courseCount < student.MAX\_COURSES) {  
 enrolledCount++;  
 student.courseCount++; // Can access private member because Course is a friend  
 cout << "Enrolled " << student.name << " in " << name << endl;  
 } else if (enrolledCount >= maxCapacity) {  
 cout << "Course " << name << " is full!" << endl;  
 } else {  
 cout << "Student " << student.name << " has too many courses!" << endl;  
 }  
}  
  
int main() {  
 Student alice("Alice", 12345);  
 Student bob("Bob", 67890);  
   
 Course cpp("C++ Programming", 3);  
 Course java("Java Programming", 2);  
 Course python("Python Programming", 2);  
   
 cpp.enrollStudent(alice);  
 cpp.enrollStudent(bob);  
 java.enrollStudent(alice);  
   
 cout << alice.getName() << " is enrolled in " << alice.getCourseCount() << " courses." << endl;  
 cout << bob.getName() << " is enrolled in " << bob.getCourseCount() << " courses." << endl;  
 cout << cpp.getName() << " has " << cpp.getEnrolledCount() << "/" << cpp.getCapacity() << " students." << endl;  
   
 return 0;  
}

### Friend Member Functions

You can also declare specific member functions of another class as friends:

#include <iostream>  
#include <string>  
using namespace std;  
  
class Engine; // Forward declaration  
  
class Car {  
private:  
 string model;  
 int year;  
 Engine\* engine;  
   
public:  
 Car(const string& model, int year);  
 void displayDetails() const;  
   
 // Friend declaration for specific Engine member function  
 friend void Engine::modifyCar(Car& car);  
};  
  
class Engine {  
private:  
 int horsepower;  
 string type;  
   
public:  
 Engine(int hp, const string& t) : horsepower(hp), type(t) {}  
   
 void displaySpecs() const {  
 cout << "Engine Type: " << type << ", Horsepower: " << horsepower << endl;  
 }  
   
 // This function can access Car's private members  
 void modifyCar(Car& car);  
};  
  
// Now we can define Car constructor, since Engine is fully defined  
Car::Car(const string& m, int y) : model(m), year(y) {  
 engine = new Engine(150, "V6");  
}  
  
void Car::displayDetails() const {  
 cout << "Car: " << model << " (" << year << ")" << endl;  
 engine->displaySpecs();  
}  
  
// Define the friend member function  
void Engine::modifyCar(Car& car) {  
 cout << "Modifying " << car.model << "'s engine..." << endl;  
 delete car.engine; // Can access private members  
 car.engine = new Engine(horsepower + 50, "Turbocharged " + type);  
 cout << "Upgrade complete!" << endl;  
}  
  
int main() {  
 Car myCar("Toyota Camry", 2022);  
 myCar.displayDetails();  
   
 // Create an engine object to call the friend member function  
 Engine enhancedEngine(200, "V8");  
 enhancedEngine.modifyCar(myCar);  
   
 myCar.displayDetails();  
   
 return 0;  
}

### Friendship Properties

1. **Friendship is not symmetric**: If class A is a friend of B, B is not automatically a friend of A
2. **Friendship is not transitive**: If A is a friend of B and B is a friend of C, A is not automatically a friend of C
3. **Friendship is not inherited**: If a base class declares a friend, derived classes don’t automatically have that friend

### When to Use Friends

Friends should be used sparingly as they can reduce encapsulation. Good cases for friends include:

1. **Operator overloading** where operators need access to private data
2. **Helper functions** that need intimate access to class internals
3. **Tightly coupled classes** that form a subsystem and need to access each other’s internals
4. **Unit testing** where test classes may need to access private members

### Best Practices for Friends

1. **Use friends sparingly**: Every friend declaration weakens encapsulation
2. **Prefer member functions** when possible
3. **Consider accessors/mutators** before making a friend
4. **Document friend relationships** to make dependencies clear
5. **Group related classes** that share friendship
6. **Don’t make every function or class a friend** - be selective
7. **Watch for compilation dependencies** - forward declarations may be needed

## 6.10 Static Members

Static members belong to the class itself rather than to any specific instance of the class. They exist even if no objects of the class have been created and are shared among all instances.

### Static Data Members

Static data members have these key properties: - Shared among all class instances - Exist even when no objects exist - Only one copy exists regardless of how many class objects are created - Must be defined outside the class (except for inline initialization of const static members in C++17)

#include <iostream>  
#include <string>  
using namespace std;  
  
class BankAccount {  
private:  
 string accountNumber;  
 double balance;  
   
 // Static data member declaration  
 static double interestRate;  
   
 // Const static data member (can be initialized inline in modern C++)  
 static const int minPasswordLength = 8;  
   
 // Static counter to generate account numbers  
 static int nextAccountId;  
   
public:  
 BankAccount(double initialBalance = 0.0)   
 : balance(initialBalance) {  
 // Generate unique account number  
 accountNumber = "ACC" + to\_string(nextAccountId++);  
 }  
   
 void deposit(double amount) {  
 if (amount > 0) {  
 balance += amount;  
 cout << "Deposited $" << amount << " to account " << accountNumber << endl;  
 }  
 }  
   
 void applyInterest() {  
 double interest = balance \* interestRate;  
 balance += interest;  
 cout << "Applied interest: $" << interest << " to account " << accountNumber << endl;  
 }  
   
 // Static method to set interest rate  
 static void setInterestRate(double rate) {  
 if (rate >= 0 && rate <= 0.5) { // Reasonable limits  
 interestRate = rate;  
 cout << "Interest rate updated to " << rate \* 100 << "%" << endl;  
 } else {  
 cout << "Invalid interest rate" << endl;  
 }  
 }  
   
 // Static method to get interest rate  
 static double getInterestRate() {  
 return interestRate;  
 }  
   
 // Regular (non-static) methods  
 double getBalance() const { return balance; }  
 string getAccountNumber() const { return accountNumber; }  
   
 static int getMinPasswordLength() {  
 return minPasswordLength;  
 }  
};  
  
// Definition of static data members (required)  
double BankAccount::interestRate = 0.03; // 3% initial interest rate  
int BankAccount::nextAccountId = 1000;  
  
int main() {  
 // Accessing static members without any objects  
 cout << "Initial interest rate: " << BankAccount::getInterestRate() << endl;  
 cout << "Minimum password length: " << BankAccount::getMinPasswordLength() << endl;  
   
 // Create some accounts  
 BankAccount acc1(1000);  
 BankAccount acc2(2000);  
   
 // Display account info  
 cout << "\nAccount Information:" << endl;  
 cout << acc1.getAccountNumber() << ": $" << acc1.getBalance() << endl;  
 cout << acc2.getAccountNumber() << ": $" << acc2.getBalance() << endl;  
   
 // Apply interest to accounts  
 acc1.applyInterest();  
 acc2.applyInterest();  
   
 // Update interest rate (affects all accounts)  
 BankAccount::setInterestRate(0.04); // 4%  
   
 // Apply interest again with new rate  
 acc1.applyInterest();  
 acc2.applyInterest();  
   
 // Display final balances  
 cout << "\nFinal Account Balances:" << endl;  
 cout << acc1.getAccountNumber() << ": $" << acc1.getBalance() << endl;  
 cout << acc2.getAccountNumber() << ": $" << acc2.getBalance() << endl;  
   
 return 0;  
}

### Static Member Functions

Static member functions have these key characteristics: - Belong to the class, not to objects - Can be called using the class name (without creating objects) - Can only access static members directly - Don’t have a this pointer - Cannot be declared as const, virtual or volatile

#include <iostream>  
#include <vector>  
#include <string>  
#include <ctime>  
using namespace std;  
  
class Logger {  
private:  
 // Static data members  
 static int messageCount;  
 static bool debugMode;  
 static string logFilePath;  
 static vector<string> messageHistory;  
   
 // Non-static data member  
 string componentName;  
   
public:  
 // Enum for log levels  
 enum LogLevel { DEBUG, INFO, WARNING, ERROR, CRITICAL };  
   
 // Constructor  
 Logger(const string& component) : componentName(component) {}  
   
 // Static method to configure the logger  
 static void configure(bool debug, const string& filePath) {  
 debugMode = debug;  
 logFilePath = filePath;  
   
 log("Logger", INFO, "Logger configured. Debug mode: " +   
 string(debugMode ? "ON" : "OFF") + ", Log file: " + logFilePath);  
 }  
   
 // Static method to get current timestamp  
 static string getTimestamp() {  
 time\_t now = time(nullptr);  
 char buffer[80];  
 strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", localtime(&now));  
 return buffer;  
 }  
   
 // Static logging method that can be called without an instance  
 static void log(const string& component, LogLevel level, const string& message) {  
 string levelStr;  
 switch (level) {  
 case DEBUG: levelStr = "DEBUG"; break;  
 case INFO: levelStr = "INFO"; break;  
 case WARNING: levelStr = "WARNING"; break;  
 case ERROR: levelStr = "ERROR"; break;  
 case CRITICAL: levelStr = "CRITICAL"; break;  
 }  
   
 // Skip debug messages if debug mode is off  
 if (level == DEBUG && !debugMode) return;  
   
 // Format the log message  
 string timestamp = getTimestamp();  
 string formattedMessage = timestamp + " [" + levelStr + "] " +   
 component + ": " + message;  
   
 // Print to console  
 cout << formattedMessage << endl;  
   
 // Add to history  
 messageHistory.push\_back(formattedMessage);  
 messageCount++;  
   
 // In a real logger, we would write to a file here  
 // ofstream logFile(logFilePath, ios::app);  
 // logFile << formattedMessage << endl;  
 }  
   
 // Non-static method that uses the component name from the instance  
 void logMessage(LogLevel level, const string& message) {  
 log(componentName, level, message);  
 }  
   
 // Static methods to get statistics  
 static int getMessageCount() { return messageCount; }  
 static vector<string> getRecentMessages(int count) {  
 vector<string> recent;  
 int start = max(0, static\_cast<int>(messageHistory.size()) - count);  
 for (size\_t i = start; i < messageHistory.size(); i++) {  
 recent.push\_back(messageHistory[i]);  
 }  
 return recent;  
 }  
};  
  
// Define static members  
int Logger::messageCount = 0;  
bool Logger::debugMode = false;  
string Logger::logFilePath = "app.log";  
vector<string> Logger::messageHistory;  
  
int main() {  
 // Configure the logger  
 Logger::configure(true, "application.log");  
   
 // Use static logging methods directly  
 Logger::log("Main", Logger::INFO, "Application started");  
 Logger::log("Main", Logger::DEBUG, "Debug message from main");  
   
 // Create logger instances for different components  
 Logger networkLogger("Network");  
 Logger dbLogger("Database");  
   
 // Use instance methods  
 networkLogger.logMessage(Logger::WARNING, "Network latency high");  
 dbLogger.logMessage(Logger::ERROR, "Failed to connect to database");  
   
 // Get statistics using static methods  
 cout << "\nTotal log messages: " << Logger::getMessageCount() << endl;  
   
 cout << "\nRecent messages:" << endl;  
 vector<string> recentLogs = Logger::getRecentMessages(3);  
 for (const auto& log : recentLogs) {  
 cout << log << endl;  
 }  
   
 return 0;  
}

### Static Initialization

Static member initialization can be tricky, especially with complex data types or interdependencies:

#include <iostream>  
#include <map>  
#include <string>  
using namespace std;  
  
class Configuration {  
private:  
 // Static const members can be initialized inline  
 static const int MAX\_CONNECTIONS = 100;  
 static const bool DEFAULT\_LOGGING = true;  
   
 // Non-const static members must be initialized outside the class  
 static string appName;  
 static map<string, string> settings;  
   
public:  
 static void initialize(const string& name) {  
 appName = name;  
 settings["logging"] = DEFAULT\_LOGGING ? "enabled" : "disabled";  
 settings["max\_connections"] = to\_string(MAX\_CONNECTIONS);  
 settings["theme"] = "default";  
 }  
   
 static void setSetting(const string& key, const string& value) {  
 settings[key] = value;  
 }  
   
 static string getSetting(const string& key) {  
 auto it = settings.find(key);  
 return it != settings.end() ? it->second : "";  
 }  
   
 static void displayConfig() {  
 cout << "Application: " << appName << endl;  
 cout << "Settings:" << endl;  
 for (const auto& [key, value] : settings) {  
 cout << " " << key << ": " << value << endl;  
 }  
 }  
};  
  
// Static member initialization  
string Configuration::appName = "Unnamed App";  
map<string, string> Configuration::settings;  
  
class ApplicationModule {  
private:  
 string moduleName;  
   
public:  
 ApplicationModule(const string& name) : moduleName(name) {  
 cout << "Module initialized: " << moduleName << endl;  
   
 // Access application configuration (static members of another class)  
 string logSetting = Configuration::getSetting("logging");  
 cout << "Logging is " << logSetting << " for module " << moduleName << endl;  
 }  
   
 void run() {  
 cout << "Running module: " << moduleName << endl;  
 }  
};  
  
int main() {  
 // Initialize configuration before using modules  
 Configuration::initialize("My Application");  
 Configuration::setSetting("database", "mysql://localhost:3306/mydb");  
   
 // Display configuration  
 Configuration::displayConfig();  
   
 // Create application modules  
 ApplicationModule module1("Core");  
 ApplicationModule module2("UI");  
   
 // Run modules  
 module1.run();  
 module2.run();  
   
 return 0;  
}

### Singleton Pattern Using Static Members

The Singleton pattern ensures that a class has only one instance and provides a global access point to that instance:

#include <iostream>  
#include <string>  
using namespace std;  
  
class DatabaseConnection {  
private:  
 // Private static instance of the class  
 static DatabaseConnection\* instance;  
   
 // Connection details  
 string host;  
 string username;  
 string password;  
 bool connected;  
   
 // Private constructor to prevent direct instantiation  
 DatabaseConnection() : host("localhost"), username(""), password(""), connected(false) {  
 cout << "Database connection object created" << endl;  
 }  
   
 // Prevent copying and assignment  
 DatabaseConnection(const DatabaseConnection&) = delete;  
 DatabaseConnection& operator=(const DatabaseConnection&) = delete;  
   
public:  
 // Static method to get the singleton instance  
 static DatabaseConnection\* getInstance() {  
 if (instance == nullptr) {  
 instance = new DatabaseConnection();  
 }  
 return instance;  
 }  
   
 // Static method to release the instance  
 static void releaseInstance() {  
 if (instance != nullptr) {  
 delete instance;  
 instance = nullptr;  
 cout << "Database connection instance released" << endl;  
 }  
 }  
   
 // Connection methods  
 bool connect(const string& h, const string& user, const string& pass) {  
 host = h;  
 username = user;  
 password = pass;  
   
 // Simulate connection  
 cout << "Connecting to " << host << " as " << username << "..." << endl;  
 connected = true;  
 return connected;  
 }  
   
 void disconnect() {  
 if (connected) {  
 cout << "Disconnecting from " << host << endl;  
 connected = false;  
 }  
 }  
   
 bool isConnected() const {  
 return connected;  
 }  
   
 // Query execution method  
 void executeQuery(const string& query) {  
 if (connected) {  
 cout << "Executing on " << host << ": " << query << endl;  
 } else {  
 cout << "Cannot execute query: Not connected" << endl;  
 }  
 }  
};  
  
// Initialize static member  
DatabaseConnection\* DatabaseConnection::instance = nullptr;  
  
// Function that uses the database connection  
void performDatabaseOperations() {  
 // Get the singleton instance  
 DatabaseConnection\* db = DatabaseConnection::getInstance();  
   
 // Use the connection  
 if (!db->isConnected()) {  
 db->connect("database.server.com", "admin", "secure\_password");  
 }  
   
 db->executeQuery("SELECT \* FROM users");  
 db->executeQuery("UPDATE products SET stock = stock - 1 WHERE id = 42");  
}  
  
int main() {  
 cout << "Program started" << endl;  
   
 // First use of database - instance will be created  
 performDatabaseOperations();  
   
 // Second use of database - same instance will be used  
 DatabaseConnection\* db = DatabaseConnection::getInstance();  
 db->executeQuery("SELECT \* FROM orders WHERE status = 'pending'");  
   
 // Clean up  
 DatabaseConnection::releaseInstance();  
   
 cout << "Program ended" << endl;  
 return 0;  
}

### Thread Safety Considerations with Static Members

Static members can cause issues in multi-threaded applications. Here’s a thread-safe singleton:

#include <iostream>  
#include <string>  
#include <mutex>  
#include <thread>  
using namespace std;  
  
class ThreadSafeSingleton {  
private:  
 static ThreadSafeSingleton\* instance;  
 static mutex instanceMutex;  
 string data;  
   
 // Private constructor  
 ThreadSafeSingleton() : data("Initial value") {  
 cout << "Singleton constructed" << endl;  
 }  
   
 // Prevent copying  
 ThreadSafeSingleton(const ThreadSafeSingleton&) = delete;  
 ThreadSafeSingleton& operator=(const ThreadSafeSingleton&) = delete;  
   
public:  
 // Thread-safe access to singleton instance  
 static ThreadSafeSingleton\* getInstance() {  
 // Double-checked locking pattern  
 if (instance == nullptr) {  
 lock\_guard<mutex> lock(instanceMutex);  
 if (instance == nullptr) {  
 instance = new ThreadSafeSingleton();  
 }  
 }  
 return instance;  
 }  
   
 // Thread-safe data manipulation  
 void setData(const string& newData) {  
 lock\_guard<mutex> lock(instanceMutex);  
 data = newData;  
 cout << "Data set to: " << data << endl;  
 }  
   
 string getData() {  
 lock\_guard<mutex> lock(instanceMutex);  
 return data;  
 }  
   
 static void destroyInstance() {  
 lock\_guard<mutex> lock(instanceMutex);  
 if (instance != nullptr) {  
 delete instance;  
 instance = nullptr;  
 cout << "Singleton destroyed" << endl;  
 }  
 }  
};  
  
// Initialize static members  
ThreadSafeSingleton\* ThreadSafeSingleton::instance = nullptr;  
mutex ThreadSafeSingleton::instanceMutex;  
  
// Worker function for threads  
void workerFunction(int id, const string& message) {  
 cout << "Thread " << id << " running..." << endl;  
   
 // Get singleton instance  
 ThreadSafeSingleton\* singleton = ThreadSafeSingleton::getInstance();  
   
 // Read current data  
 string currentData = singleton->getData();  
 cout << "Thread " << id << " read: " << currentData << endl;  
   
 // Simulate some work  
 this\_thread::sleep\_for(chrono::milliseconds(100 \* id));  
   
 // Update data  
 singleton->setData(message + " from thread " + to\_string(id));  
   
 cout << "Thread " << id << " finished" << endl;  
}  
  
int main() {  
 cout << "Main thread started" << endl;  
   
 // Create multiple threads that use the singleton  
 thread t1(workerFunction, 1, "Hello");  
 thread t2(workerFunction, 2, "Bonjour");  
 thread t3(workerFunction, 3, "Hola");  
   
 // Join threads  
 t1.join();  
 t2.join();  
 t3.join();  
   
 // Final value  
 cout << "Final data: " << ThreadSafeSingleton::getInstance()->getData() << endl;  
   
 // Clean up  
 ThreadSafeSingleton::destroyInstance();  
   
 cout << "Main thread ended" << endl;  
 return 0;  
}

### Best Practices for Static Members

1. **Use static data members for:**
   * Class-wide constants
   * Shared resources among all instances
   * Counters or statistics
   * Configuration settings
2. **Use static member functions for:**
   * Operations that don’t require object state
   * Factory methods
   * Global access points (carefully)
   * Utility operations related to the class
3. **Follow these guidelines:**
   * Initialize static data members outside the class definition
   * Keep thread safety in mind for static members
   * Don’t overuse static members as global variables
   * Document the purpose of static members
   * Consider alternatives to static members for complex scenarios

## 6.11 Object Slicing

Object slicing occurs when an object of a derived class is assigned to a base class object (not a pointer or reference). When this happens, the derived part of the object is “sliced off,” leaving only the base class portion.

### Basic Example of Object Slicing

#include <iostream>  
#include <string>  
using namespace std;  
  
class Base {  
protected:  
 string name;  
   
public:  
 Base(const string& n) : name(n) {}  
   
 virtual void display() const {  
 cout << "Base class: " << name << endl;  
 }  
   
 string getName() const { return name; }  
};  
  
class Derived : public Base {  
private:  
 int extraData;  
   
public:  
 Derived(const string& n, int extra) : Base(n), extraData(extra) {}  
   
 void display() const override {  
 cout << "Derived class: " << name << ", Extra data: " << extraData << endl;  
 }  
   
 int getExtraData() const { return extraData; }  
};  
  
int main() {  
 Derived derived("Object1", 100);  
   
 // No slicing - pointer points to Derived object  
 Base\* basePtr = &derived;  
 basePtr->display(); // Calls Derived::display()  
   
 // No slicing - reference to Derived object  
 Base& baseRef = derived;  
 baseRef.display(); // Calls Derived::display()  
   
 // Object slicing happens here!  
 Base baseObj = derived; // Only Base part is copied  
 baseObj.display(); // Calls Base::display()  
   
 // We can't access derived class members through the sliced object  
 // cout << baseObj.getExtraData(); // Error: Base has no getExtraData()  
   
 return 0;  
}

### Where Object Slicing Occurs

1. **During Assignment**:

Derived derived("Derived object", 42);  
Base base = derived; // Slicing occurs

1. **Passing by Value to Functions**:

void processBase(Base baseObj) { // Slicing occurs if a Derived is passed  
 baseObj.display(); // Always calls Base::display()  
}  
  
// Usage  
Derived derived("Test", 100);  
processBase(derived); // Derived object is sliced to Base

1. **Returning by Value from Functions**:

Base getObject() {  
 Derived derived("From function", 200);  
 return derived; // Slicing occurs  
}

1. **Storing in Containers of Base Objects**:

vector<Base> objects;  
objects.push\_back(derived); // Slicing occurs

### Issues Caused by Object Slicing

1. **Loss of Derived Class Data**:
   * Derived class members are lost during slicing
   * Only base class members remain
2. **Loss of Polymorphic Behavior**:
   * Virtual function calls no longer dispatch to derived implementations
   * Always call the base class versions
3. **Unexpected Behavior**:
   * Code that expects full objects may not work correctly
   * Can lead to subtle bugs that are hard to track

### Detailed Example Showing the Problems

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
class Shape {  
protected:  
 string name;  
   
public:  
 Shape(const string& n) : name(n) {}  
   
 virtual double area() const {  
 return 0.0;  
 }  
   
 virtual double perimeter() const {  
 return 0.0;  
 }  
   
 virtual void display() const {  
 cout << "Shape: " << name << endl;  
 }  
   
 string getName() const { return name; }  
};  
  
class Circle : public Shape {  
private:  
 double radius;  
   
public:  
 Circle(const string& n, double r) : Shape(n), radius(r) {}  
   
 double area() const override {  
 return 3.14159 \* radius \* radius;  
 }  
   
 double perimeter() const override {  
 return 2 \* 3.14159 \* radius;  
 }  
   
 void display() const override {  
 cout << "Circle: " << name << ", Radius: " << radius << endl;  
 cout << " Area: " << area() << endl;  
 cout << " Perimeter: " << perimeter() << endl;  
 }  
   
 double getRadius() const { return radius; }  
};  
  
class Rectangle : public Shape {  
private:  
 double width;  
 double height;  
   
public:  
 Rectangle(const string& n, double w, double h)   
 : Shape(n), width(w), height(h) {}  
   
 double area() const override {  
 return width \* height;  
 }  
   
 double perimeter() const override {  
 return 2 \* (width + height);  
 }  
   
 void display() const override {  
 cout << "Rectangle: " << name << ", Width: " << width   
 << ", Height: " << height << endl;  
 cout << " Area: " << area() << endl;  
 cout << " Perimeter: " << perimeter() << endl;  
 }  
   
 double getWidth() const { return width; }  
 double getHeight() const { return height; }  
};  
  
// Functions demonstrating different ways of passing objects  
void displayByValue(Shape shape) {  
 cout << "Display by value (slicing occurs): " << endl;  
 shape.display(); // Always calls Shape::display()  
 cout << endl;  
}  
  
void displayByPointer(Shape\* shape) {  
 cout << "Display by pointer (no slicing): " << endl;  
 shape->display(); // Calls appropriate derived class display()  
 cout << endl;  
}  
  
void displayByReference(const Shape& shape) {  
 cout << "Display by reference (no slicing): " << endl;  
 shape.display(); // Calls appropriate derived class display()  
 cout << endl;  
}  
  
int main() {  
 Circle circle("Circle1", 5.0);  
 Rectangle rectangle("Rectangle1", 4.0, 6.0);  
   
 cout << "Original objects:" << endl;  
 circle.display();  
 cout << endl;  
 rectangle.display();  
 cout << endl;  
   
 // Demonstrate slicing with assignment  
 cout << "Slicing with assignment:" << endl;  
 Shape shape1 = circle;  
 Shape shape2 = rectangle;  
 shape1.display(); // Only Shape::display() is called  
 shape2.display(); // Only Shape::display() is called  
 cout << endl;  
   
 // Demonstrate function calls with different parameter types  
 displayByValue(circle); // Slicing occurs  
 displayByPointer(&circle); // No slicing  
 displayByReference(circle); // No slicing  
   
 // Demonstrate slicing in container  
 cout << "Slicing in container:" << endl;  
 vector<Shape> shapes;  
 shapes.push\_back(circle); // Slicing occurs  
 shapes.push\_back(rectangle); // Slicing occurs  
   
 for (const auto& shape : shapes) {  
 shape.display(); // Always calls Shape::display()  
 }  
 cout << endl;  
   
 // Contrast with container of pointers (no slicing)  
 cout << "Container of pointers (no slicing):" << endl;  
 vector<Shape\*> shapePointers;  
 shapePointers.push\_back(&circle);  
 shapePointers.push\_back(&rectangle);  
   
 for (const auto& shapePtr : shapePointers) {  
 shapePtr->display(); // Calls appropriate derived class display()  
 }  
   
 return 0;  
}

### Avoiding Object Slicing

1. **Use Pointers or References**:

void processShape(Shape& shape) { // Use reference  
 shape.display(); // Will call appropriate derived version  
}  
  
void processShapePtr(Shape\* shape) { // Use pointer  
 if (shape) {  
 shape->display(); // Will call appropriate derived version  
 }  
}

1. **Use Smart Pointers for Collections**:

#include <memory>  
#include <vector>  
  
vector<unique\_ptr<Shape>> shapes;  
shapes.push\_back(make\_unique<Circle>("Circle1", 5.0));  
shapes.push\_back(make\_unique<Rectangle>("Rectangle1", 4.0, 6.0));  
  
for (const auto& shape : shapes) {  
 shape->display(); // No slicing, correct polymorphic behavior  
}

1. **Make Base Class Abstract**:

class Shape {  
public:  
 virtual ~Shape() {}  
 virtual double area() const = 0; // Pure virtual function  
 // ...  
};

1. **Disable Slicing with Delete**:

class Shape {  
public:  
 Shape(const Shape&) = delete; // Delete copy constructor  
 Shape& operator=(const Shape&) = delete; // Delete copy assignment  
 // ...  
};

1. **Clone Pattern for Object Copying**:

class Shape {  
public:  
 // ...  
 virtual Shape\* clone() const = 0; // Virtual constructor pattern  
};  
  
class Circle : public Shape {  
public:  
 // ...  
 Shape\* clone() const override {  
 return new Circle(\*this); // Create a full copy  
 }  
};

### Using the Clone Pattern in Detail

#include <iostream>  
#include <string>  
#include <vector>  
#include <memory>  
using namespace std;  
  
class Shape {  
protected:  
 string name;  
   
public:  
 Shape(const string& n) : name(n) {}  
 virtual ~Shape() {}  
   
 virtual double area() const = 0;  
 virtual double perimeter() const = 0;  
 virtual void display() const = 0;  
   
 string getName() const { return name; }  
   
 // Virtual clone method  
 virtual unique\_ptr<Shape> clone() const = 0;  
};  
  
class Circle : public Shape {  
private:  
 double radius;  
   
public:  
 Circle(const string& n, double r) : Shape(n), radius(r) {}  
   
 double area() const override {  
 return 3.14159 \* radius \* radius;  
 }  
   
 double perimeter() const override {  
 return 2 \* 3.14159 \* radius;  
 }  
   
 void display() const override {  
 cout << "Circle: " << name << ", Radius: " << radius << endl;  
 cout << " Area: " << area() << endl;  
 cout << " Perimeter: " << perimeter() << endl;  
 }  
   
 double getRadius() const { return radius; }  
   
 // Implement clone for Circle  
 unique\_ptr<Shape> clone() const override {  
 return make\_unique<Circle>(\*this);  
 }  
};  
  
class Rectangle : public Shape {  
private:  
 double width;  
 double height;  
   
public:  
 Rectangle(const string& n, double w, double h)   
 : Shape(n), width(w), height(h) {}  
   
 double area() const override {  
 return width \* height;  
 }  
   
 double perimeter() const override {  
 return 2 \* (width + height);  
 }  
   
 void display() const override {  
 cout << "Rectangle: " << name << ", Width: " << width   
 << ", Height: " << height << endl;  
 cout << " Area: " << area() << endl;  
 cout << " Perimeter: " << perimeter() << endl;  
 }  
   
 double getWidth() const { return width; }  
 double getHeight() const { return height; }  
   
 // Implement clone for Rectangle  
 unique\_ptr<Shape> clone() const override {  
 return make\_unique<Rectangle>(\*this);  
 }  
};  
  
class ShapeCollection {  
private:  
 vector<unique\_ptr<Shape>> shapes;  
   
public:  
 void addShape(const Shape& shape) {  
 // Use clone to add a full copy of the shape  
 shapes.push\_back(shape.clone());  
 }  
   
 void displayAll() const {  
 for (const auto& shape : shapes) {  
 shape->display();  
 cout << endl;  
 }  
 }  
   
 double getTotalArea() const {  
 double total = 0.0;  
 for (const auto& shape : shapes) {  
 total += shape->area();  
 }  
 return total;  
 }  
};  
  
int main() {  
 Circle circle("Circle1", 5.0);  
 Rectangle rectangle("Rectangle1", 4.0, 6.0);  
   
 // Create a collection and add shapes  
 ShapeCollection collection;  
 collection.addShape(circle); // Adds a clone (no slicing)  
 collection.addShape(rectangle); // Adds a clone (no slicing)  
   
 // Display all shapes (polymorphic behavior preserved)  
 cout << "Shapes in collection:" << endl;  
 collection.displayAll();  
   
 cout << "Total area of all shapes: " << collection.getTotalArea() << endl;  
   
 return 0;  
}

### Best Practices to Avoid Object Slicing

1. **Never pass polymorphic objects by value**; use references or pointers instead
2. **Store collections of polymorphic objects as pointers or smart pointers**
3. **Use the clone pattern** for deep copying of polymorphic objects
4. **Consider making base classes abstract** to prevent instantiation and slicing
5. **Use final for classes not meant to be derived from** to prevent unexpected polymorphic behavior
6. **Make copy constructors private or deleted** in base classes if slicing would be problematic
7. **Understand when polymorphism is actually needed** and when simple value semantics suffice
8. **Document slicing risks** in classes where it might occur

### Recap of Key Object-Oriented Concepts (Topics 6.7-6.11)

1. **Encapsulation**
   * Bundle data with methods that operate on that data
   * Control access through public interfaces
   * Hide implementation details
2. **Abstraction**
   * Hide complexity by showing only essential features
   * Abstract classes define interfaces without implementation details
   * Focus on what an object does, not how it does it
3. **this Pointer**
   * Self-reference to the current object
   * Used to disambiguate variables, enable method chaining, etc.
   * Not available in static member functions
4. **Friend Functions/Classes**
   * Allow specific external functions or classes to access private members
   * Break encapsulation in controlled ways
   * Should be used sparingly
5. **Static Members**
   * Belong to the class itself rather than instances
   * Shared by all objects of the class
   * Can be accessed without creating objects
6. **Object Slicing**
   * Loss of derived class information when assigning to base class objects
   * Prevents polymorphic behavior
   * Avoided using references, pointers, or the clone pattern

These concepts work together to create well-organized, maintainable object-oriented systems. Understanding them thoroughly allows you to design classes that are robust, reusable, and correctly implement the appropriate relationships between objects.

# Chapter 7: Advanced C++ Features (Part 1)

## 7.1 Templates (Function and Class)

Templates are one of the most powerful features in C++, enabling generic programming by allowing you to write code that works with any data type. They provide a way to create functions and classes that operate on different types without having to rewrite the same code for each type.

### Function Templates

Function templates allow you to define a function that can work with different types of parameters.

#### Basic Syntax

template <typename T> // 'class' can be used instead of 'typename'  
T functionName(T parameter) {  
 // Function body  
 return parameter;  
}

#### Simple Example

#include <iostream>  
using namespace std;  
  
// A template function to find the maximum of two values  
template <typename T>  
T findMax(T a, T b) {  
 return (a > b) ? a : b;  
}  
  
int main() {  
 // Using the template with different types  
 cout << "Max of 3 and 7: " << findMax(3, 7) << endl;  
 cout << "Max of 3.5 and 7.2: " << findMax(3.5, 7.2) << endl;  
 cout << "Max of 'a' and 'z': " << findMax('a', 'z') << endl;  
 cout << "Max of \"apple\" and \"zebra\": " << findMax(string("apple"), string("zebra")) << endl;  
   
 return 0;  
}

Output:

Max of 3 and 7: 7  
Max of 3.5 and 7.2: 7.2  
Max of 'a' and 'z': z  
Max of "apple" and "zebra": zebra

#### Multiple Template Parameters

You can define templates with multiple type parameters:

#include <iostream>  
using namespace std;  
  
// Template with multiple parameters  
template <typename T, typename U>  
T convert(U value) {  
 return static\_cast<T>(value);  
}  
  
int main() {  
 // Converting between different types  
 int i = convert<int>(3.14);  
 double d = convert<double>(42);  
 char c = convert<char>(65); // ASCII for 'A'  
   
 cout << "double to int: " << i << endl;  
 cout << "int to double: " << d << endl;  
 cout << "int to char: " << c << endl;  
   
 return 0;  
}

Output:

double to int: 3  
int to double: 42  
int to char: A

#### Template Specialization

Sometimes you need to handle specific types differently. Template specialization allows you to provide a different implementation for a particular type:

#include <iostream>  
#include <cstring>  
using namespace std;  
  
// General template for any type  
template <typename T>  
bool areEqual(T a, T b) {  
 return a == b;  
}  
  
// Specialization for C-style strings  
template <>  
bool areEqual<const char\*>(const char\* a, const char\* b) {  
 return strcmp(a, b) == 0;  
}  
  
int main() {  
 // Using the general template  
 cout << "Are 10 and 10 equal? " << areEqual(10, 10) << endl; // 1 (true)  
 cout << "Are 10 and 20 equal? " << areEqual(10, 20) << endl; // 0 (false)  
   
 // Using the specialized template for C-strings  
 const char\* str1 = "hello";  
 const char\* str2 = "hello";  
 const char\* str3 = "world";  
   
 cout << "Are \"" << str1 << "\" and \"" << str2 << "\" equal? "   
 << areEqual(str1, str2) << endl; // 1 (true)  
 cout << "Are \"" << str1 << "\" and \"" << str3 << "\" equal? "   
 << areEqual(str1, str3) << endl; // 0 (false)  
   
 return 0;  
}

### Class Templates

Class templates allow you to define classes that can work with different data types.

#### Basic Syntax

template <typename T>  
class ClassName {  
private:  
 T member;  
   
public:  
 void setMember(T value) {  
 member = value;  
 }  
   
 T getMember() {  
 return member;  
 }  
};

#### Simple Container Example

#include <iostream>  
#include <vector>  
using namespace std;  
  
template <typename T>  
class Stack {  
private:  
 vector<T> elements;  
   
public:  
 void push(const T& element) {  
 elements.push\_back(element);  
 }  
   
 T pop() {  
 if (elements.empty()) {  
 throw runtime\_error("Stack underflow");  
 }  
   
 T top = elements.back();  
 elements.pop\_back();  
 return top;  
 }  
   
 bool isEmpty() const {  
 return elements.empty();  
 }  
   
 size\_t size() const {  
 return elements.size();  
 }  
   
 void print() const {  
 cout << "Stack contents (top to bottom): ";  
 for (int i = elements.size() - 1; i >= 0; i--) {  
 cout << elements[i] << " ";  
 }  
 cout << endl;  
 }  
};  
  
int main() {  
 // Create an integer stack  
 Stack<int> intStack;  
 intStack.push(10);  
 intStack.push(20);  
 intStack.push(30);  
   
 intStack.print();  
 cout << "Popped: " << intStack.pop() << endl;  
 intStack.print();  
   
 // Create a double stack  
 Stack<double> doubleStack;  
 doubleStack.push(3.14);  
 doubleStack.push(2.71);  
 doubleStack.print();  
   
 // Create a string stack  
 Stack<string> stringStack;  
 stringStack.push("Hello");  
 stringStack.push("World");  
 stringStack.print();  
   
 return 0;  
}

Output:

Stack contents (top to bottom): 30 20 10   
Popped: 30  
Stack contents (top to bottom): 20 10   
Stack contents (top to bottom): 2.71 3.14   
Stack contents (top to bottom): World Hello

#### Class Template with Multiple Parameters

#include <iostream>  
using namespace std;  
  
template <typename KeyType, typename ValueType>  
class Pair {  
private:  
 KeyType key;  
 ValueType value;  
   
public:  
 Pair(const KeyType& k, const ValueType& v) : key(k), value(v) {}  
   
 KeyType getKey() const { return key; }  
 ValueType getValue() const { return value; }  
   
 void setKey(const KeyType& k) { key = k; }  
 void setValue(const ValueType& v) { value = v; }  
   
 void display() const {  
 cout << key << " => " << value << endl;  
 }  
};  
  
int main() {  
 // Pair with integer key and string value  
 Pair<int, string> student(101, "Alice");  
 student.display();  
   
 // Pair with string key and double value  
 Pair<string, double> temperature("Celsius", 25.5);  
 temperature.display();  
   
 // Modify values  
 student.setValue("Bob");  
 student.display();  
   
 return 0;  
}

Output:

101 => Alice  
Celsius => 25.5  
101 => Bob

#### Class Template Specialization

Like function templates, class templates can also be specialized for specific types:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Primary template  
template <typename T>  
class DataHandler {  
private:  
 T data;  
   
public:  
 DataHandler(T d) : data(d) {}  
   
 void process() {  
 cout << "Processing generic data: " << data << endl;  
 }  
};  
  
// Full specialization for int  
template <>  
class DataHandler<int> {  
private:  
 int data;  
   
public:  
 DataHandler(int d) : data(d) {}  
   
 void process() {  
 cout << "Processing integer: " << data;  
 cout << " (square: " << data \* data << ")" << endl;  
 }  
};  
  
// Full specialization for string  
template <>  
class DataHandler<string> {  
private:  
 string data;  
   
public:  
 DataHandler(string d) : data(d) {}  
   
 void process() {  
 cout << "Processing string: " << data;  
 cout << " (length: " << data.length() << ")" << endl;  
 }  
};  
  
int main() {  
 DataHandler<int> intHandler(42);  
 DataHandler<double> doubleHandler(3.14);  
 DataHandler<string> stringHandler("Hello, templates!");  
 DataHandler<char> charHandler('X');  
   
 intHandler.process();  
 doubleHandler.process();  
 stringHandler.process();  
 charHandler.process();  
   
 return 0;  
}

Output:

Processing integer: 42 (square: 1764)  
Processing generic data: 3.14  
Processing string: Hello, templates! (length: 17)  
Processing generic data: X

### Non-Type Template Parameters

Templates can also have non-type parameters, such as integers, pointers, or references:

#include <iostream>  
#include <array>  
using namespace std;  
  
template <typename T, size\_t SIZE>  
class FixedArray {  
private:  
 array<T, SIZE> data;  
   
public:  
 FixedArray() {  
 // Initialize elements to default values  
 for (size\_t i = 0; i < SIZE; ++i) {  
 data[i] = T();  
 }  
 }  
   
 T& operator[](size\_t index) {  
 if (index >= SIZE) {  
 throw out\_of\_range("Index out of bounds");  
 }  
 return data[index];  
 }  
   
 const T& operator[](size\_t index) const {  
 if (index >= SIZE) {  
 throw out\_of\_range("Index out of bounds");  
 }  
 return data[index];  
 }  
   
 size\_t size() const {  
 return SIZE;  
 }  
   
 void fill(const T& value) {  
 data.fill(value);  
 }  
};  
  
int main() {  
 // Create a fixed-size array of 5 integers  
 FixedArray<int, 5> intArray;  
   
 // Set values  
 for (size\_t i = 0; i < intArray.size(); ++i) {  
 intArray[i] = i \* 10;  
 }  
   
 // Display values  
 cout << "Integer array: ";  
 for (size\_t i = 0; i < intArray.size(); ++i) {  
 cout << intArray[i] << " ";  
 }  
 cout << endl;  
   
 // Create a fixed-size array of 3 strings  
 FixedArray<string, 3> strArray;  
 strArray[0] = "Hello";  
 strArray[1] = "Template";  
 strArray[2] = "World";  
   
 // Display values  
 cout << "String array: ";  
 for (size\_t i = 0; i < strArray.size(); ++i) {  
 cout << strArray[i] << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

Output:

Integer array: 0 10 20 30 40   
String array: Hello Template World

### Template Template Parameters

Template template parameters allow you to pass a template as a parameter to another template:

#include <iostream>  
#include <vector>  
#include <list>  
#include <deque>  
using namespace std;  
  
// Template template parameter  
template <typename T, template <typename, typename = allocator<T>> class Container>  
class DataCollection {  
private:  
 Container<T> data;  
   
public:  
 void add(const T& item) {  
 data.push\_back(item);  
 }  
   
 void display() const {  
 for (const T& item : data) {  
 cout << item << " ";  
 }  
 cout << endl;  
 }  
};  
  
int main() {  
 // Using vector as container  
 DataCollection<int, vector> vecCollection;  
 vecCollection.add(10);  
 vecCollection.add(20);  
 vecCollection.add(30);  
   
 cout << "Vector collection: ";  
 vecCollection.display();  
   
 // Using list as container  
 DataCollection<string, list> listCollection;  
 listCollection.add("Hello");  
 listCollection.add("World");  
   
 cout << "List collection: ";  
 listCollection.display();  
   
 // Using deque as container  
 DataCollection<double, deque> dequeCollection;  
 dequeCollection.add(3.14);  
 dequeCollection.add(2.71);  
   
 cout << "Deque collection: ";  
 dequeCollection.display();  
   
 return 0;  
}

Output:

Vector collection: 10 20 30   
List collection: Hello World   
Deque collection: 3.14 2.71

### Variadic Templates (C++11)

Variadic templates allow you to define templates that take a variable number of arguments:

#include <iostream>  
using namespace std;  
  
// Base case - no parameters  
void print() {  
 cout << endl;  
}  
  
// Variadic template function  
template <typename T, typename... Args>  
void print(T first, Args... rest) {  
 cout << first;  
 if (sizeof...(rest) > 0) {  
 cout << ", ";  
 }  
 print(rest...); // Recursive call with remaining arguments  
}  
  
// Variadic template sum function  
template <typename T>  
T sum(T value) {  
 return value;  
}  
  
template <typename T, typename... Args>  
T sum(T first, Args... rest) {  
 return first + sum(rest...);  
}  
  
int main() {  
 // Call print with different arguments  
 print("Hello", 42, 3.14, 'X', "World");  
   
 // Call sum with integers  
 int sumInt = sum(1, 2, 3, 4, 5);  
 cout << "Sum of integers: " << sumInt << endl;  
   
 // Call sum with doubles  
 double sumDouble = sum(1.1, 2.2, 3.3, 4.4);  
 cout << "Sum of doubles: " << sumDouble << endl;  
   
 return 0;  
}

Output:

Hello, 42, 3.14, X, World  
Sum of integers: 15  
Sum of doubles: 11

### Best Practices for Templates

1. **Use templates for generic algorithms and data structures** that work with multiple types.
2. **Minimize dependencies on the template parameters** - only use operations that are required.
3. **Add constraints to template parameters** using SFINAE (C++11) or concepts (C++20).
4. **Provide clear error messages** when template instantiation fails.
5. **Keep the template code in header files**, as template instantiation occurs at compile time.
6. **Consider template specializations** for better efficiency with specific types.
7. **Comment thoroughly**, as template code can be complex and hard to understand.

### Common Template Mistakes

1. **Assuming too much about template parameters** - specify requirements clearly.
2. **Neglecting to check if operations are valid** for the provided types.
3. **Complex template errors** - start simple and incrementally add complexity.
4. **Forgetting that templates are instantiated at compile time** - all required code must be available.
5. **Overuse of templates** - don’t use them when simple overloading would suffice.

## 7.2 Exception Handling (try, catch, throw)

Exception handling in C++ provides a structured way to detect and handle runtime errors. It separates error-handling code from normal code, making programs more robust and readable.

### Basic Exception Handling

The try-catch-throw mechanism is the foundation of C++ exception handling:

#include <iostream>  
using namespace std;  
  
double divide(double a, double b) {  
 if (b == 0) {  
 throw "Division by zero!"; // Throw an exception  
 }  
 return a / b;  
}  
  
int main() {  
 try {  
 // Code that might throw exceptions  
 cout << "5 / 2 = " << divide(5, 2) << endl;  
 cout << "5 / 0 = " << divide(5, 0) << endl; // This will throw  
 cout << "This line is never executed" << endl;  
 }  
 catch (const char\* message) {  
 // Handle the exception  
 cout << "Error caught: " << message << endl;  
 }  
   
 cout << "Program continues after exception handling" << endl;  
   
 return 0;  
}

Output:

5 / 2 = 2.5  
Error caught: Division by zero!  
Program continues after exception handling

### Multiple catch Blocks

You can have multiple catch blocks to handle different types of exceptions:

#include <iostream>  
#include <stdexcept>  
using namespace std;  
  
// Function that throws different types of exceptions  
void testExceptions(int type) {  
 switch (type) {  
 case 1:  
 throw 100; // Throw an integer  
 case 2:  
 throw 2.5; // Throw a double  
 case 3:  
 throw "C string exception"; // Throw a C-string  
 case 4:  
 throw string("std::string exception"); // Throw an std::string  
 case 5:  
 throw runtime\_error("Runtime error occurred"); // Throw a standard exception  
 }  
}  
  
int main() {  
 for (int i = 1; i <= 6; i++) {  
 cout << "\nTest case " << i << ":" << endl;  
 try {  
 testExceptions(i);  
 cout << "No exception thrown" << endl;  
 }   
 catch (int e) {  
 cout << "Integer exception caught: " << e << endl;  
 }   
 catch (double e) {  
 cout << "Double exception caught: " << e << endl;  
 }   
 catch (const char\* e) {  
 cout << "C-string exception caught: " << e << endl;  
 }   
 catch (const string& e) {  
 cout << "String exception caught: " << e << endl;  
 }   
 catch (const runtime\_error& e) {  
 cout << "Runtime error caught: " << e.what() << endl;  
 }   
 catch (...) {  
 cout << "Unknown exception caught" << endl;  
 }  
 }  
   
 return 0;  
}

Output:

Test case 1:  
Integer exception caught: 100  
  
Test case 2:  
Double exception caught: 2.5  
  
Test case 3:  
C-string exception caught: C string exception  
  
Test case 4:  
String exception caught: std::string exception  
  
Test case 5:  
Runtime error caught: Runtime error occurred  
  
Test case 6:  
No exception thrown

### Standard Exception Classes

C++ provides a hierarchy of standard exception classes in the <stdexcept> header:

#include <iostream>  
#include <stdexcept>  
#include <vector>  
using namespace std;  
  
void demoStandardExceptions() {  
 try {  
 vector<int> vec(5);  
 // Accessing element out of bounds  
 cout << vec.at(10) << endl; // This will throw std::out\_of\_range  
 }   
 catch (const out\_of\_range& e) {  
 cout << "out\_of\_range exception: " << e.what() << endl;  
 }  
   
 try {  
 // Invalid argument  
 throw invalid\_argument("The argument is invalid");  
 }   
 catch (const invalid\_argument& e) {  
 cout << "invalid\_argument exception: " << e.what() << endl;  
 }  
   
 try {  
 // Length error  
 throw length\_error("Length exceeded maximum allowed");  
 }   
 catch (const length\_error& e) {  
 cout << "length\_error exception: " << e.what() << endl;  
 }  
   
 try {  
 // Runtime error  
 throw runtime\_error("A runtime error occurred");  
 }   
 catch (const runtime\_error& e) {  
 cout << "runtime\_error exception: " << e.what() << endl;  
 }  
}  
  
int main() {  
 demoStandardExceptions();  
 return 0;  
}

Output:

out\_of\_range exception: vector::\_M\_range\_check: \_\_n (which is 10) >= this->size() (which is 5)  
invalid\_argument exception: The argument is invalid  
length\_error exception: Length exceeded maximum allowed  
runtime\_error exception: A runtime error occurred

### Common Standard Exceptions

1. **std::exception**: Base class for all standard exceptions
2. **std::logic\_error**: Errors that could be detected at compile time
   * std::invalid\_argument
   * std::domain\_error
   * std::length\_error
   * std::out\_of\_range
3. **std::runtime\_error**: Errors that can only be detected at runtime
   * std::range\_error
   * std::overflow\_error
   * std::underflow\_error

### Creating Custom Exception Classes

You can define your own exception classes by inheriting from std::exception or its derived classes:

#include <iostream>  
#include <stdexcept>  
using namespace std;  
  
// Custom exception class  
class DatabaseException : public runtime\_error {  
private:  
 int errorCode;  
   
public:  
 DatabaseException(const string& message, int code)   
 : runtime\_error(message), errorCode(code) {}  
   
 int getErrorCode() const {  
 return errorCode;  
 }  
};  
  
// Function that throws a custom exception  
void connectToDatabase(const string& connectionString) {  
 if (connectionString.empty()) {  
 throw DatabaseException("Connection string cannot be empty", 1001);  
 }  
   
 if (connectionString.find("password") == string::npos) {  
 throw DatabaseException("Missing password in connection string", 1002);  
 }  
   
 cout << "Connected to database successfully" << endl;  
}  
  
int main() {  
 try {  
 // Test with empty connection string  
 cout << "Test case 1:" << endl;  
 connectToDatabase("");  
 }   
 catch (const DatabaseException& e) {  
 cout << "Database error: " << e.what() << endl;  
 cout << "Error code: " << e.getErrorCode() << endl;  
 }  
   
 try {  
 // Test with invalid connection string  
 cout << "\nTest case 2:" << endl;  
 connectToDatabase("server=localhost;database=testdb;user=admin");  
 }   
 catch (const DatabaseException& e) {  
 cout << "Database error: " << e.what() << endl;  
 cout << "Error code: " << e.getErrorCode() << endl;  
 }  
   
 try {  
 // Test with valid connection string  
 cout << "\nTest case 3:" << endl;  
 connectToDatabase("server=localhost;database=testdb;user=admin;password=secret");  
 }   
 catch (const DatabaseException& e) {  
 cout << "Database error: " << e.what() << endl;  
 cout << "Error code: " << e.getErrorCode() << endl;  
 }  
   
 return 0;  
}

Output:

Test case 1:  
Database error: Connection string cannot be empty  
Error code: 1001  
  
Test case 2:  
Database error: Missing password in connection string  
Error code: 1002  
  
Test case 3:  
Connected to database successfully

### Exception Specifications and noexcept

In modern C++, you can use noexcept to specify that a function doesn’t throw exceptions:

#include <iostream>  
using namespace std;  
  
// Function with noexcept specification  
void safeFunction() noexcept {  
 // This function promises not to throw exceptions  
 cout << "Safe function executing" << endl;  
}  
  
// Function without noexcept specification  
void unsafeFunction(bool throwException) {  
 cout << "Unsafe function executing" << endl;  
 if (throwException) {  
 throw runtime\_error("Exception from unsafeFunction");  
 }  
}  
  
int main() {  
 // Testing noexcept functions  
 try {  
 safeFunction();  
 }   
 catch (...) {  
 cout << "Exception from safeFunction caught (should never happen)" << endl;  
 }  
   
 // Testing functions that might throw  
 try {  
 unsafeFunction(false); // No exception  
 unsafeFunction(true); // Will throw  
 }   
 catch (const exception& e) {  
 cout << "Exception caught: " << e.what() << endl;  
 }  
   
 // Using noexcept operator  
 cout << "\nIs safeFunction() noexcept? " << (noexcept(safeFunction()) ? "Yes" : "No") << endl;  
 cout << "Is unsafeFunction() noexcept? " << (noexcept(unsafeFunction(false)) ? "Yes" : "No") << endl;  
   
 return 0;  
}

Output:

Safe function executing  
Unsafe function executing  
Unsafe function executing  
Exception caught: Exception from unsafeFunction  
  
Is safeFunction() noexcept? Yes  
Is unsafeFunction() noexcept? No

### Stack Unwinding

When an exception is thrown, C++ performs “stack unwinding,” which involves destroying all local objects in reverse order of their creation:

#include <iostream>  
using namespace std;  
  
class Resource {  
private:  
 string name;  
   
public:  
 Resource(const string& n) : name(n) {  
 cout << "Resource acquired: " << name << endl;  
 }  
   
 ~Resource() {  
 cout << "Resource released: " << name << endl;  
 }  
};  
  
void function3() {  
 Resource r3("Function3 Resource");  
 cout << "function3() throwing exception..." << endl;  
 throw runtime\_error("Exception from function3()");  
 cout << "This line will never execute" << endl;  
}  
  
void function2() {  
 Resource r2("Function2 Resource");  
 cout << "function2() calling function3()..." << endl;  
 function3();  
 cout << "This line will never execute" << endl;  
}  
  
void function1() {  
 Resource r1("Function1 Resource");  
 cout << "function1() calling function2()..." << endl;  
 function2();  
 cout << "This line will never execute" << endl;  
}  
  
int main() {  
 try {  
 cout << "main() calling function1()..." << endl;  
 function1();  
 cout << "This line will never execute" << endl;  
 }   
 catch (const exception& e) {  
 cout << "Exception caught in main(): " << e.what() << endl;  
 }  
   
 cout << "Program continues after exception handling" << endl;  
   
 return 0;  
}

Output:

main() calling function1()...  
Resource acquired: Function1 Resource  
function1() calling function2()...  
Resource acquired: Function2 Resource  
function2() calling function3()...  
Resource acquired: Function3 Resource  
function3() throwing exception...  
Resource released: Function3 Resource  
Resource released: Function2 Resource  
Resource released: Function1 Resource  
Exception caught in main(): Exception from function3()  
Program continues after exception handling

### Exception Safety Guarantees

Exception safety refers to how well a program preserves its invariants and prevents resource leaks when exceptions occur:

1. **No-throw guarantee**: The operation will not throw exceptions
2. **Strong guarantee**: If an exception occurs, the program state remains unchanged (as if the operation hadn’t been called)
3. **Basic guarantee**: If an exception occurs, the program is in a valid but unspecified state
4. **No guarantee**: If an exception occurs, the program might be in an invalid state

Example demonstrating exception safety:

#include <iostream>  
#include <vector>  
#include <string>  
#include <memory>  
using namespace std;  
  
class Resource {  
private:  
 string name;  
   
public:  
 Resource(const string& n) : name(n) {  
 cout << "Resource created: " << name << endl;  
 if (name == "BadResource") {  
 throw runtime\_error("Failed to create BadResource");  
 }  
 }  
   
 ~Resource() {  
 cout << "Resource destroyed: " << name << endl;  
 }  
   
 void use() {  
 cout << "Using resource: " << name << endl;  
 }  
};  
  
// Demonstrates basic guarantee (no leaks but state might change)  
class BasicGuarantee {  
private:  
 vector<Resource\*> resources;  
   
public:  
 // This might leak if an exception occurs midway  
 void unsafe\_add(const string& name1, const string& name2) {  
 Resource\* r1 = new Resource(name1);  
 resources.push\_back(r1);  
   
 // If this throws, r1 won't be deleted  
 Resource\* r2 = new Resource(name2);  
 resources.push\_back(r2);  
 }  
   
 // This provides basic guarantee (no leaks)  
 void basic\_add(const string& name1, const string& name2) {  
 Resource\* r1 = nullptr;  
 Resource\* r2 = nullptr;  
   
 try {  
 r1 = new Resource(name1);  
 resources.push\_back(r1);  
   
 r2 = new Resource(name2);  
 resources.push\_back(r2);  
 }   
 catch (...) {  
 // Clean up if exception occurs  
 delete r2; // Safe even if nullptr  
   
 // r1 is already in the vector, so we don't delete it  
 throw; // Rethrow the exception  
 }  
 }  
   
 ~BasicGuarantee() {  
 for (auto resource : resources) {  
 delete resource;  
 }  
 }  
};  
  
// Demonstrates strong guarantee (all-or-nothing)  
class StrongGuarantee {  
private:  
 vector<unique\_ptr<Resource>> resources;  
   
public:  
 // This provides strong guarantee (state unchanged if exception occurs)  
 void strong\_add(const string& name1, const string& name2) {  
 // Temporary vector to hold new resources  
 vector<unique\_ptr<Resource>> temp;  
   
 temp.push\_back(make\_unique<Resource>(name1));  
 temp.push\_back(make\_unique<Resource>(name2));  
   
 // If we get here, both resources were created successfully  
 // Now we can modify our actual state  
 for (auto& resource : temp) {  
 resources.push\_back(move(resource));  
 }  
 }  
   
 void use\_all() {  
 for (const auto& resource : resources) {  
 resource->use();  
 }  
 }  
};  
  
int main() {  
 // Test basic guarantee  
 cout << "Testing basic guarantee:" << endl;  
 try {  
 BasicGuarantee basic;  
 basic.basic\_add("Resource1", "Resource2"); // Should succeed  
 cout << "First add succeeded" << endl;  
   
 basic.basic\_add("Resource3", "BadResource"); // Should throw  
 cout << "Second add succeeded (shouldn't see this)" << endl;  
 }   
 catch (const exception& e) {  
 cout << "Exception caught: " << e.what() << endl;  
 }  
   
 cout << "\nTesting strong guarantee:" << endl;  
 try {  
 StrongGuarantee strong;  
 strong.strong\_add("Resource4", "Resource5"); // Should succeed  
 cout << "First add succeeded" << endl;  
 strong.use\_all();  
   
 strong.strong\_add("Resource6", "BadResource"); // Should throw  
 cout << "Second add succeeded (shouldn't see this)" << endl;  
 }   
 catch (const exception& e) {  
 cout << "Exception caught: " << e.what() << endl;  
 }  
   
 return 0;  
}

### Best Practices for Exception Handling

1. **Use exceptions for exceptional conditions**, not normal flow control.
2. **Catch exceptions by reference** to avoid object slicing and unnecessary copying.
3. **Order catch blocks from most specific to most general** to ensure proper handling.
4. **Clean up resources** using RAII (Resource Acquisition Is Initialization) to prevent leaks.
5. **Keep exception specifications up-to-date** if you use them.
6. **Be specific about which exceptions your code might throw** in documentation.
7. **Don’t let destructors throw** exceptions, as this can lead to program termination.
8. **Use smart pointers and containers** to automate resource management.
9. **Consider exception safety** when designing and implementing functions.

## 7.3 Namespaces

Namespaces provide a way to organize code into logical groups and prevent name conflicts. They allow you to create scope where identifiers (variables, functions, classes) can be placed, ensuring they don’t collide with identifiers in other parts of your code.

### Basic Namespace Syntax

#include <iostream>  
using namespace std;  
  
// Define a namespace  
namespace MyNamespace {  
 // Variables in namespace  
 int value = 100;  
   
 // Functions in namespace  
 void display() {  
 cout << "Value from MyNamespace: " << value << endl;  
 }  
   
 // Classes in namespace  
 class MyClass {  
 public:  
 void show() {  
 cout << "Hello from MyNamespace::MyClass" << endl;  
 }  
 };  
}  
  
int main() {  
 // Accessing namespace members using scope resolution operator ::  
 cout << "Value: " << MyNamespace::value << endl;  
 MyNamespace::display();  
   
 MyNamespace::MyClass obj;  
 obj.show();  
   
 return 0;  
}

Output:

Value: 100  
Value from MyNamespace: 100  
Hello from MyNamespace::MyClass

### The using Directive

The using directive brings all names from a namespace into the current scope:

#include <iostream>  
  
// Define a namespace  
namespace Math {  
 const double PI = 3.14159265358979;  
   
 double square(double x) {  
 return x \* x;  
 }  
   
 double cube(double x) {  
 return x \* x \* x;  
 }  
}  
  
int main() {  
 // Without using  
 std::cout << "PI: " << Math::PI << std::endl;  
 std::cout << "Square of 5: " << Math::square(5) << std::endl;  
   
 // With using directive  
 using namespace Math;  
 std::cout << "Cube of 3: " << cube(3) << std::endl;  
 std::cout << "PI squared: " << square(PI) << std::endl;  
   
 return 0;  
}

Output:

PI: 3.14159  
Square of 5: 25  
Cube of 3: 27  
PI squared: 9.86961

### The using Declaration

The using declaration brings specific names from a namespace into the current scope:

#include <iostream>  
  
namespace Math {  
 const double PI = 3.14159265358979;  
   
 double square(double x) {  
 return x \* x;  
 }  
   
 double cube(double x) {  
 return x \* x \* x;  
 }  
}  
  
namespace Physics {  
 const double G = 9.81; // Gravitational acceleration  
   
 double calculateEnergy(double mass, double height) {  
 return mass \* G \* height;  
 }  
}  
  
int main() {  
 // Using specific names from namespaces  
 using Math::PI;  
 using Math::square;  
 using Physics::G;  
   
 std::cout << "PI: " << PI << std::endl;  
 std::cout << "Square of 4: " << square(4) << std::endl;  
 std::cout << "Gravity: " << G << " m/s²" << std::endl;  
   
 // Need to use full qualification for names not explicitly imported  
 std::cout << "Cube of 3: " << Math::cube(3) << std::endl;  
 std::cout << "Energy: " << Physics::calculateEnergy(10, 5) << " joules" << std::endl;  
   
 return 0;  
}

Output:

PI: 3.14159  
Square of 4: 16  
Gravity: 9.81 m/s²  
Cube of 3: 27  
Energy: 490.5 joules

### Namespace Aliases

Namespace aliases allow you to create shorter names for namespaces:

#include <iostream>  
using namespace std;  
  
namespace VeryLongNamespaceName {  
 void display() {  
 cout << "This is a function in a very long namespace name" << endl;  
 }  
   
 namespace NestedNamespace {  
 void show() {  
 cout << "This is a nested function" << endl;  
 }  
 }  
}  
  
int main() {  
 // Without alias - verbose  
 VeryLongNamespaceName::display();  
 VeryLongNamespaceName::NestedNamespace::show();  
   
 // With alias - more concise  
 namespace Short = VeryLongNamespaceName;  
 Short::display();  
   
 namespace ShortNested = VeryLongNamespaceName::NestedNamespace;  
 ShortNested::show();  
   
 return 0;  
}

Output:

This is a function in a very long namespace name  
This is a nested function  
This is a function in a very long namespace name  
This is a nested function

### Nested Namespaces

Namespaces can be nested inside other namespaces:

#include <iostream>  
using namespace std;  
  
// Parent namespace  
namespace Organization {  
 void info() {  
 cout << "Organization namespace" << endl;  
 }  
   
 // Nested namespace  
 namespace Department {  
 void info() {  
 cout << "Department namespace" << endl;  
 }  
   
 // Further nested namespace  
 namespace Team {  
 void info() {  
 cout << "Team namespace" << endl;  
 }  
 }  
 }  
}  
  
// C++17 nested namespace syntax  
namespace Project::Module::Component {  
 void info() {  
 cout << "Component namespace using C++17 syntax" << endl;  
 }  
}  
  
int main() {  
 // Accessing nested namespaces  
 Organization::info();  
 Organization::Department::info();  
 Organization::Department::Team::info();  
   
 // Using C++17 syntax  
 Project::Module::Component::info();  
   
 // Using aliases for nested namespaces  
 namespace OrgTeam = Organization::Department::Team;  
 OrgTeam::info();  
   
 return 0;  
}

Output:

Organization namespace  
Department namespace  
Team namespace  
Component namespace using C++17 syntax  
Team namespace

### Unnamed/Anonymous Namespaces

Unnamed namespaces provide internal linkage, similar to using the static keyword for global variables and functions:

#include <iostream>  
using namespace std;  
  
// Unnamed namespace - contents have internal linkage  
namespace {  
 int hiddenValue = 100;  
   
 void privateFunction() {  
 cout << "This function is only visible in this file" << endl;  
 }  
}  
  
// Global function that can access unnamed namespace contents  
void accessHiddenValue() {  
 cout << "Hidden value: " << hiddenValue << endl;  
 privateFunction();  
}  
  
int main() {  
 // Can access members of unnamed namespace directly  
 cout << "Hidden value from main: " << hiddenValue << endl;  
 privateFunction();  
   
 accessHiddenValue();  
   
 return 0;  
}

Output:

Hidden value from main: 100  
This function is only visible in this file  
Hidden value: 100  
This function is only visible in this file

### Inline Namespaces (C++11)

Inline namespaces automatically expose their members to the enclosing namespace:

#include <iostream>  
using namespace std;  
  
namespace Library {  
 // Original version  
 namespace v1 {  
 void process() {  
 cout << "Using v1 process function" << endl;  
 }  
 }  
   
 // New version - marked inline to be the default  
 inline namespace v2 {  
 void process() {  
 cout << "Using v2 process function (newer)" << endl;  
 }  
   
 void newFeature() {  
 cout << "This feature is only available in v2" << endl;  
 }  
 }  
}  
  
int main() {  
 // Call v1 explicitly  
 Library::v1::process();  
   
 // Call v2 explicitly  
 Library::v2::process();  
   
 // Call default version (v2, because it's inline)  
 Library::process();  
   
 // Use new feature available only in v2  
 Library::newFeature();  
   
 return 0;  
}

Output:

Using v1 process function  
Using v2 process function (newer)  
Using v2 process function (newer)  
This feature is only available in v2

### Resolving Name Conflicts

Namespaces are primarily used to avoid name conflicts between different libraries or modules:

#include <iostream>  
using namespace std;  
  
// First library  
namespace Graphics {  
 struct Point {  
 int x, y;  
   
 Point(int a, int b) : x(a), y(b) {}  
   
 void display() const {  
 cout << "Graphics::Point(" << x << ", " << y << ")" << endl;  
 }  
 };  
   
 void drawLine(Point start, Point end) {  
 cout << "Drawing line from ";  
 start.display();  
 cout << " to ";  
 end.display();  
 }  
}  
  
// Second library with conflicting names  
namespace Math {  
 struct Point {  
 double x, y;  
   
 Point(double a, double b) : x(a), y(b) {}  
   
 void display() const {  
 cout << "Math::Point(" << x << ", " << y << ")" << endl;  
 }  
 };  
   
 double distance(Point a, Point b) {  
 double dx = b.x - a.x;  
 double dy = b.y - a.y;  
 return sqrt(dx\*dx + dy\*dy);  
 }  
}  
  
int main() {  
 // Using fully qualified names to avoid confusion  
 Graphics::Point p1(10, 20);  
 Graphics::Point p2(50, 60);  
   
 Math::Point mp1(1.5, 2.5);  
 Math::Point mp2(4.5, 5.5);  
   
 // Use functions from different namespaces  
 p1.display();  
 mp1.display();  
   
 Graphics::drawLine(p1, p2);  
 cout << "Distance between points: " << Math::distance(mp1, mp2) << endl;  
   
 return 0;  
}

Output:

Graphics::Point(10, 20)  
Math::Point(1.5, 2.5)  
Drawing line from Graphics::Point(10, 20) to Graphics::Point(50, 60)  
Distance between points: 4.24264

### Argument-Dependent Lookup (ADL)

ADL, also known as Koenig lookup, allows unqualified functions to be found in the namespaces of their arguments:

#include <iostream>  
#include <string>  
using namespace std;  
  
namespace MyLibrary {  
 struct User {  
 string name;  
 int id;  
   
 User(const string& n, int i) : name(n), id(i) {}  
 };  
   
 // Function in the same namespace as the User type  
 void display(const User& user) {  
 cout << "User ID: " << user.id << ", Name: " << user.name << endl;  
 }  
}  
  
int main() {  
 MyLibrary::User user("Alice", 12345);  
   
 // No need for MyLibrary:: prefix due to ADL  
 display(user); // Calls MyLibrary::display  
   
 // Explicit qualification also works  
 MyLibrary::display(user);  
   
 return 0;  
}

Output:

User ID: 12345, Name: Alice  
User ID: 12345, Name: Alice

### Best Practices for Namespaces

1. **Use namespaces to organize related code** and prevent name collisions.
2. **Avoid using namespace std; in header files** as it can lead to name conflicts.
3. **Prefer using declarations over using directives** for better control.
4. **Use meaningful namespace names** that reflect the purpose or project.
5. **Use unnamed namespaces** for internal linkage instead of the static keyword.
6. **Keep namespace hierarchies relatively flat** to avoid overly complex qualification.
7. **Consider using namespace aliases** for long namespace names.
8. **Be aware of argument-dependent lookup** when designing libraries.
9. **Use inline namespaces for versioning** your library APIs.
10. **Document namespace contents** clearly, especially in public APIs.

### Common Namespace Pitfalls

1. **Global using directives** bringing in too many names
2. **Deeply nested namespaces** making code hard to read
3. **Confusing ADL behavior** leading to unexpected function calls
4. **Namespace pollution** from unnecessary using directives
5. **Forgetting to close namespace blocks** properly

### Example of a Well-Structured Namespace System

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
// Main library namespace  
namespace MyApp {  
 // Common utilities  
 namespace Utils {  
 string formatDate(int day, int month, int year) {  
 return to\_string(day) + "/" + to\_string(month) + "/" + to\_string(year);  
 }  
   
 string toUpper(string text) {  
 for (char& c : text) {  
 c = toupper(c);  
 }  
 return text;  
 }  
 }  
   
 // Database functionality  
 namespace DB {  
 struct Record {  
 int id;  
 string name;  
 string date;  
 };  
   
 vector<Record> getRecords() {  
 // Simulated database query  
 return {  
 {1, "Alice", Utils::formatDate(15, 3, 2023)},  
 {2, "Bob", Utils::formatDate(20, 4, 2023)}  
 };  
 }  
   
 void displayRecord(const Record& record) {  
 cout << "Record ID: " << record.id << endl;  
 cout << "Name: " << record.name << endl;  
 cout << "Date: " << record.date << endl;  
 }  
 }  
   
 // User interface  
 namespace UI {  
 void showWelcome() {  
 cout << Utils::toUpper("Welcome to MyApp") << endl;  
 }  
   
 void displayRecords() {  
 cout << "Database Records:" << endl;  
 cout << "----------------" << endl;  
   
 auto records = DB::getRecords();  
 for (const auto& record : records) {  
 DB::displayRecord(record);  
 cout << endl;  
 }  
 }  
 }  
}  
  
int main() {  
 // Using namespace aliases for convenience  
 namespace UI = MyApp::UI;  
   
 UI::showWelcome();  
 UI::displayRecords();  
   
 return 0;  
}

Output:

WELCOME TO MYAPP  
Database Records:  
----------------  
Record ID: 1  
Name: Alice  
Date: 15/3/2023  
  
Record ID: 2  
Name: Bob  
Date: 20/4/2023

# Chapter 7: Advanced C++ Features (Part 2)

## 7.4 Type Inference with auto

Type inference is a powerful feature in modern C++ that allows the compiler to automatically deduce the type of a variable from its initializer expression. This not only simplifies code but also makes it more robust when types change.

### Basic Usage of auto

The auto keyword tells the compiler to automatically deduce the type from the initialization expression:

#include <iostream>  
#include <string>  
#include <vector>  
#include <map>  
using namespace std;  
  
int main() {  
 // Basic types  
 auto i = 42; // int  
 auto d = 3.14; // double  
 auto b = true; // bool  
 auto c = 'a'; // char  
 auto s = "Hello"; // const char\*  
 auto str = string("Hello"); // std::string  
   
 // Print values and their types  
 cout << "i = " << i << " (type deduced as int)" << endl;  
 cout << "d = " << d << " (type deduced as double)" << endl;  
 cout << "b = " << b << " (type deduced as bool)" << endl;  
 cout << "c = " << c << " (type deduced as char)" << endl;  
 cout << "s = " << s << " (type deduced as const char\*)" << endl;  
 cout << "str = " << str << " (type deduced as std::string)" << endl;  
   
 // More complex types  
 vector<int> numbers = {1, 2, 3, 4, 5};  
 auto iter = numbers.begin(); // vector<int>::iterator  
   
 map<string, vector<int>> data;  
 auto it = data.find("key"); // map<string, vector<int>>::iterator  
   
 return 0;  
}

### Type Deduction Rules

When using auto, the compiler applies these type deduction rules:

1. **Reference and const-ness are ignored**
   * const int x = 42;
   * auto y = x; → y becomes int, not const int
2. **To preserve const-ness, use const auto**
   * const int x = 42;
   * const auto y = x; → y becomes const int
3. **To get a reference, use auto& or auto&&**
   * auto& z = x; → z becomes int&

Here’s a code example demonstrating these rules:

#include <iostream>  
#include <typeinfo>  
#include <string>  
using namespace std;  
  
int main() {  
 // Original variables  
 int number = 42;  
 const int constNumber = 100;  
 string name = "John";  
   
 // Basic auto (drops const and references)  
 auto a1 = number; // int  
 auto a2 = constNumber; // int (const is dropped)  
 auto a3 = name; // string (copy)  
   
 // Preserving const with const auto  
 const auto a4 = number; // const int  
 const auto a5 = constNumber; // const int  
   
 // Getting references with auto&  
 auto& a6 = number; // int&  
 auto& a7 = constNumber; // const int&  
   
 // Modifying through references  
 a6 = 50; // Changes 'number' to 50  
 // a7 = 200; // Error: can't modify through const reference  
   
 // Print values  
 cout << "number = " << number << endl;  
 cout << "constNumber = " << constNumber << endl;  
   
 return 0;  
}

### References with auto

The most common mistake with auto is forgetting to use references when needed:

#include <iostream>  
#include <vector>  
using namespace std;  
  
class BigObject {  
private:  
 vector<int> data;  
 string name;  
   
public:  
 BigObject(const string& n, int size) : name(n) {  
 data.resize(size, 0);  
 }  
   
 BigObject(const BigObject& other) : data(other.data), name(other.name) {  
 cout << "Copy constructor: copying " << name << endl;  
 }  
   
 void setName(const string& n) { name = n; }  
 string getName() const { return name; }  
};  
  
void processObject(const BigObject& obj) {  
 cout << "Processing: " << obj.getName() << endl;  
}  
  
int main() {  
 vector<BigObject> objects;  
 objects.push\_back(BigObject("Object 1", 1000));  
 objects.push\_back(BigObject("Object 2", 2000));  
   
 // Bad: creates a copy of each object  
 cout << "\nWithout references:" << endl;  
 for (auto obj : objects) {  
 processObject(obj);  
 }  
   
 // Good: uses references, no copies made  
 cout << "\nWith references:" << endl;  
 for (const auto& obj : objects) {  
 processObject(obj);  
 }  
   
 return 0;  
}

### Using auto with Function Return Types (C++14)

C++14 allows using auto for function return types, which is useful for complex return types or template functions:

#include <iostream>  
#include <vector>  
#include <string>  
using namespace std;  
  
// Function returning auto  
auto add(int a, int b) {  
 return a + b; // Return type is int  
}  
  
// Function returning complex type  
auto getVector() {  
 return vector<int>{1, 2, 3, 4, 5}; // Return type is vector<int>  
}  
  
// Template function with auto return  
template<typename T, typename U>  
auto multiply(T t, U u) {  
 return t \* u; // Return type depends on T and U  
}  
  
int main() {  
 auto result1 = add(5, 3);  
 cout << "5 + 3 = " << result1 << endl;  
   
 auto vec = getVector();  
 cout << "Vector size: " << vec.size() << endl;  
   
 auto result2 = multiply(5, 3.0);  
 cout << "5 \* 3.0 = " << result2 << " (type: double)" << endl;  
   
 return 0;  
}

### Trailing Return Type Syntax

For more complex return types, C++11 introduced the trailing return type syntax:

#include <iostream>  
#include <string>  
#include <map>  
using namespace std;  
  
// Function that returns a reference to an element in a map  
auto findInMap(map<string, int>& m, const string& key) -> int& {  
 return m[key];  
}  
  
// Template function using decltype to determine return type  
template<typename T, typename U>  
auto complexCalc(T t, U u) -> decltype(t \* u + t) {  
 return t \* u + t;  
}  
  
int main() {  
 map<string, int> ages = {  
 {"Alice", 25},  
 {"Bob", 30},  
 {"Charlie", 35}  
 };  
   
 // Find and modify a value in the map  
 auto& age = findInMap(ages, "Bob");  
 cout << "Bob's age: " << age << endl;  
   
 age = 31;  
 cout << "Bob's new age: " << ages["Bob"] << endl;  
   
 // Use complex calculation  
 auto result = complexCalc(5, 3.5);  
 cout << "5 \* 3.5 + 5 = " << result << endl;  
   
 return 0;  
}

### auto with Lambdas

C++14 allows using auto for lambda parameter types:

#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
int main() {  
 // Lambda with explicit parameter types  
 auto lambda1 = [](int x, int y) { return x + y; };  
   
 // Lambda with auto parameters (C++14)  
 auto lambda2 = [](auto x, auto y) { return x + y; };  
   
 cout << "Using lambda1: " << lambda1(5, 3) << endl;  
   
 // Same lambda works with different types  
 cout << "lambda2 with ints: " << lambda2(5, 3) << endl;  
 cout << "lambda2 with doubles: " << lambda2(2.5, 3.5) << endl;  
 cout << "lambda2 with mixed types: " << lambda2(10, 3.5) << endl;  
   
 // Using generic lambda with algorithm  
 vector<int> numbers = {1, 2, 3, 4, 5};  
 transform(numbers.begin(), numbers.end(), numbers.begin(),  
 [](auto x) { return x \* x; });  
   
 cout << "Squared numbers: ";  
 for (auto n : numbers) {  
 cout << n << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### decltype

While auto is used for type deduction from an initializer, decltype is used to deduce the type of an expression without evaluating it:

#include <iostream>  
#include <vector>  
using namespace std;  
  
int main() {  
 int x = 10;  
 const int& rx = x;  
   
 // decltype preserves constness and references  
 decltype(x) y = 20; // y is int  
 decltype(rx) ry = y; // ry is const int&  
   
 // decltype with expression  
 vector<int> vec = {1, 2, 3};  
 decltype(vec[0]) element = vec[1]; // element is int&  
   
 // Special rule: decltype((x)) is always a reference type  
 decltype((x)) z = x; // z is int&  
   
 // Use case: alias template  
 template<typename Container>  
 using ElementType = decltype(declval<Container>()[0]);  
   
 return 0;  
}

### Best Practices with auto

1. **Use auto when the type is obvious or verbose**

* auto it = myMap.find("key"); // Better than std::map<std::string, std::vector<int>>::iterator

1. **Use auto with initialization**

* auto x = 42; // Good - initialized with obvious type  
  auto y; // Error - must be initialized

1. **Use const auto& for read-only references**

* for (const auto& item : container) // Efficient - no copying, read-only

1. **Use auto&& with forwarding references**

* template <typename T>  
  void wrapper(T&& param) {  
   auto&& forwarded = std::forward<T>(param);  
  }

1. **Be cautious with expressions that hide complexity**

* auto result = functionThatReturnsAProxy(); // Type might not be what you expect

1. **Consider readability over brevity**

* auto age = 42; // OK but int age = 42 might be clearer  
  int count = 10; // Explicit is sometimes better

## 7.5 const, mutable, volatile

These three keywords modify how variables and member functions behave with respect to changes and optimization.

### const

The const keyword indicates that a value cannot be modified and communicates intent to both the compiler and other programmers.

#### Basic const Usage

#include <iostream>  
using namespace std;  
  
int main() {  
 // Basic constant variables  
 const int MAX\_USERS = 100;  
 // MAX\_USERS = 200; // Error: assignment of read-only variable  
   
 // Constant reference to variable  
 int value = 42;  
 const int& ref = value;  
 value = 43; // OK - modifying the original variable  
 // ref = 44; // Error - can't modify through const reference  
   
 cout << "value = " << value << ", ref = " << ref << endl;  
   
 return 0;  
}

#### const with Pointers

There are two ways to use const with pointers:

#include <iostream>  
using namespace std;  
  
int main() {  
 int x = 10;  
 int y = 20;  
   
 // Pointer to constant (can't modify what it points to)  
 const int\* p1 = &x;  
 // \*p1 = 20; // Error: can't modify through p1  
 p1 = &y; // OK: can change what p1 points to  
   
 // Constant pointer (can't change what it points to)  
 int\* const p2 = &x;  
 \*p2 = 20; // OK: can modify through p2  
 // p2 = &y; // Error: can't change what p2 points to  
   
 // Constant pointer to constant (can't change pointer or value)  
 const int\* const p3 = &x;  
 // \*p3 = 20; // Error: can't modify through p3  
 // p3 = &y; // Error: can't change what p3 points to  
   
 cout << "x = " << x << ", y = " << y << endl;  
 cout << "\*p1 = " << \*p1 << ", \*p2 = " << \*p2 << ", \*p3 = " << \*p3 << endl;  
   
 return 0;  
}

#### const in Function Parameters

Using const in function parameters prevents modification and allows passing literal values:

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
// Non-const parameter (creates a copy)  
void processValue(int value) {  
 value \*= 2; // Modifies the copy, not the original  
 cout << "Inside processValue: " << value << endl;  
}  
  
// Const reference parameter (no copy, can't modify)  
void processConstRef(const int& value) {  
 // value \*= 2; // Error: can't modify const reference  
 cout << "Inside processConstRef: " << value << endl;  
}  
  
// Const pointer parameter  
void processConstPtr(const int\* ptr) {  
 // \*ptr = 100; // Error: can't modify what ptr points to  
 cout << "Inside processConstPtr: " << \*ptr << endl;  
}  
  
// Function taking constant vector reference  
int sum(const vector<int>& numbers) {  
 int total = 0;  
 for (const auto& num : numbers) {  
 total += num;  
 }  
 return total;  
}  
  
int main() {  
 int x = 10;  
 processValue(x);  
 cout << "After processValue: " << x << endl; // x remains 10  
   
 processConstRef(x);  
 cout << "After processConstRef: " << x << endl; // x remains 10  
   
 processConstPtr(&x);  
 cout << "After processConstPtr: " << x << endl; // x remains 10  
   
 // Can pass literal values to const reference parameters  
 processConstRef(42); // Works because param is const reference  
 // processValue(42); // Also works but less efficient (creates a copy)  
   
 // Using with containers  
 vector<int> numbers = {1, 2, 3, 4, 5};  
 cout << "Sum: " << sum(numbers) << endl;  
   
 return 0;  
}

#### const Member Functions

A const member function promises not to modify the object and can be called on const objects:

#include <iostream>  
#include <string>  
using namespace std;  
  
class Person {  
private:  
 string name;  
 int age;  
 mutable int accessCount; // Can be modified even in const functions  
   
public:  
 Person(const string& n, int a) : name(n), age(a), accessCount(0) {}  
   
 // Non-const function - can modify the object  
 void birthday() {  
 age++;  
 cout << name << " is now " << age << " years old" << endl;  
 }  
   
 // Const function - can't modify the object  
 string getName() const {  
 accessCount++; // OK because accessCount is mutable  
 return name;  
 }  
   
 int getAge() const {  
 accessCount++; // OK because accessCount is mutable  
 return age;  
 }  
   
 int getAccessCount() const {  
 return accessCount;  
 }  
};  
  
void displayPerson(const Person& person) {  
 // Can only call const member functions on const reference  
 cout << person.getName() << " is " << person.getAge() << " years old" << endl;  
 // person.birthday(); // Error: can't call non-const function on const object  
}  
  
int main() {  
 Person alice("Alice", 30);  
   
 alice.birthday(); // OK - alice is non-const  
 displayPerson(alice);  
   
 // Create const object  
 const Person bob("Bob", 25);  
 // bob.birthday(); // Error - can't call non-const function  
 cout << bob.getName() << " is " << bob.getAge() << " years old" << endl;  
   
 // Demonstrate mutable  
 cout << "Access count for Bob: " << bob.getAccessCount() << endl;  
   
 return 0;  
}

#### const and References

The combination of const and references is very powerful for efficient, safe programming:

#include <iostream>  
#include <string>  
#include <vector>  
using namespace std;  
  
struct Point {  
 double x, y;  
   
 Point(double \_x, double \_y) : x(\_x), y(\_y) {}  
   
 void move(double dx, double dy) {  
 x += dx;  
 y += dy;  
 }  
   
 // Print function is const - doesn't change the object  
 void print() const {  
 cout << "(" << x << ", " << y << ")" << endl;  
 }  
};  
  
// Function taking const reference - efficient and safe  
double calculateDistance(const Point& p1, const Point& p2) {  
 double dx = p2.x - p1.x;  
 double dy = p2.y - p1.y;  
 return sqrt(dx\*dx + dy\*dy);  
}  
  
int main() {  
 Point p1(3, 4);  
 Point p2(7, 9);  
   
 p1.print(); // Can call const method on non-const object  
 p1.move(1, 2); // Can call non-const method on non-const object  
 p1.print();  
   
 cout << "Distance: " << calculateDistance(p1, p2) << endl;  
   
 // References to const objects  
 const Point origin(0, 0);  
 const Point& ref = origin; // const ref to const object  
   
 // origin.move(1, 1); // Error - can't modify const object  
 origin.print(); // OK - print() is const  
   
 return 0;  
}

### mutable

The mutable keyword allows a member variable to be modified even in a const object or const member function:

#include <iostream>  
#include <string>  
#include <mutex>  
using namespace std;  
  
// Example 1: Basic mutable usage  
class Counter {  
private:  
 mutable int count = 0; // Can be modified in const functions  
   
public:  
 void increment() {  
 count++;  
 }  
   
 int getCount() const {  
 count++; // Legal because count is mutable  
 return count;  
 }  
};  
  
// Example 2: Mutable with caching  
class MathHelper {  
private:  
 int base;  
 mutable int lastPower = 0;  
 mutable int lastResult = 0;  
 mutable bool hasCache = false;  
   
public:  
 MathHelper(int b) : base(b) {}  
   
 int getPower(int exponent) const {  
 if (hasCache && lastPower == exponent) {  
 cout << "Using cached result" << endl;  
 return lastResult;  
 }  
   
 cout << "Computing new result" << endl;  
 int result = 1;  
 for (int i = 0; i < exponent; i++) {  
 result \*= base;  
 }  
   
 // Update cache - allowed because these are mutable  
 lastPower = exponent;  
 lastResult = result;  
 hasCache = true;  
   
 return result;  
 }  
};  
  
// Example 3: Mutable with thread safety  
class ThreadSafeCounter {  
private:  
 mutable mutex mtx; // Mutex can be locked in const functions  
 mutable int count = 0;  
   
public:  
 void increment() {  
 lock\_guard<mutex> lock(mtx);  
 count++;  
 }  
   
 int getCount() const {  
 lock\_guard<mutex> lock(mtx); // Legal because mtx is mutable  
 return count;  
 }  
};  
  
int main() {  
 // Example 1  
 const Counter c;  
 cout << "Initial count: " << c.getCount() << endl;  
 cout << "Next count: " << c.getCount() << endl;  
   
 // Example 2  
 const MathHelper helper(2);  
 cout << "2^8 = " << helper.getPower(8) << endl;  
 cout << "2^8 again = " << helper.getPower(8) << endl;  
 cout << "2^10 = " << helper.getPower(10) << endl;  
   
 // Example 3  
 const ThreadSafeCounter tsc;  
 cout << "Thread-safe count: " << tsc.getCount() << endl;  
   
 return 0;  
}

### volatile

The volatile keyword tells the compiler that a variable may change in ways that the compiler cannot predict, so it should not optimize away accesses to the variable:

#include <iostream>  
#include <thread>  
#include <chrono>  
using namespace std;  
  
// Simulating a memory-mapped hardware register  
volatile int hardwareStatus = 0;  
  
// Function to simulate hardware changing the status register  
void hardwareSimulation() {  
 this\_thread::sleep\_for(chrono::seconds(1));  
 hardwareStatus = 1;  
 this\_thread::sleep\_for(chrono::seconds(1));  
 hardwareStatus = 2;  
 this\_thread::sleep\_for(chrono::seconds(1));  
 hardwareStatus = 3;  
}  
  
int main() {  
 cout << "Starting hardware simulation..." << endl;  
 thread hardwareThread(hardwareSimulation);  
   
 // Poll the volatile variable - compiler won't optimize this away  
 while (hardwareStatus < 3) {  
 cout << "Current hardware status: " << hardwareStatus << endl;  
 this\_thread::sleep\_for(chrono::milliseconds(500));  
 }  
   
 cout << "Final hardware status: " << hardwareStatus << endl;  
 hardwareThread.join();  
   
 return 0;  
}

#### When to Use volatile

volatile is used in these situations: - Memory-mapped hardware registers - Variables shared between threads without synchronization mechanisms - Signal handlers - setjmp/longjmp constructs

#### What volatile Does Not Do

Common misconceptions about volatile: - It’s **not** a synchronization mechanism between threads - It **doesn’t** create atomic operations - It **doesn’t** establish memory barriers

For thread synchronization, use <atomic> or proper mutexes instead.

### Combining const, mutable, and volatile

These qualifiers can be used together in some cases:

#include <iostream>  
using namespace std;  
  
class SensorReader {  
private:  
 // Register that can change asynchronously, and we won't modify it  
 const volatile int\* statusRegister;  
   
 // Cache that we can modify even in const functions  
 mutable int lastReading = 0;  
 mutable bool hasReading = false;  
   
public:  
 SensorReader(const volatile int\* reg) : statusRegister(reg) {}  
   
 int getReading() const {  
 if (!hasReading) {  
 lastReading = \*statusRegister; // Read volatile through const method  
 hasReading = true;  
 }  
 return lastReading;  
 }  
   
 void resetCache() {  
 hasReading = false;  
 }  
};  
  
int main() {  
 volatile int hardwareReg = 42;  
   
 SensorReader reader(&hardwareReg);  
 cout << "First reading: " << reader.getReading() << endl;  
   
 // Hardware changes the value  
 hardwareReg = 100;  
   
 cout << "Second reading (cached): " << reader.getReading() << endl;  
 reader.resetCache();  
 cout << "Third reading (after reset): " << reader.getReading() << endl;  
   
 return 0;  
}

### Best Practices

1. **Use const aggressively**: It helps catch bugs, enables compiler optimizations, and communicates intent.

* void processData(const vector<int>& data); // Shows data won't be modified

1. **Prefer const references for parameters**: Avoids copying while ensuring data isn’t modified.

* void process(const BigObject& obj); // Better than passing by value

1. **Use mutable sparingly**: Only for implementation details that don’t affect the logical state.

* mutable mutex mtx; // Good use - synchronization doesn't affect logical state

1. **Use volatile only for hardware interaction**: Don’t use it for thread synchronization.

* volatile int\* hardwareRegister; // Good use

1. **Make member functions const whenever possible**: Enables them to work with const objects and communicates that they don’t modify state.

## 7.6 Type Casting (static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast)

C++ provides several mechanisms for type conversion, ranging from implicit conversions to explicit casts.

### Traditional C-style Cast

The C-style cast is inherited from C and has several drawbacks:

#include <iostream>  
using namespace std;  
  
int main() {  
 // C-style cast examples  
 int i = 42;  
 double d = (double)i; // Convert int to double  
   
 // Potentially dangerous - forces the conversion  
 char\* str = "Hello";  
 int\* ptr = (int\*)str; // Very dangerous cast!  
   
 // C-style cast can remove const  
 const int ci = 10;  
 int\* p = (int\*)&ci; // Removes const qualifier  
 \*p = 20; // Modifies a const variable!  
   
 cout << "ci = " << ci << ", \*p = " << \*p << endl;  
   
 return 0;  
}

C-style casts are problematic because: - They can perform multiple types of conversions - They are hard to identify in code - The compiler checks are minimal - They can silently subvert the type system

### Modern C++ Casts

C++ provides four specific cast operators that make the intent clearer and safer:

1. static\_cast
2. dynamic\_cast
3. const\_cast
4. reinterpret\_cast

#### static\_cast

static\_cast performs conversions where the types are related and the conversion makes sense:

#include <iostream>  
using namespace std;  
  
int main() {  
 // Numeric conversions  
 int i = 42;  
 double d = static\_cast<double>(i); // int to double  
 cout << "d = " << d << endl;  
   
 double pi = 3.14159;  
 int rounded = static\_cast<int>(pi); // double to int (truncates)  
 cout << "rounded = " << rounded << endl;  
   
 // Pointer upcast (derived to base)  
 class Base {};  
 class Derived : public Base {};  
   
 Derived derived;  
 Base\* basePtr = static\_cast<Base\*>(&derived); // Upcast - safe  
   
 // void\* conversions  
 void\* voidPtr = &i;  
 int\* intPtr = static\_cast<int\*>(voidPtr); // void\* to int\*  
 cout << "\*intPtr = " << \*intPtr << endl;  
   
 // Cannot be used to cast away const  
 const int ci = 10;  
 // int\* p = static\_cast<int\*>(&ci); // Error: cast drops const qualifier  
   
 return 0;  
}

When to use static\_cast: - Numeric type conversions (int to float, etc.) - Void pointer conversions - Base to derived pointer conversions (when you’re sure it’s safe) - Explicit constructor calls - Explicit conversion operator calls

#### dynamic\_cast

dynamic\_cast is used primarily for safe downcasting (base class pointer to derived class pointer). It performs runtime type checking:

#include <iostream>  
#include <memory>  
using namespace std;  
  
class Base {  
public:  
 virtual ~Base() {} // Virtual destructor makes it polymorphic  
 virtual void speak() {  
 cout << "Base speaking" << endl;  
 }  
};  
  
class Derived1 : public Base {  
public:  
 void speak() override {  
 cout << "Derived1 speaking" << endl;  
 }  
   
 void derived1Method() {  
 cout << "Method specific to Derived1" << endl;  
 }  
};  
  
class Derived2 : public Base {  
public:  
 void speak() override {  
 cout << "Derived2 speaking" << endl;  
 }  
   
 void derived2Method() {  
 cout << "Method specific to Derived2" << endl;  
 }  
};  
  
void processObject(Base\* obj) {  
 // Try to cast to Derived1  
 if (Derived1\* d1 = dynamic\_cast<Derived1\*>(obj)) {  
 cout << "Object is Derived1" << endl;  
 d1->derived1Method();  
 }  
 // Try to cast to Derived2  
 else if (Derived2\* d2 = dynamic\_cast<Derived2\*>(obj)) {  
 cout << "Object is Derived2" << endl;  
 d2->derived2Method();  
 }  
 else {  
 cout << "Object is just a Base" << endl;  
 }  
   
 // Polymorphic call works for any type  
 obj->speak();  
}  
  
int main() {  
 Base base;  
 Derived1 derived1;  
 Derived2 derived2;  
   
 cout << "Processing Base:" << endl;  
 processObject(&base);  
   
 cout << "\nProcessing Derived1:" << endl;  
 processObject(&derived1);  
   
 cout << "\nProcessing Derived2:" << endl;  
 processObject(&derived2);  
   
 // Also works with references  
 Base& ref = derived1;  
 try {  
 Derived1& d1Ref = dynamic\_cast<Derived1&>(ref); // OK  
 cout << "\nSuccessful reference cast to Derived1" << endl;  
   
 Derived2& d2Ref = dynamic\_cast<Derived2&>(ref); // Will throw  
 cout << "This line shouldn't execute" << endl;  
 }  
 catch (const bad\_cast& e) {  
 cout << "Exception caught: " << e.what() << endl;  
 }  
   
 return 0;  
}

Important points about dynamic\_cast: - Only works with polymorphic types (classes with at least one virtual function) - Performs runtime type checking - Returns nullptr for pointers or throws std::bad\_cast for references if the cast fails - Has runtime overhead due to the type checking - Use when the base class doesn’t provide a virtual method to determine the derived type safely

#### const\_cast

const\_cast is used to add or remove const or volatile qualifiers from a variable:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Function that doesn't modify string but doesn't declare it const  
void legacyFunction(char\* str) {  
 cout << "Legacy function received: " << str << endl;  
}  
  
int main() {  
 // Removing const  
 const char\* constStr = "Hello, world";  
 char\* mutableStr = const\_cast<char\*>(constStr);  
   
 // DANGER: modifying string literal leads to undefined behavior  
 // mutableStr[0] = 'h'; // Don't do this!  
   
 // Safe use: calling non-const API with const data  
 legacyFunction(mutableStr); // OK, we know it won't modify the string  
   
 // Adding const  
 char buffer[] = "Modifiable buffer";  
 const char\* constBuffer = const\_cast<const char\*>(buffer);  
 // constBuffer[0] = 'm'; // Error: constBuffer is const  
   
 // Modifying the original is still allowed  
 buffer[0] = 'm';  
 cout << "Modified buffer: " << buffer << endl;  
 cout << "Through const view: " << constBuffer << endl;  
   
 return 0;  
}

When to use const\_cast: - Calling non-const API with const data when you’re sure it won’t modify - Adding const for additional compile-time safety - Working with legacy code that doesn’t use const correctly

**Warning**: Modifying an originally const object through a non-const pointer created by const\_cast leads to undefined behavior.

#### reinterpret\_cast

reinterpret\_cast performs low-level reinterpretation of bit patterns. It’s the most dangerous cast and should be used with extreme caution:

#include <iostream>  
using namespace std;  
  
struct Coordinates {  
 double x, y, z;  
};  
  
int main() {  
 // Converting between unrelated pointer types  
 int number = 0x12345678;  
 char\* charPtr = reinterpret\_cast<char\*>(&number);  
   
 // Examining individual bytes (depends on endianness)  
 cout << "Bytes of int 0x12345678:" << endl;  
 for (int i = 0; i < sizeof(int); i++) {  
 printf("%02X ", static\_cast<unsigned char>(charPtr[i]));  
 }  
 cout << endl;  
   
 // Pointer to integer conversion  
 Coordinates coords = {1.0, 2.0, 3.0};  
 Coordinates\* coordsPtr = &coords;  
   
 uintptr\_t addressAsInt = reinterpret\_cast<uintptr\_t>(coordsPtr);  
 cout << "Address as integer: " << addressAsInt << endl;  
   
 // And back again  
 Coordinates\* recoveredPtr = reinterpret\_cast<Coordinates\*>(addressAsInt);  
 cout << "Recovered coordinates: (" << recoveredPtr->x << ", "   
 << recoveredPtr->y << ", " << recoveredPtr->z << ")" << endl;  
   
 return 0;  
}

When to use reinterpret\_cast: - Working with device drivers where memory maps to hardware - Implementing serialization/deserialization - Bit manipulation routines - Converting between function pointers and data pointers (platform-specific)

**Warning**: reinterpret\_cast is extremely dangerous and should be used only when absolutely necessary. It can easily lead to undefined behavior.

### Comparing the Different Casts

Here’s a comparison table:

| Cast Type | Purpose | Safety Level | Runtime Check |
| --- | --- | --- | --- |
| static\_cast | Related type conversions | Medium | No |
| dynamic\_cast | Safe downcasting | High | Yes |
| const\_cast | Add/remove const/volatile | Low | No |
| reinterpret\_cast | Reinterpret bits | Very Low | No |

#include <iostream>  
using namespace std;  
  
class Base {  
public:  
 virtual void method() { cout << "Base::method()" << endl; }  
 virtual ~Base() {}  
};  
  
class Derived : public Base {  
public:  
 void method() override { cout << "Derived::method()" << endl; }  
 void derivedOnly() { cout << "Derived::derivedOnly()" << endl; }  
};  
  
int main() {  
 // Create objects  
 Base base;  
 Derived derived;  
   
 // Setup pointers  
 Base\* basePtr = &base;  
 Base\* derivedAsBsePtr = &derived;  
   
 // 1. static\_cast  
 // Upcast (always safe)  
 Base\* upcastPtr = static\_cast<Base\*>(&derived);  
 upcastPtr->method(); // Calls Derived::method() due to virtual function  
   
 // Downcast (unsafe - no runtime check)  
 Derived\* downcastPtr1 = static\_cast<Derived\*>(basePtr); // Points to Base but cast to Derived  
 // downcastPtr1->derivedOnly(); // Undefined behavior - basePtr isn't actually a Derived  
   
 Derived\* downcastPtr2 = static\_cast<Derived\*>(derivedAsBsePtr); // Safe in this case  
 downcastPtr2->derivedOnly(); // Works because it really is a Derived  
   
 // 2. dynamic\_cast  
 Derived\* safePtr1 = dynamic\_cast<Derived\*>(basePtr);  
 if (safePtr1) {  
 cout << "basePtr is a Derived (should not happen)" << endl;  
 safePtr1->derivedOnly();  
 } else {  
 cout << "basePtr is NOT a Derived (correct)" << endl;  
 }  
   
 Derived\* safePtr2 = dynamic\_cast<Derived\*>(derivedAsBsePtr);  
 if (safePtr2) {  
 cout << "derivedAsBsePtr is a Derived (correct)" << endl;  
 safePtr2->derivedOnly();  
 } else {  
 cout << "derivedAsBsePtr is NOT a Derived (should not happen)" << endl;  
 }  
   
 // 3. const\_cast  
 const Base\* constBasePtr = &base;  
 // constBasePtr->method(); // OK - method() might be const  
 Base\* mutableBasePtr = const\_cast<Base\*>(constBasePtr);  
 mutableBasePtr->method(); // Now allowed to call non-const methods  
   
 // 4. reinterpret\_cast  
 long addr = reinterpret\_cast<long>(&derived);  
 cout << "Address of derived: " << addr << endl;  
 Derived\* recoveredPtr = reinterpret\_cast<Derived\*>(addr);  
 recoveredPtr->method(); // Works because we got the original address back  
   
 return 0;  
}

### Casting Best Practices

1. **Avoid C-style casts**: Use modern C++ casts instead

* // Avoid  
  double d = (double)intValue;  
    
  // Prefer  
  double d = static\_cast<double>(intValue);

1. **Use dynamic\_cast for polymorphic downcasts**

* if (Derived\* derivedPtr = dynamic\_cast<Derived\*>(basePtr)) {  
   derivedPtr->derivedMethod();  
  }

1. **Minimize use of const\_cast**: Often indicates design issues

* // Try to fix the API instead of using const\_cast  
  void properAPI(const char\* str) { ... }

1. **Treat reinterpret\_cast as dangerous**: Use only when absolutely necessary

* // Document your reasoning whenever you use reinterpret\_cast  
  // Ensure you understand the platform-specific implications

1. **Consider alternatives to casting**:
   * Virtual functions instead of downcasting
   * Templates instead of casting
   * Function overloading
   * Type-safe containers
2. **Document why you’re using a cast**, especially const\_cast and reinterpret\_cast

### Practical Examples

#### Example 1: Visitor Pattern with dynamic\_cast

#include <iostream>  
#include <vector>  
#include <memory>  
using namespace std;  
  
// Base class for all shapes  
class Shape {  
public:  
 virtual ~Shape() {}  
};  
  
// Derived classes  
class Circle : public Shape {  
private:  
 double radius;  
public:  
 Circle(double r) : radius(r) {}  
 double getRadius() const { return radius; }  
};  
  
class Rectangle : public Shape {  
private:  
 double width, height;  
public:  
 Rectangle(double w, double h) : width(w), height(h) {}  
 double getWidth() const { return width; }  
 double getHeight() const { return height; }  
};  
  
class Triangle : public Shape {  
private:  
 double base, height;  
public:  
 Triangle(double b, double h) : base(b), height(h) {}  
 double getBase() const { return base; }  
 double getHeight() const { return height; }  
};  
  
// Visitor using dynamic\_cast  
class AreaCalculator {  
public:  
 double calculateArea(const Shape& shape) {  
 if (auto circle = dynamic\_cast<const Circle\*>(&shape)) {  
 return 3.14159 \* circle->getRadius() \* circle->getRadius();  
 }  
 else if (auto rect = dynamic\_cast<const Rectangle\*>(&shape)) {  
 return rect->getWidth() \* rect->getHeight();  
 }  
 else if (auto tri = dynamic\_cast<const Triangle\*>(&shape)) {  
 return 0.5 \* tri->getBase() \* tri->getHeight();  
 }  
 return 0.0; // Unknown shape  
 }  
};  
  
int main() {  
 vector<unique\_ptr<Shape>> shapes;  
 shapes.push\_back(make\_unique<Circle>(5.0));  
 shapes.push\_back(make\_unique<Rectangle>(4.0, 6.0));  
 shapes.push\_back(make\_unique<Triangle>(3.0, 8.0));  
   
 AreaCalculator calculator;  
   
 for (const auto& shape : shapes) {  
 cout << "Area: " << calculator.calculateArea(\*shape) << endl;  
 }  
   
 return 0;  
}

#### Example 2: Working with Legacy API using const\_cast

#include <iostream>  
#include <string>  
using namespace std;  
  
// Legacy API that doesn't respect const-correctness  
void legacyPrint(char\* buffer) {  
 cout << "Legacy function printing: " << buffer << endl;  
 // Doesn't modify buffer but doesn't declare it const  
}  
  
class ModernClass {  
private:  
 const char\* message;  
   
public:  
 ModernClass(const char\* msg) : message(msg) {}  
   
 void printMessage() const {  
 // We know legacyPrint doesn't modify the string, so const\_cast is safe here  
 legacyPrint(const\_cast<char\*>(message));  
 }  
};  
  
int main() {  
 ModernClass obj("Hello from modern C++");  
 obj.printMessage();  
   
 return 0;  
}

#### Example 3: Memory Manipulation with reinterpret\_cast

#include <iostream>  
#include <cstring>  
using namespace std;  
  
struct RGBColor {  
 uint8\_t r, g, b;  
};  
  
struct PackedColor {  
 uint32\_t value; // Contains RGB in lowest 24 bits  
};  
  
int main() {  
 // Convert between memory representations  
 RGBColor rgb = {255, 128, 64};  
   
 // Pack RGB into 32-bit integer  
 PackedColor packed;  
 packed.value = (static\_cast<uint32\_t>(rgb.r) << 16) |  
 (static\_cast<uint32\_t>(rgb.g) << 8) |  
 rgb.b;  
   
 cout << "Packed color: 0x" << hex << packed.value << dec << endl;  
   
 // Reinterpret packed bits as array of bytes  
 uint8\_t\* bytes = reinterpret\_cast<uint8\_t\*>(&packed.value);  
   
 cout << "Individual bytes: ";  
 for (int i = 0; i < sizeof(uint32\_t); i++) {  
 cout << "0x" << hex << static\_cast<int>(bytes[i]) << " ";  
 }  
 cout << dec << endl;  
   
 return 0;  
}

Remember, casts should be used judiciously. Each time you use a cast, especially const\_cast or reinterpret\_cast, you should consider whether there’s a better design that would avoid the need for casting.

# Chapter 7: Advanced C++ Features (Part 3)

## 7.7 Move Semantics & Rvalue References

Move semantics, introduced in C++11, revolutionized the way resources are managed and transferred between objects. This feature significantly improves performance when working with objects that manage resources (like memory, file handles, or network connections).

### Understanding L-values and R-values

To understand move semantics, we first need to understand the concepts of l-values and r-values:

* **L-value**: An expression that refers to a memory location and can appear on the left side of an assignment.
* **R-value**: An expression that isn’t an l-value; typically a temporary value or literal that cannot appear on the left side of an assignment.

#include <iostream>  
using namespace std;  
  
int main() {  
 int x = 10; // x is an l-value, 10 is an r-value  
   
 int y = x; // y is an l-value, x is an l-value but used as an r-value  
   
 // int& ref1 = 42; // Error: cannot bind non-const lvalue reference to an rvalue  
 const int& ref2 = 42; // OK: can bind const lvalue reference to an rvalue  
   
 // These are all r-values:  
 // 10  
 // x + y  
 // x++ (the value produced, not x itself)  
   
 // These are all l-values:  
 // x  
 // ++x (both the value produced and x itself)  
 // ref2  
   
 return 0;  
}

### Rvalue References

C++11 introduced a new reference type called an **rvalue reference**, denoted by &&. This allows us to bind specifically to r-values:

#include <iostream>  
using namespace std;  
  
int main() {  
 int x = 10;  
   
 // lvalue reference - binds to lvalues  
 int& lref = x;  
 // int& invalid\_lref = 10; // Error: can't bind lvalue reference to rvalue  
   
 // rvalue reference - binds to rvalues  
 int&& rref = 20; // OK: 20 is an rvalue  
 // int&& invalid\_rref = x; // Error: can't bind rvalue reference to lvalue  
   
 // But we can force it with std::move  
 int&& forced\_rref = move(x); // OK: move(x) creates an rvalue  
   
 cout << "x: " << x << endl;  
 cout << "lref: " << lref << endl;  
 cout << "rref: " << rref << endl;  
 cout << "forced\_rref: " << forced\_rref << endl;  
   
 // Changing through references affects the original  
 lref = 100;  
 cout << "After lref = 100, x: " << x << endl;  
   
 forced\_rref = 200;  
 cout << "After forced\_rref = 200, x: " << x << endl;  
   
 return 0;  
}

### Understanding std::move

std::move doesn’t actually move anything - it’s just a cast that produces an rvalue reference to its argument:

#include <iostream>  
#include <utility> // For std::move  
#include <string>  
using namespace std;  
  
int main() {  
 // Example with a fundamental type  
 int a = 10;  
 int b = move(a); // Same as: int b = a;  
 cout << "a: " << a << ", b: " << b << endl; // a is still 10  
   
 // Example with a string  
 string str1 = "Hello, World!";  
 string str2 = move(str1); // str1's contents are moved to str2  
   
 cout << "str1: '" << str1 << "'" << endl; // str1 might be empty or in a valid but unspecified state  
 cout << "str2: '" << str2 << "'" << endl; // str2 now has "Hello, World!"  
   
 return 0;  
}

### Move Constructor and Move Assignment

Move semantics are implemented through two special member functions:

1. **Move Constructor**: ClassName(ClassName&& other)
2. **Move Assignment Operator**: ClassName& operator=(ClassName&& other)

Here’s an example of a class with both move and copy operations:

#include <iostream>  
#include <utility>  
#include <vector>  
using namespace std;  
  
class Resource {  
private:  
 int\* data;  
 size\_t size;  
   
 void logOperation(const string& op) const {  
 cout << op << " [this=" << this << ", data=" << data << ", size=" << size << "]" << endl;  
 }  
   
public:  
 // Constructor  
 Resource(size\_t sz) : size(sz) {  
 data = new int[size];  
 for (size\_t i = 0; i < size; ++i) {  
 data[i] = i;  
 }  
 logOperation("Constructor");  
 }  
   
 // Destructor  
 ~Resource() {  
 logOperation("Destructor");  
 delete[] data;  
 }  
   
 // Copy constructor  
 Resource(const Resource& other) : size(other.size) {  
 data = new int[size];  
 copy(other.data, other.data + size, data);  
 logOperation("Copy constructor");  
 }  
   
 // Copy assignment operator  
 Resource& operator=(const Resource& other) {  
 logOperation("Copy assignment operator");  
 if (this != &other) {  
 delete[] data;  
 size = other.size;  
 data = new int[size];  
 copy(other.data, other.data + size, data);  
 }  
 return \*this;  
 }  
   
 // Move constructor  
 Resource(Resource&& other) noexcept : data(other.data), size(other.size) {  
 logOperation("Move constructor");  
 // Important: leave the source in a valid state  
 other.data = nullptr;  
 other.size = 0;  
 }  
   
 // Move assignment operator  
 Resource& operator=(Resource&& other) noexcept {  
 logOperation("Move assignment operator");  
 if (this != &other) {  
 delete[] data;  
 data = other.data;  
 size = other.size;  
   
 // Important: leave the source in a valid state  
 other.data = nullptr;  
 other.size = 0;  
 }  
 return \*this;  
 }  
   
 // Utility method  
 void print() const {  
 cout << "Resource [data=" << data << ", size=" << size << "]: ";  
 for (size\_t i = 0; i < size && i < 10; ++i) {  
 cout << data[i] << " ";  
 }  
 if (size > 10) cout << "...";  
 cout << endl;  
 }  
   
 // Add some data for testing  
 void update(int value) {  
 for (size\_t i = 0; i < size; ++i) {  
 data[i] = value + i;  
 }  
 }  
};  
  
int main() {  
 cout << "Creating r1" << endl;  
 Resource r1(5);  
 r1.update(10);  
 r1.print();  
   
 cout << "\nCopy construction:" << endl;  
 Resource r2 = r1; // Copy constructor  
 r2.print();  
   
 cout << "\nMove construction:" << endl;  
 Resource r3 = move(r1); // Move constructor  
 r3.print();  
   
 // r1 has been moved from, should be in a valid but unspecified state  
 cout << "\nr1 after move:" << endl;  
 r1.print(); // Should be empty or in a valid state  
   
 cout << "\nCopy assignment:" << endl;  
 Resource r4(3);  
 r4 = r2; // Copy assignment  
 r4.print();  
   
 cout << "\nMove assignment:" << endl;  
 Resource r5(2);  
 r5 = move(r2); // Move assignment  
 r5.print();  
   
 cout << "\nr2 after move:" << endl;  
 r2.print(); // Should be empty or in a valid state  
   
 return 0;  
}

### Perfect Forwarding

Perfect forwarding allows a function template to preserve the value category and cv-qualifiers of its arguments when passing them to another function:

#include <iostream>  
#include <utility>  
#include <string>  
using namespace std;  
  
// A function that takes both lvalue and rvalue arguments  
void process(string& s) {  
 cout << "Process lvalue: " << s << endl;  
}  
  
void process(string&& s) {  
 cout << "Process rvalue: " << s << endl;  
}  
  
// A forwarding function template  
template<typename T>  
void perfectForward(T&& arg) {  
 // Forward keeps the value category (lvalue or rvalue) of the argument  
 cout << "Forwarding to: ";  
 process(forward<T>(arg));  
}  
  
int main() {  
 string hello = "Hello";  
   
 // Direct calls  
 cout << "Direct calls:" << endl;  
 process(hello); // Calls lvalue version  
 process(move(hello)); // Calls rvalue version  
 process("temporary"); // Calls rvalue version  
   
 // With perfect forwarding  
 cout << "\nWith forwarding:" << endl;  
 perfectForward(hello); // Calls lvalue version  
 perfectForward(move(hello)); // Calls rvalue version  
 perfectForward("another temp"); // Calls rvalue version  
   
 return 0;  
}

### Universal References

The term “universal reference” (coined by Scott Meyers) refers to a special case of rvalue references:

#include <iostream>  
#include <utility>  
#include <string>  
using namespace std;  
  
// T&& here is a universal reference, not an rvalue reference  
template<typename T>  
void universalRef(T&& param) {  
 // Type deduction happens with T&&  
 cout << "T&& param: " << param << endl;  
   
 // We can check if it's an lvalue or rvalue reference  
 if (is\_lvalue\_reference<T>::value) {  
 cout << "Parameter is an lvalue reference" << endl;  
 } else {  
 cout << "Parameter is an rvalue reference" << endl;  
 }  
}  
  
int main() {  
 string hello = "Hello";  
   
 universalRef(hello); // T deduced as string&, param is string&  
 universalRef("World"); // T deduced as const char[6], param is const char(&&)[6]  
 universalRef(move(hello)); // T deduced as string, param is string&&  
   
 return 0;  
}

### Move Semantics with Standard Library

Standard library containers take advantage of move semantics when available:

#include <iostream>  
#include <vector>  
#include <string>  
using namespace std;  
  
class ExpensiveObject {  
private:  
 string\* data;  
   
public:  
 ExpensiveObject() : data(new string("Default")) {  
 cout << "Default constructor [" << \*data << "]" << endl;  
 }  
   
 ExpensiveObject(const string& str) : data(new string(str)) {  
 cout << "String constructor [" << \*data << "]" << endl;  
 }  
   
 ~ExpensiveObject() {  
 if (data) cout << "Destructor [" << \*data << "]" << endl;  
 delete data;  
 }  
   
 // Copy operations  
 ExpensiveObject(const ExpensiveObject& other) : data(new string(\*other.data)) {  
 cout << "Copy constructor [" << \*data << "]" << endl;  
 }  
   
 ExpensiveObject& operator=(const ExpensiveObject& other) {  
 cout << "Copy assignment [" << \*data << " = " << \*other.data << "]" << endl;  
 if (this != &other) {  
 \*data = \*other.data;  
 }  
 return \*this;  
 }  
   
 // Move operations  
 ExpensiveObject(ExpensiveObject&& other) noexcept : data(other.data) {  
 cout << "Move constructor [" << \*data << "]" << endl;  
 other.data = nullptr; // Prevent double deletion  
 }  
   
 ExpensiveObject& operator=(ExpensiveObject&& other) noexcept {  
 cout << "Move assignment [" << \*data << " = " << \*other.data << "]" << endl;  
 if (this != &other) {  
 delete data;  
 data = other.data;  
 other.data = nullptr; // Prevent double deletion  
 }  
 return \*this;  
 }  
   
 // For displaying  
 void display() const {  
 cout << "ExpensiveObject: " << (data ? \*data : "null") << endl;  
 }  
};  
  
int main() {  
 // Using vector's emplace\_back (constructs in-place)  
 cout << "--- Vector with emplace\_back ---" << endl;  
 vector<ExpensiveObject> v1;  
 v1.emplace\_back("First"); // Constructs directly in the vector  
 v1.emplace\_back("Second");  
   
 // Using push\_back with move  
 cout << "\n--- Vector with push\_back and move ---" << endl;  
 vector<ExpensiveObject> v2;  
 ExpensiveObject obj("Third");  
 v2.push\_back(move(obj)); // Moves obj into the vector  
   
 // obj was moved from, should be in a valid but unspecified state  
 cout << "\nAfter move, obj is: ";  
 obj.display();  
   
 // Using temporary objects  
 cout << "\n--- Vector with temporary objects ---" << endl;  
 vector<ExpensiveObject> v3;  
 v3.push\_back(ExpensiveObject("Fourth")); // Temporary is moved, not copied  
   
 return 0;  
}

### Best Practices for Move Semantics

1. **Always mark move operations with noexcept**
   * This allows standard library containers to optimize certain operations
2. **Leave moved-from objects in a valid (albeit unspecified) state**
   * Typically this means nulling out pointers
   * Setting primitive types to safe defaults
3. **Consider move operations for classes that manage resources**
   * Memory, file handles, network connections, etc.
4. **Use std::move with caution**
   * Don’t use it when you still need the object’s value
   * Be especially careful with forwarded parameters
5. **Don’t inhibit compiler optimizations**
   * Return local objects by value - copy elision will optimize away the copy

## 7.8 Rule of 3 / 5 / 0

The “Rule of Three/Five/Zero” is a guideline for class design regarding special member functions in C++.

### Rule of Three

The Rule of Three states that if a class requires any of these three special member functions, it probably needs all three:

1. Destructor
2. Copy constructor
3. Copy assignment operator

This is because these functions are usually needed together when a class manages a resource (like memory or file handles).

#include <iostream>  
#include <cstring> // For strlen, strcpy  
using namespace std;  
  
// Class that follows the Rule of Three  
class Buffer {  
private:  
 char\* data;  
 size\_t size;  
   
public:  
 // Constructor  
 Buffer(const char\* str) {  
 size = strlen(str) + 1;  
 data = new char[size];  
 strcpy(data, str);  
 cout << "Constructor: " << data << endl;  
 }  
   
 // Destructor (1/3)  
 ~Buffer() {  
 cout << "Destructor: " << (data ? data : "nullptr") << endl;  
 delete[] data;  
 }  
   
 // Copy constructor (2/3)  
 Buffer(const Buffer& other) : size(other.size) {  
 data = new char[size];  
 strcpy(data, other.data);  
 cout << "Copy constructor: " << data << endl;  
 }  
   
 // Copy assignment operator (3/3)  
 Buffer& operator=(const Buffer& other) {  
 cout << "Copy assignment operator" << endl;  
 if (this != &other) {  
 delete[] data; // Free old resource  
   
 size = other.size;  
 data = new char[size];  
 strcpy(data, other.data);  
 }  
 return \*this;  
 }  
   
 // Display the buffer  
 void display() const {  
 cout << "Buffer: " << data << " (size: " << size << ")" << endl;  
 }  
};  
  
void showRuleOfThree() {  
 Buffer b1("Hello");  
 b1.display();  
   
 // Copy construction  
 Buffer b2 = b1; // or Buffer b2(b1);  
 b2.display();  
   
 // Copy assignment  
 Buffer b3("Temporary");  
 b3 = b1;  
 b3.display();  
}  
  
int main() {  
 cout << "Demonstrating Rule of Three:" << endl;  
 showRuleOfThree();  
   
 return 0;  
}

### Rule of Five

With the introduction of move semantics in C++11, the Rule of Three became the Rule of Five. If a class needs custom versions of any of these five special member functions, it probably needs all five:

1. Destructor
2. Copy constructor
3. Copy assignment operator
4. Move constructor
5. Move assignment operator

#include <iostream>  
#include <cstring> // For strlen, strcpy  
using namespace std;  
  
// Class that follows the Rule of Five  
class DynamicArray {  
private:  
 int\* data;  
 size\_t size;  
   
public:  
 // Constructor  
 DynamicArray(size\_t sz) : size(sz) {  
 data = new int[size];  
 fill(data, data + size, 0);  
 cout << "Constructor: size " << size << endl;  
 }  
   
 // Destructor (1/5)  
 ~DynamicArray() {  
 cout << "Destructor: size " << size << endl;  
 delete[] data;  
 }  
   
 // Copy constructor (2/5)  
 DynamicArray(const DynamicArray& other) : size(other.size) {  
 data = new int[size];  
 copy(other.data, other.data + size, data);  
 cout << "Copy constructor: size " << size << endl;  
 }  
   
 // Copy assignment operator (3/5)  
 DynamicArray& operator=(const DynamicArray& other) {  
 cout << "Copy assignment operator" << endl;  
 if (this != &other) {  
 delete[] data; // Free old resource  
   
 size = other.size;  
 data = new int[size];  
 copy(other.data, other.data + size, data);  
 }  
 return \*this;  
 }  
   
 // Move constructor (4/5)  
 DynamicArray(DynamicArray&& other) noexcept  
 : data(other.data), size(other.size) {  
 cout << "Move constructor: size " << size << endl;  
 // Leave source in valid, empty state  
 other.data = nullptr;  
 other.size = 0;  
 }  
   
 // Move assignment operator (5/5)  
 DynamicArray& operator=(DynamicArray&& other) noexcept {  
 cout << "Move assignment operator" << endl;  
 if (this != &other) {  
 delete[] data; // Free old resource  
   
 // Take ownership of other's resources  
 data = other.data;  
 size = other.size;  
   
 // Leave source in valid, empty state  
 other.data = nullptr;  
 other.size = 0;  
 }  
 return \*this;  
 }  
   
 // Utility methods  
 void setValue(size\_t index, int value) {  
 if (index < size && data != nullptr) {  
 data[index] = value;  
 }  
 }  
   
 int getValue(size\_t index) const {  
 if (index < size && data != nullptr) {  
 return data[index];  
 }  
 return -1;  
 }  
   
 void display() const {  
 cout << "DynamicArray [size=" << size << ", data=" << data << "]: ";  
 if (data != nullptr) {  
 for (size\_t i = 0; i < size && i < 10; ++i) {  
 cout << data[i] << " ";  
 }  
 if (size > 10) cout << "...";  
 } else {  
 cout << "nullptr";  
 }  
 cout << endl;  
 }  
};  
  
void showRuleOfFive() {  
 DynamicArray a1(5);  
 for (int i = 0; i < 5; ++i) {  
 a1.setValue(i, i \* 10);  
 }  
 a1.display();  
   
 cout << "\n--- Copy operations ---" << endl;  
 DynamicArray a2 = a1; // Copy construction  
 a2.display();  
   
 DynamicArray a3(3);  
 a3 = a1; // Copy assignment  
 a3.display();  
   
 cout << "\n--- Move operations ---" << endl;  
 DynamicArray a4 = move(a1); // Move construction  
 a4.display();  
   
 cout << "a1 after move: ";  
 a1.display(); // Should be in a valid but empty state  
   
 DynamicArray a5(2);  
 a5 = move(a2); // Move assignment  
 a5.display();  
   
 cout << "a2 after move: ";  
 a2.display(); // Should be in a valid but empty state  
}  
  
int main() {  
 cout << "Demonstrating Rule of Five:" << endl;  
 showRuleOfFive();  
   
 return 0;  
}

### Rule of Zero

The Rule of Zero states that if a class can avoid defining any special member functions, it should. Instead, it should rely on the compiler-generated defaults or compose from classes that already handle their resources properly.

#include <iostream>  
#include <vector>  
#include <string>  
#include <memory>  
using namespace std;  
  
// Class that follows the Rule of Zero  
class Document {  
private:  
 string title; // string already manages its memory  
 vector<string> paragraphs; // vector already manages its memory  
 shared\_ptr<string> metadata; // shared\_ptr already manages its resource  
   
public:  
 // Regular constructor - no special handling needed for resources  
 Document(const string& t) : title(t), metadata(make\_shared<string>("Created: 2023")) {  
 cout << "Document created: " << title << endl;  
 }  
   
 // No need for destructor, copy/move operations - they're generated correctly  
   
 void addParagraph(const string& text) {  
 paragraphs.push\_back(text);  
 }  
   
 void display() const {  
 cout << "Document: " << title << endl;  
 cout << "Metadata: " << \*metadata << endl;  
 cout << "Contents:" << endl;  
 for (size\_t i = 0; i < paragraphs.size(); ++i) {  
 cout << " " << (i + 1) << ". " << paragraphs[i] << endl;  
 }  
 }  
};  
  
void showRuleOfZero() {  
 Document doc1("Rule of Zero Example");  
 doc1.addParagraph("Paragraph 1: Introduction to the Rule of Zero");  
 doc1.addParagraph("Paragraph 2: Let standard components handle resources");  
   
 // Copy construction works correctly without custom implementation  
 Document doc2 = doc1;  
 doc2.addParagraph("Paragraph 3: This was added only to doc2");  
   
 cout << "\n--- Original document ---" << endl;  
 doc1.display();  
   
 cout << "\n--- Copied document with extra paragraph ---" << endl;  
 doc2.display();  
   
 // Move operations also work correctly  
 Document doc3 = move(doc1);  
   
 cout << "\n--- Moved document ---" << endl;  
 doc3.display();  
   
 // doc1 is in a valid but unspecified state after the move  
 // Depending on the implementation, doc1.title might be empty now  
}  
  
int main() {  
 cout << "Demonstrating Rule of Zero:" << endl;  
 showRuleOfZero();  
   
 return 0;  
}

### When to Apply Each Rule

1. **Rule of Zero** (preferred):
   * Apply when your class doesn’t manage any resources directly
   * Use standard containers and smart pointers
   * Let the compiler generate the special member functions
2. **Rule of Five**:
   * Apply when your class directly manages a resource
   * Implement all five special member functions consistently
   * Consider if you could redesign to use the Rule of Zero instead
3. **Rule of Three** (C++98/03):
   * Apply in legacy code before C++11
   * Consider upgrading to Rule of Five when moving to C++11 or later

### Special Member Functions and = default, = delete

C++11 introduced explicit ways to control special member functions:

#include <iostream>  
using namespace std;  
  
class ControlledCopy {  
private:  
 int value;  
   
public:  
 ControlledCopy(int v) : value(v) {  
 cout << "Constructor with value " << value << endl;  
 }  
   
 // Default constructor explicitly defaulted  
 ControlledCopy() = default;  
   
 // Copy constructor explicitly defaulted  
 ControlledCopy(const ControlledCopy&) = default;  
   
 // Move constructor explicitly deleted  
 ControlledCopy(ControlledCopy&&) = delete;  
   
 // Copy assignment explicitly deleted  
 ControlledCopy& operator=(const ControlledCopy&) = delete;  
   
 // Move assignment explicitly defaulted (but unused since move constructor is deleted)  
 ControlledCopy& operator=(ControlledCopy&&) = default;  
   
 // Display the value  
 void display() const {  
 cout << "Value: " << value << endl;  
 }  
};  
  
int main() {  
 ControlledCopy a(42);  
 ControlledCopy b; // Default constructor  
   
 // Copy constructor (allowed)  
 ControlledCopy c = a;  
   
 // Copy assignment (not allowed)  
 // b = a; // Error: copy assignment operator is deleted  
   
 // Move constructor (not allowed)  
 // ControlledCopy d = move(a); // Error: move constructor is deleted  
   
 return 0;  
}

## 7.9 Copy Constructor vs Move Constructor

Copy and move constructors both create new objects from existing ones, but they do it in fundamentally different ways:

### Copy Constructor

The copy constructor creates a new object as a copy of an existing object, duplicating its resources:

* Takes a constant reference to an existing object
* Creates a duplicate of all resources
* The source object remains unchanged
* Uses the form: ClassName(const ClassName& other)

### Move Constructor

The move constructor creates a new object by transferring ownership of resources from an existing object:

* Takes an rvalue reference to an existing object
* Steals (moves) resources from the source
* Leaves the source object in a valid but unspecified state
* Uses the form: ClassName(ClassName&& other) noexcept

### Key Differences

#### 1. Parameter Types

class MyClass {  
public:  
 // Copy constructor - takes const reference  
 MyClass(const MyClass& other);  
   
 // Move constructor - takes rvalue reference  
 MyClass(MyClass&& other) noexcept;  
};

#### 2. Resource Handling

#include <iostream>  
#include <string>  
using namespace std;  
  
class StringWrapper {  
private:  
 string\* text;  
   
public:  
 // Constructor  
 StringWrapper(const string& str) : text(new string(str)) {  
 cout << "Constructor: " << \*text << endl;  
 }  
   
 // Destructor  
 ~StringWrapper() {  
 cout << "Destructor: " << (text ? \*text : "null") << endl;  
 delete text;  
 }  
   
 // Copy constructor - duplicates the resource  
 StringWrapper(const StringWrapper& other) : text(new string(\*other.text)) {  
 cout << "Copy constructor: " << \*text << endl;  
 }  
   
 // Move constructor - takes ownership of the resource  
 StringWrapper(StringWrapper&& other) noexcept : text(other.text) {  
 cout << "Move constructor: " << \*text << endl;  
 other.text = nullptr; // Prevent destruction of the moved resource  
 }  
   
 // Get the string value  
 string getValue() const {   
 return text ? \*text : "null";   
 }  
};  
  
int main() {  
 // Original object  
 StringWrapper original("Hello world");  
 cout << "Original: " << original.getValue() << endl;  
   
 // Copy construction  
 StringWrapper copy = original;  
 cout << "Copy: " << copy.getValue() << endl;  
 cout << "Original after copy: " << original.getValue() << endl;  
   
 // Move construction  
 StringWrapper moved = move(original);  
 cout << "Moved: " << moved.getValue() << endl;  
 cout << "Original after move: " << original.getValue() << endl;  
   
 return 0;  
}

#### 3. Performance Implications

#include <iostream>  
#include <string>  
#include <vector>  
#include <chrono>  
using namespace std;  
using namespace chrono;  
  
class LargeObject {  
private:  
 vector<int> data;  
   
public:  
 // Constructor to create a large object  
 LargeObject(size\_t size) : data(size) {  
 for (size\_t i = 0; i < size; ++i) {  
 data[i] = static\_cast<int>(i);  
 }  
 }  
   
 // Copy constructor - expensive, copies all data  
 LargeObject(const LargeObject& other) : data(other.data) {  
 // Expensive deep copy  
 }  
   
 // Move constructor - cheap, just transfers ownership  
 LargeObject(LargeObject&& other) noexcept : data(move(other.data)) {  
 // Fast ownership transfer  
 }  
   
 // Get data size  
 size\_t size() const {  
 return data.size();  
 }  
};  
  
// Test function for performance comparison  
void testPerformance() {  
 const int NUM\_OBJECTS = 1000;  
 const int OBJECT\_SIZE = 100000;  
   
 // Test copy performance  
 auto copyStart = high\_resolution\_clock::now();  
 vector<LargeObject> copyVec;  
 copyVec.reserve(NUM\_OBJECTS); // Reserve to avoid reallocation  
   
 for (int i = 0; i < NUM\_OBJECTS; ++i) {  
 LargeObject obj(OBJECT\_SIZE);  
 copyVec.push\_back(obj); // Copy construction  
 }  
   
 auto copyEnd = high\_resolution\_clock::now();  
 auto copyDuration = duration\_cast<milliseconds>(copyEnd - copyStart).count();  
   
 // Test move performance  
 auto moveStart = high\_resolution\_clock::now();  
 vector<LargeObject> moveVec;  
 moveVec.reserve(NUM\_OBJECTS); // Reserve to avoid reallocation  
   
 for (int i = 0; i < NUM\_OBJECTS; ++i) {  
 LargeObject obj(OBJECT\_SIZE);  
 moveVec.push\_back(move(obj)); // Move construction  
 }  
   
 auto moveEnd = high\_resolution\_clock::now();  
 auto moveDuration = duration\_cast<milliseconds>(moveEnd - moveStart).count();  
   
 cout << "Copy performance: " << copyDuration << " ms" << endl;  
 cout << "Move performance: " << moveDuration << " ms" << endl;  
 cout << "Performance ratio: " << (copyDuration / (moveDuration > 0 ? moveDuration : 1)) << "x" << endl;  
}  
  
int main() {  
 cout << "Performance comparison of copy vs move:" << endl;  
 testPerformance();  
   
 return 0;  
}

#### 4. When Each is Called

#include <iostream>  
#include <utility> // For std::move  
#include <vector>  
using namespace std;  
  
class TraceableObject {  
private:  
 int id;  
   
public:  
 // Constructor  
 TraceableObject(int i) : id(i) {  
 cout << "Constructor: " << id << endl;  
 }  
   
 // Destructor  
 ~TraceableObject() {  
 cout << "Destructor: " << id << endl;  
 }  
   
 // Copy constructor  
 TraceableObject(const TraceableObject& other) : id(other.id) {  
 cout << "Copy constructor: " << id << endl;  
 }  
   
 // Move constructor  
 TraceableObject(TraceableObject&& other) noexcept : id(other.id) {  
 cout << "Move constructor: " << id << endl;  
 other.id = -1; // Mark as moved  
 }  
   
 // For display  
 int getId() const { return id; }  
};  
  
// Function that takes by value - copy or move happens  
void takeByValue(TraceableObject obj) {  
 cout << "Inside takeByValue: id = " << obj.getId() << endl;  
}  
  
// Function that returns by value - might use move or RVO/NRVO  
TraceableObject createObject() {  
 TraceableObject obj(42);  
 cout << "createObject - before return" << endl;  
 return obj; // Could use NRVO (Named Return Value Optimization)  
}  
  
TraceableObject createObjectWithMove() {  
 TraceableObject obj(43);  
 cout << "createObjectWithMove - before return" << endl;  
 return move(obj); // Explicitly moving, prevents NRVO  
}  
  
int main() {  
 cout << "1. Local object creation:" << endl;  
 TraceableObject local(1);  
   
 cout << "\n2. Copy construction:" << endl;  
 TraceableObject copy = local;  
   
 cout << "\n3. Move construction:" << endl;  
 TraceableObject moved = move(local);  
   
 cout << "\n4. Passing to function by value:" << endl;  
 TraceableObject arg(2);  
 takeByValue(arg); // Makes a copy  
   
 cout << "\n5. Passing temporary to function:" << endl;  
 takeByValue(TraceableObject(3)); // Move construction usually happens  
   
 cout << "\n6. Return value optimization:" << endl;  
 TraceableObject fromFunc = createObject(); // Might be constructed in place  
   
 cout << "\n7. Returning with explicit move:" << endl;  
 TraceableObject fromFuncMoved = createObjectWithMove();  
   
 cout << "\n8. Adding to a vector (may need to reallocate):" << endl;  
 vector<TraceableObject> vec;  
 vec.push\_back(TraceableObject(4)); // Move construction from temporary  
 TraceableObject pushMe(5);  
 vec.push\_back(pushMe); // Copy construction  
 vec.push\_back(move(pushMe)); // Move construction  
   
 cout << "\n9. End of program - all objects will be destroyed" << endl;  
 return 0;  
}

### Best Practices

1. **Always mark move operations as noexcept**
   * Enables optimizations in standard containers
2. **Be consistent with the Rule of Three/Five/Zero**
   * If you define one special member function, consider if you need the others
3. **Avoid unnecessary copying when moving is sufficient**
   * Use std::move when appropriate, especially for returning large objects
4. **Remember that copy operations must not modify their source**
   * While move operations should leave the source in a valid but unspecified state
5. **Make move operations efficient**
   * They should be much faster than copying, otherwise there’s no benefit
6. **Consider special cases where copy and move logic differ**
   * Reference counting, internal bookkeeping, etc.
7. **Watch out for self-assignment**
   * Both copy and move assignment operators should handle self-assignment correctly

### Example: Move vs Copy in Standard Library

#include <iostream>  
#include <string>  
#include <vector>  
#include <memory>  
using namespace std;  
  
int main() {  
 // String examples  
 string original = "This is a fairly long string that would require allocation";  
   
 // Copy vs Move with strings  
 cout << "--- String copy vs move ---" << endl;  
 string strCopy = original; // Copy constructor  
 cout << "Original after copy: " << original << endl;  
   
 string strMove = move(original); // Move constructor  
 cout << "Original after move: '" << original << "'" << endl;  
 cout << "Moved-to string: " << strMove << endl;  
   
 // Vector examples  
 cout << "\n--- Vector copy vs move ---" << endl;  
 vector<int> vecOriginal = {1, 2, 3, 4, 5};  
 cout << "Original size: " << vecOriginal.size() << endl;  
   
 vector<int> vecCopy = vecOriginal; // Copy constructor  
 cout << "Original size after copy: " << vecOriginal.size() << endl;  
   
 vector<int> vecMove = move(vecOriginal); // Move constructor  
 cout << "Original size after move: " << vecOriginal.size() << endl;  
 cout << "Moved-to vector size: " << vecMove.size() << endl;  
   
 // Unique pointer (can only be moved, not copied)  
 cout << "\n--- Unique pointer (move-only) ---" << endl;  
 unique\_ptr<int> ptrOriginal = make\_unique<int>(42);  
 cout << "Original pointer value: " << \*ptrOriginal << endl;  
   
 // unique\_ptr<int> ptrCopy = ptrOriginal; // Error: copy not allowed  
   
 unique\_ptr<int> ptrMove = move(ptrOriginal);  
 cout << "Moved-to pointer value: " << \*ptrMove << endl;  
 cout << "Original pointer after move is null? " << (ptrOriginal == nullptr ? "yes" : "no") << endl;  
   
 return 0;  
}

### Example: Custom Copy and Move Operations with Complex Resources

#include <iostream>  
#include <memory>  
#include <string>  
using namespace std;  
  
// A resource class that maintains a reference count  
class SharedResource {  
private:  
 string name;  
 int\* refCount;  
   
public:  
 SharedResource(const string& n) : name(n) {  
 refCount = new int(1);  
 cout << "Resource created: " << name << " (refCount = " << \*refCount << ")" << endl;  
 }  
   
 ~SharedResource() {  
 (\*refCount)--;  
 cout << "Reference removed from: " << name << " (refCount = " << \*refCount << ")" << endl;  
 if (\*refCount == 0) {  
 cout << "Deleting resource: " << name << endl;  
 delete refCount;  
 }  
 }  
   
 // Copy constructor increases reference count  
 SharedResource(const SharedResource& other)  
 : name(other.name), refCount(other.refCount) {  
 (\*refCount)++;  
 cout << "Resource copied: " << name << " (refCount = " << \*refCount << ")" << endl;  
 }  
   
 // Copy assignment operator  
 SharedResource& operator=(const SharedResource& other) {  
 cout << "Copy assignment for: " << name << " = " << other.name << endl;  
 if (this != &other) {  
 // Decrease our current reference  
 (\*refCount)--;  
 if (\*refCount == 0) {  
 cout << "Deleting old resource: " << name << endl;  
 delete refCount;  
 }  
   
 // Point to the new resource and increase its count  
 name = other.name;  
 refCount = other.refCount;  
 (\*refCount)++;  
 }  
 return \*this;  
 }  
   
 // Move constructor takes ownership without changing reference count  
 SharedResource(SharedResource&& other) noexcept  
 : name(move(other.name)), refCount(other.refCount) {  
 other.refCount = new int(1); // Source gets a new reference count  
 cout << "Resource moved: " << name << " (refCount unchanged = " << \*refCount << ")" << endl;  
 }  
   
 // Move assignment operator  
 SharedResource& operator=(SharedResource&& other) noexcept {  
 cout << "Move assignment for: " << name << " = " << other.name << endl;  
 if (this != &other) {  
 // Decrease our current reference  
 (\*refCount)--;  
 if (\*refCount == 0) {  
 cout << "Deleting old resource: " << name << endl;  
 delete refCount;  
 }  
   
 // Take ownership of the other's resource  
 name = move(other.name);  
 refCount = other.refCount;  
   
 // Give other a new reference count  
 other.refCount = new int(1);  
 }  
 return \*this;  
 }  
   
 string getName() const { return name; }  
 int getRefCount() const { return \*refCount; }  
};  
  
int main() {  
 cout << "Creating original resources:" << endl;  
 SharedResource res1("Resource A");  
 SharedResource res2("Resource B");  
   
 cout << "\nCopying resources:" << endl;  
 SharedResource res3 = res1; // Copy constructor  
 cout << "res1: " << res1.getName() << ", refCount = " << res1.getRefCount() << endl;  
 cout << "res3: " << res3.getName() << ", refCount = " << res3.getRefCount() << endl;  
   
 cout << "\nMoving resources:" << endl;  
 SharedResource res4 = move(res2); // Move constructor  
 cout << "res2: " << res2.getName() << ", refCount = " << res2.getRefCount() << endl;  
 cout << "res4: " << res4.getName() << ", refCount = " << res4.getRefCount() << endl;  
   
 cout << "\nCopy assignment:" << endl;  
 res2 = res1; // Copy assignment  
 cout << "res1: " << res1.getName() << ", refCount = " << res1.getRefCount() << endl;  
 cout << "res2: " << res2.getName() << ", refCount = " << res2.getRefCount() << endl;  
   
 cout << "\nMove assignment:" << endl;  
 res4 = move(res3); // Move assignment  
 cout << "res3: " << res3.getName() << ", refCount = " << res3.getRefCount() << endl;  
 cout << "res4: " << res4.getName() << ", refCount = " << res4.getRefCount() << endl;  
   
 cout << "\nEnd of program - all resources will be cleaned up" << endl;  
 return 0;  
}

### Conclusion

Understanding the differences between copy and move operations is essential for writing efficient C++ code, especially when dealing with classes that manage resources. By implementing both types correctly and following the appropriate rules (Zero, Three, or Five), you can create classes that perform well and manage resources safely.

# Chapter 8: Standard Template Library (STL) (Part 1)

## 8.1 STL Overview

The Standard Template Library (STL) is a powerful set of C++ template classes and functions that provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

### Core Components of the STL

The STL consists of three primary components:

1. **Containers**: Objects that store data
2. **Algorithms**: Functions that operate on containers
3. **Iterators**: Objects that connect algorithms to containers

### Benefits of Using the STL

1. **Reliability**: The STL is extensively tested and optimized
2. **Efficiency**: Implementations are highly optimized for performance
3. **Productivity**: Reduces development time significantly
4. **Reusability**: Promotes code reuse through generic programming
5. **Standardization**: Part of the C++ standard library

### Overview of STL Components

#include <iostream>  
#include <vector> // Container  
#include <algorithm> // Algorithms  
#include <string>  
using namespace std;  
  
int main() {  
 // Create a vector container  
 vector<int> numbers = {5, 2, 8, 1, 9};  
   
 // Use an algorithm with iterators  
 sort(numbers.begin(), numbers.end()); // Sort in ascending order  
   
 // Iterate through the container using an iterator  
 cout << "Sorted numbers: ";  
 for (auto it = numbers.begin(); it != numbers.end(); ++it) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 // Find an element using an algorithm and iterators  
 auto it = find(numbers.begin(), numbers.end(), 8);  
 if (it != numbers.end()) {  
 cout << "Found: " << \*it << " at position: "   
 << (it - numbers.begin()) << endl;  
 }  
   
 return 0;  
}

### Headers for STL Components

| Component | Headers |
| --- | --- |
| **Containers** | <vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <unordered\_map>, <unordered\_set> |
| **Algorithms** | <algorithm>, <numeric> |
| **Iterators** | Built into containers |
| **Function Objects** | <functional> |
| **Adaptors** | Built into their respective headers |

### STL Design Philosophy

The STL was designed based on the concept of generic programming, with these key principles:

1. **Separation of data and algorithms**: Algorithms are not part of containers
2. **Extension through composition**: Components can be combined flexibly
3. **Type safety**: Compile-time type checking provides safety without runtime overhead
4. **Efficiency**: Performance is a key consideration in all implementations

## 8.2 Containers

STL containers are objects that store collections of other objects. They are implemented as class templates, allowing them to hold elements of any type that meets certain requirements.

### Types of STL Containers

1. **Sequence containers**: Store elements in a linear sequence
   * vector
   * list
   * deque
   * array (C++11)
   * forward\_list (C++11)
2. **Associative containers**: Store elements in ordered structures
   * set
   * multiset
   * map
   * multimap
3. **Unordered associative containers** (C++11): Store elements in unordered structures
   * unordered\_set
   * unordered\_multiset
   * unordered\_map
   * unordered\_multimap
4. **Container adaptors**: Provide a different interface for specific sequence containers
   * stack
   * queue
   * priority\_queue

### Common Container Operations

Most STL containers provide these operations:

* **Constructors**: Default, copy, move (C++11), and initializer list (C++11)
* **Assignment operators**: Copy, move (C++11), and initializer list (C++11)
* **Size operations**: size(), empty(), max\_size()
* **Element access**: Depends on container type
* **Iterators**: begin(), end(), rbegin(), rend()
* **Modifiers**: insert(), erase(), clear()

Here’s a basic comparison of some common containers:

#include <iostream>  
#include <vector>  
#include <list>  
#include <deque>  
#include <stack>  
using namespace std;  
  
template <typename Container>  
void displayContainer(const Container& c, const string& name) {  
 cout << name << ": ";  
 for (const auto& item : c) {  
 cout << item << " ";  
 }  
 cout << endl;  
}  
  
int main() {  
 // Creating different containers  
 vector<int> vec = {1, 2, 3, 4, 5};  
 list<int> lst = {1, 2, 3, 4, 5};  
 deque<int> deq = {1, 2, 3, 4, 5};  
   
 // Display initial state  
 displayContainer(vec, "vector");  
 displayContainer(lst, "list");  
 displayContainer(deq, "deque");  
   
 // Adding elements to the front  
 // vec.insert(vec.begin(), 0); // Potentially expensive for vector  
 lst.push\_front(0); // Efficient for list  
 deq.push\_front(0); // Efficient for deque  
   
 // Adding elements to the back  
 vec.push\_back(6); // Efficient for vector  
 lst.push\_back(6); // Efficient for list  
 deq.push\_back(6); // Efficient for deque  
   
 // Display after modifications  
 cout << "\nAfter modifications:" << endl;  
 displayContainer(vec, "vector");  
 displayContainer(lst, "list");  
 displayContainer(deq, "deque");  
   
 // Create a stack using deque as the underlying container  
 stack<int> stk(deq);  
   
 cout << "\nStack (top->bottom): ";  
 while (!stk.empty()) {  
 cout << stk.top() << " ";  
 stk.pop();  
 }  
 cout << endl;  
   
 return 0;  
}

### 8.2.1 Vector

std::vector is a sequence container that represents a dynamic array, capable of growing or shrinking in size. It provides random access to elements and is the most commonly used STL container.

#### Key Features

* **Dynamic size**: Can grow or shrink automatically
* **Random access**: Constant-time access to elements by index
* **Contiguous storage**: Elements are stored in contiguous memory
* **Efficient insertion/deletion at the end**: Amortized constant time
* **Inefficient insertion/deletion elsewhere**: Linear time

#### Basic Vector Operations

#include <iostream>  
#include <vector>  
#include <algorithm> // For algorithms like sort, find  
using namespace std;  
  
int main() {  
 // Different ways to create a vector  
 vector<int> vec1; // Empty vector  
 vector<int> vec2(5, 0); // Vector with 5 elements, all 0  
 vector<int> vec3 = {10, 20, 30, 40}; // Initializer list (C++11)  
 vector<int> vec4(vec3); // Copy constructor  
   
 // Size operations  
 cout << "vec1 size: " << vec1.size() << endl;  
 cout << "vec2 size: " << vec2.size() << endl;  
 cout << "vec3 size: " << vec3.size() << endl;  
 cout << "vec3 capacity: " << vec3.capacity() << endl; // Number of elements it can hold  
   
 // Adding elements  
 vec1.push\_back(100);  
 vec1.push\_back(200);  
 vec1.push\_back(300);  
   
 // Accessing elements  
 cout << "vec1[0]: " << vec1[0] << endl; // No bounds checking  
 cout << "vec1.at(1): " << vec1.at(1) << endl; // With bounds checking  
 cout << "vec1.front(): " << vec1.front() << endl;  
 cout << "vec1.back(): " << vec1.back() << endl;  
   
 // Iteration  
 cout << "vec3 elements: ";  
 for (const auto& item : vec3) { // Range-based for loop (C++11)  
 cout << item << " ";  
 }  
 cout << endl;  
   
 // Insert and erase  
 vec3.insert(vec3.begin() + 2, 25); // Insert 25 at position 2  
   
 cout << "vec3 after insert: ";  
 for (size\_t i = 0; i < vec3.size(); ++i) { // Traditional iteration  
 cout << vec3[i] << " ";  
 }  
 cout << endl;  
   
 vec3.erase(vec3.begin() + 1); // Erase element at position 1  
   
 cout << "vec3 after erase: ";  
 for (auto it = vec3.begin(); it != vec3.end(); ++it) { // Iterator-based loop  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 // Using algorithms  
 sort(vec3.begin(), vec3.end()); // Sort in ascending order  
   
 cout << "vec3 after sorting: ";  
 for (const auto& item : vec3) {  
 cout << item << " ";  
 }  
 cout << endl;  
   
 // Finding elements  
 auto findIt = find(vec3.begin(), vec3.end(), 30);  
 if (findIt != vec3.end()) {  
 cout << "Found element: " << \*findIt << endl;  
 }  
   
 // Clearing the vector  
 vec3.clear();  
 cout << "vec3 size after clearing: " << vec3.size() << endl;  
   
 return 0;  
}

#### Vector Performance Considerations

* **Memory reallocation**: When a vector grows beyond its capacity, it reallocates memory
* **Reserve**: Use reserve() to prevent multiple reallocations
* **Shrink to fit**: Use shrink\_to\_fit() to reduce capacity to fit size

#include <iostream>  
#include <vector>  
using namespace std;  
  
int main() {  
 // Demonstrating vector capacity and reallocation  
 vector<int> numbers;  
   
 cout << "Initial capacity: " << numbers.capacity() << endl;  
   
 // Add elements and observe capacity changes  
 for (int i = 0; i < 100; ++i) {  
 numbers.push\_back(i);  
   
 // Print capacity every time it changes  
 if (numbers.size() == numbers.capacity()) {  
 cout << "Size: " << numbers.size()   
 << ", Capacity: " << numbers.capacity() << endl;  
 }  
 }  
   
 // Using reserve to pre-allocate memory  
 vector<int> efficientVec;  
 efficientVec.reserve(1000); // Pre-allocate space for 1000 elements  
   
 cout << "\nefficient vector - Initial capacity after reserve: "   
 << efficientVec.capacity() << endl;  
   
 for (int i = 0; i < 100; ++i) {  
 efficientVec.push\_back(i);  
 }  
   
 cout << "efficient vector - Size: " << efficientVec.size()   
 << ", Capacity: " << efficientVec.capacity() << endl;  
   
 // Using shrink\_to\_fit to reduce capacity  
 efficientVec.shrink\_to\_fit();  
 cout << "After shrink\_to\_fit - Size: " << efficientVec.size()   
 << ", Capacity: " << efficientVec.capacity() << endl;  
   
 return 0;  
}

#### Vector of Custom Objects

#include <iostream>  
#include <vector>  
#include <string>  
#include <algorithm>  
using namespace std;  
  
// Custom class  
class Person {  
private:  
 string name;  
 int age;  
   
public:  
 Person(const string& n, int a) : name(n), age(a) {}  
   
 // Getters  
 string getName() const { return name; }  
 int getAge() const { return age; }  
   
 // For sorting and comparison  
 bool operator<(const Person& other) const {  
 return age < other.age;  
 }  
};  
  
// For displaying Person objects  
ostream& operator<<(ostream& os, const Person& person) {  
 return os << person.getName() << " (" << person.getAge() << ")";  
}  
  
int main() {  
 // Create a vector of Person objects  
 vector<Person> people;  
   
 // Add elements  
 people.push\_back(Person("Alice", 25));  
 people.push\_back(Person("Bob", 30));  
 people.push\_back(Person("Charlie", 22));  
 people.emplace\_back("David", 35); // Construct in place (more efficient)  
   
 // Display people  
 cout << "People:" << endl;  
 for (const auto& person : people) {  
 cout << person << endl;  
 }  
   
 // Sort by age (using Person::operator<)  
 sort(people.begin(), people.end());  
   
 cout << "\nPeople sorted by age:" << endl;  
 for (const auto& person : people) {  
 cout << person << endl;  
 }  
   
 // Sort by name using a lambda function  
 sort(people.begin(), people.end(),   
 [](const Person& a, const Person& b) {  
 return a.getName() < b.getName();  
 });  
   
 cout << "\nPeople sorted by name:" << endl;  
 for (const auto& person : people) {  
 cout << person << endl;  
 }  
   
 return 0;  
}

### 8.2.2 List

std::list is a sequence container that implements a doubly-linked list. It allows efficient insertion and removal of elements anywhere within the sequence.

#### Key Features

* **Non-contiguous memory**: Elements are not stored in contiguous memory
* **Bidirectional iterators**: Can move forward and backward
* **Efficient insertion/deletion anywhere**: Constant time
* **No random access**: Cannot use [] or at()
* **Slow element search**: Linear time complexity

#### Basic List Operations

#include <iostream>  
#include <list>  
#include <string>  
#include <algorithm> // for find  
using namespace std;  
  
int main() {  
 // Create a list  
 list<int> numbers = {10, 20, 30, 40, 50};  
   
 // Display the list  
 cout << "List contents: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Add elements  
 numbers.push\_back(60); // Add to the end  
 numbers.push\_front(5); // Add to the beginning  
   
 // Display after adding  
 cout << "After adding elements: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Insert in the middle  
 auto it = find(numbers.begin(), numbers.end(), 30);  
 if (it != numbers.end()) {  
 numbers.insert(it, 25); // Insert before the found element  
 }  
   
 // Display after inserting  
 cout << "After inserting 25: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Remove elements  
 numbers.remove(20); // Remove all occurrences of 20  
   
 numbers.pop\_front(); // Remove the first element  
 numbers.pop\_back(); // Remove the last element  
   
 // Display after removing  
 cout << "After removing elements: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Size operations  
 cout << "List size: " << numbers.size() << endl;  
 cout << "Is empty? " << (numbers.empty() ? "Yes" : "No") << endl;  
   
 // Sort the list  
 numbers.sort(); // Lists have their own sort method!  
   
 cout << "After sorting: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Reverse the list  
 numbers.reverse(); // Lists have their own reverse method!  
   
 cout << "After reversing: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Clear the list  
 numbers.clear();  
 cout << "Size after clearing: " << numbers.size() << endl;  
   
 return 0;  
}

#### Unique List Operations

Lists have specialized member functions that other containers don’t have:

#include <iostream>  
#include <list>  
using namespace std;  
  
int main() {  
 // Create two lists  
 list<int> list1 = {10, 20, 30, 20, 40, 30, 30, 50};  
 list<int> list2 = {5, 15, 25, 35, 45};  
   
 cout << "Original list1: ";  
 for (const auto& item : list1) {  
 cout << item << " ";  
 }  
 cout << endl;  
   
 // Remove duplicate elements  
 list1.unique();  
   
 cout << "After unique(): ";  
 for (const auto& item : list1) {  
 cout << item << " ";  
 }  
 cout << endl;  
   
 // Sort both lists  
 list1.sort();  
 list2.sort();  
   
 cout << "list1 sorted: ";  
 for (const auto& item : list1) {  
 cout << item << " ";  
 }  
 cout << endl;  
   
 cout << "list2 sorted: ";  
 for (const auto& item : list2) {  
 cout << item << " ";  
 }  
 cout << endl;  
   
 // Merge two sorted lists  
 list1.merge(list2); // list2 will be empty after merging  
   
 cout << "After merge: ";  
 for (const auto& item : list1) {  
 cout << item << " ";  
 }  
 cout << endl;  
   
 cout << "list2 size after merge: " << list2.size() << endl;  
   
 // Splice (insert elements from another list)  
 list<int> source = {100, 200, 300};  
 auto it = list1.begin();  
 advance(it, 3); // Move iterator to the 4th position  
   
 list1.splice(it, source); // Insert all elements from source at position it  
   
 cout << "After splice: ";  
 for (const auto& item : list1) {  
 cout << item << " ";  
 }  
 cout << endl;  
   
 cout << "source size after splice: " << source.size() << endl;  
   
 return 0;  
}

#### When to Use List

* When you need frequent insertions and deletions at arbitrary positions
* When elements should not be relocated (pointers/references remain valid)
* When you don’t need random access to elements
* When the specific operations (splice, unique, etc.) are needed

#### List vs Vector Performance

#include <iostream>  
#include <vector>  
#include <list>  
#include <chrono>  
#include <algorithm>  
using namespace std;  
using namespace chrono;  
  
template<typename Container>  
void performanceTest(const string& containerName) {  
 const int NUM\_ELEMENTS = 100000;  
   
 Container container;  
   
 // Test insertion at the end  
 auto startTime = high\_resolution\_clock::now();  
 for (int i = 0; i < NUM\_ELEMENTS; ++i) {  
 container.push\_back(i);  
 }  
 auto endTime = high\_resolution\_clock::now();  
 auto durationPushBack = duration\_cast<milliseconds>(endTime - startTime).count();  
   
 cout << containerName << " - Push back time: " << durationPushBack << "ms" << endl;  
   
 // Test insertion at the beginning  
 Container container2;  
 startTime = high\_resolution\_clock::now();  
 for (int i = 0; i < NUM\_ELEMENTS / 10; ++i) { // Using fewer elements for vector  
 container2.insert(container2.begin(), i);  
 }  
 endTime = high\_resolution\_clock::now();  
 auto durationInsertFront = duration\_cast<milliseconds>(endTime - startTime).count();  
   
 cout << containerName << " - Insert at front time: " << durationInsertFront << "ms" << endl;  
   
 // Test iteration (if not empty)  
 if (!container.empty()) {  
 startTime = high\_resolution\_clock::now();  
 typename Container::value\_type sum = 0;  
 for (const auto& item : container) {  
 sum += item;  
 }  
 endTime = high\_resolution\_clock::now();  
 auto durationIteration = duration\_cast<milliseconds>(endTime - startTime).count();  
   
 cout << containerName << " - Iteration time: " << durationIteration << "ms" << endl;  
 }  
   
 cout << endl;  
}  
  
int main() {  
 cout << "Performance comparison between vector and list:" << endl;  
 cout << "-------------------------------------------" << endl;  
   
 performanceTest<vector<int>>("vector");  
 performanceTest<list<int>>("list");  
   
 return 0;  
}

### 8.2.3 Deque

std::deque (double-ended queue) is a sequence container that allows fast insertion and deletion at both its beginning and end. Unlike vector, deque elements are not stored in contiguous memory.

#### Key Features

* **Random access**: Constant-time access to elements by index
* **Efficient insertion/deletion at both ends**: Constant time
* **Inefficient insertion/deletion in the middle**: Linear time
* **Non-contiguous memory**: Elements are stored in multiple chunks
* **No reallocation**: No need to reserve capacity

#### Basic Deque Operations

#include <iostream>  
#include <deque>  
#include <string>  
using namespace std;  
  
int main() {  
 // Create a deque  
 deque<int> numbers;  
   
 // Add elements to both ends  
 numbers.push\_back(30); // Add to the end  
 numbers.push\_back(40);  
 numbers.push\_front(20); // Add to the beginning  
 numbers.push\_front(10);  
   
 // Display contents  
 cout << "Deque contents: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Access elements  
 cout << "First element: " << numbers.front() << endl;  
 cout << "Last element: " << numbers.back() << endl;  
 cout << "Element at position 2: " << numbers[2] << endl;  
 cout << "Element at position 1: " << numbers.at(1) << endl;  
   
 // Insert in the middle  
 auto it = numbers.begin() + 2;  
 numbers.insert(it, 25);  
   
 cout << "After inserting 25: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Remove elements from both ends  
 numbers.pop\_front(); // Remove from the beginning  
 numbers.pop\_back(); // Remove from the end  
   
 cout << "After removing from both ends: ";  
 for (size\_t i = 0; i < numbers.size(); ++i) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
   
 // Size operations  
 cout << "Size: " << numbers.size() << endl;  
 cout << "Is empty: " << (numbers.empty() ? "Yes" : "No") << endl;  
   
 // Resize the deque  
 numbers.resize(6, 0); // Resize to 6 elements, fill new positions with 0  
   
 cout << "After resizing: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Clear the deque  
 numbers.clear();  
 cout << "Size after clearing: " << numbers.size() << endl;  
   
 return 0;  
}

#### Using a Deque as a Double-Ended Queue

This is the natural use case for a deque:

#include <iostream>  
#include <deque>  
#include <string>  
using namespace std;  
  
// Simple double-ended queue demonstration  
class ProcessQueue {  
private:  
 deque<string> processes;  
   
public:  
 // Add high-priority process to the front  
 void addHighPriority(const string& process) {  
 processes.push\_front(process);  
 cout << "Added high-priority process: " << process << endl;  
 }  
   
 // Add normal process to the back  
 void addNormal(const string& process) {  
 processes.push\_back(process);  
 cout << "Added normal process: " << process << endl;  
 }  
   
 // Process next (from the front)  
 void processNext() {  
 if (!processes.empty()) {  
 cout << "Processing: " << processes.front() << endl;  
 processes.pop\_front();  
 } else {  
 cout << "No processes to handle." << endl;  
 }  
 }  
   
 // Display queue  
 void displayQueue() const {  
 if (processes.empty()) {  
 cout << "Queue is empty." << endl;  
 return;  
 }  
   
 cout << "Current process queue (front->back): ";  
 for (const auto& proc : processes) {  
 cout << proc << " ";  
 }  
 cout << endl;  
 }  
   
 // Queue size  
 size\_t size() const {  
 return processes.size();  
 }  
};  
  
int main() {  
 ProcessQueue queue;  
   
 queue.addNormal("Background task");  
 queue.addNormal("File indexing");  
 queue.addHighPriority("User input");  
 queue.addNormal("Cleanup");  
 queue.addHighPriority("System alert");  
   
 queue.displayQueue();  
   
 cout << "\nProcessing queue:" << endl;  
 while (queue.size() > 0) {  
 queue.processNext();  
 }  
   
 queue.displayQueue();  
   
 return 0;  
}

#### When to Use Deque

* When you need frequent insertions and deletions at both ends
* When you need random access to elements
* When memory efficiency is important (no overallocation like vector)
* When relocating elements is undesirable

#### Comparison: Vector vs List vs Deque

| Operation | Vector | List | Deque |
| --- | --- | --- | --- |
| Random Access | O(1) | O(n) | O(1) |
| Insert/Delete at beginning | O(n) | O(1) | O(1) |
| Insert/Delete at end | Amortized O(1) | O(1) | O(1) |
| Insert/Delete in middle | O(n) | O(1) + search time | O(n) |
| Memory Layout | Contiguous | Non-contiguous | Chunked blocks |
| Iterator Invalidation on Insert/Delete | Yes (at/after insertion point) | No | Possibly |

### 8.2.4 Stack

std::stack is a container adapter that provides a LIFO (Last-In, First-Out) data structure. It is implemented as an adapter over other containers, typically deque by default.

#### Key Features

* **LIFO (Last-In, First-Out)**: Elements are inserted and removed from the same end
* **Container Adapter**: Built on top of other containers
* **Limited Interface**: Only allows operations appropriate for stacks
* **No iterators**: Can’t traverse all elements

#### Basic Stack Operations

#include <iostream>  
#include <stack>  
#include <string>  
using namespace std;  
  
int main() {  
 // Create a stack with default container (deque)  
 stack<int> numbers;  
   
 // Push elements onto the stack  
 numbers.push(10);  
 numbers.push(20);  
 numbers.push(30);  
 numbers.push(40);  
   
 cout << "Stack size: " << numbers.size() << endl;  
 cout << "Top element: " << numbers.top() << endl;  
   
 // Pop elements from the stack  
 cout << "Popping elements: ";  
 while (!numbers.empty()) {  
 cout << numbers.top() << " ";  
 numbers.pop();  
 }  
 cout << endl;  
   
 cout << "Stack empty? " << (numbers.empty() ? "Yes" : "No") << endl;  
   
 // Creating a stack with a different underlying container  
 stack<string, vector<string>> strStack;  
   
 strStack.push("First");  
 strStack.push("Second");  
 strStack.push("Third");  
   
 cout << "String stack top: " << strStack.top() << endl;  
   
 return 0;  
}

#### Real-World Stack Applications

#include <iostream>  
#include <stack>  
#include <string>  
#include <cctype> // for isdigit  
using namespace std;  
  
// Example 1: Check for balanced parentheses  
bool areParenthesesBalanced(const string& expr) {  
 stack<char> s;  
   
 for (char c : expr) {  
 if (c == '(' || c == '[' || c == '{') {  
 // Push opening bracket to stack  
 s.push(c);  
 } else if (c == ')' || c == ']' || c == '}') {  
 // Check for matching opening bracket  
 if (s.empty()) {  
 return false; // Unmatched closing bracket  
 }  
   
 char top = s.top();  
 if ((c == ')' && top == '(') ||  
 (c == ']' && top == '[') ||  
 (c == '}' && top == '{')) {  
 s.pop(); // Matched, remove the opening bracket  
 } else {  
 return false; // Mismatched brackets  
 }  
 }  
 }  
   
 // If stack is empty, all brackets are matched  
 return s.empty();  
}  
  
// Example 2: Evaluate postfix expression  
int evaluatePostfix(const string& expr) {  
 stack<int> s;  
   
 for (char c : expr) {  
 // If character is a digit, push it to the stack  
 if (isdigit(c)) {  
 s.push(c - '0'); // Convert char to integer  
 }   
 // If character is an operator, pop two elements and apply operator  
 else if (c == '+' || c == '-' || c == '\*' || c == '/') {  
 // Need at least two operands  
 if (s.size() < 2) {  
 cerr << "Invalid expression" << endl;  
 return 0;  
 }  
   
 int operand2 = s.top();  
 s.pop();  
 int operand1 = s.top();  
 s.pop();  
   
 switch (c) {  
 case '+': s.push(operand1 + operand2); break;  
 case '-': s.push(operand1 - operand2); break;  
 case '\*': s.push(operand1 \* operand2); break;  
 case '/': s.push(operand1 / operand2); break;  
 }  
 }  
 }  
   
 // Result should be the only item left in the stack  
 if (s.size() != 1) {  
 cerr << "Invalid expression" << endl;  
 return 0;  
 }  
   
 return s.top();  
}  
  
// Example 3: Function call simulation  
void simulateFunctionCalls() {  
 stack<string> callStack;  
   
 cout << "Function call simulation:" << endl;  
   
 // Main calls function A  
 callStack.push("main");  
 cout << "Entering function: main" << endl;  
   
 // Function A calls function B  
 callStack.push("functionA");  
 cout << "Entering function: functionA" << endl;  
   
 // Function B calls function C  
 callStack.push("functionB");  
 cout << "Entering function: functionB" << endl;  
   
 callStack.push("functionC");  
 cout << "Entering function: functionC" << endl;  
   
 // Functions return in reverse order  
 cout << "\nFunctions returning:" << endl;  
 while (!callStack.empty()) {  
 cout << "Returning from function: " << callStack.top() << endl;  
 callStack.pop();  
 }  
}  
  
int main() {  
 // Test parentheses balancing  
 string expr1 = "((a+b)\*(c-d))";  
 string expr2 = "{[a+b]\*(c+d)}";  
 string expr3 = "((a+b)\*(c-d)";  
   
 cout << expr1 << " is " << (areParenthesesBalanced(expr1) ? "balanced" : "not balanced") << endl;  
 cout << expr2 << " is " << (areParenthesesBalanced(expr2) ? "balanced" : "not balanced") << endl;  
 cout << expr3 << " is " << (areParenthesesBalanced(expr3) ? "balanced" : "not balanced") << endl;  
   
 // Test postfix evaluation  
 string postfix = "534\*+"; // Represents 5+(3\*4) = 17  
 cout << "\nEvaluating postfix expression: " << postfix << endl;  
 cout << "Result: " << evaluatePostfix(postfix) << endl;  
   
 // Simulate function calls  
 cout << endl;  
 simulateFunctionCalls();  
   
 return 0;  
}

#### When to Use Stack

* When you need LIFO (Last-In, First-Out) behavior
* When you need to track function calls, recursive algorithms
* When you need to maintain history/state for undo operations
* For parsing expressions, syntax validation, etc.

#### Custom Stack Implementation

While you typically use the STL std::stack, here’s a basic implementation to understand how it works internally:

#include <iostream>  
#include <vector>  
#include <stdexcept>  
using namespace std;  
  
template <typename T, typename Container = vector<T>>  
class MyStack {  
private:  
 Container container; // Underlying container  
  
public:  
 // Check if stack is empty  
 bool empty() const {  
 return container.empty();  
 }  
   
 // Get size of stack  
 size\_t size() const {  
 return container.size();  
 }  
   
 // Access top element  
 T& top() {  
 if (empty()) {  
 throw runtime\_error("Stack is empty");  
 }  
 return container.back();  
 }  
   
 const T& top() const {  
 if (empty()) {  
 throw runtime\_error("Stack is empty");  
 }  
 return container.back();  
 }  
   
 // Add element to top  
 void push(const T& value) {  
 container.push\_back(value);  
 }  
   
 void push(T&& value) {  
 container.push\_back(move(value));  
 }  
   
 // Remove top element  
 void pop() {  
 if (empty()) {  
 throw runtime\_error("Stack is empty");  
 }  
 container.pop\_back();  
 }  
   
 // Swap contents with another stack  
 void swap(MyStack& other) noexcept {  
 container.swap(other.container);  
 }  
};  
  
int main() {  
 MyStack<int> stack;  
   
 // Push elements  
 for (int i = 1; i <= 5; ++i) {  
 stack.push(i \* 10);  
 cout << "Pushed: " << stack.top() << endl;  
 }  
   
 cout << "\nStack contents (top to bottom): ";  
 while (!stack.empty()) {  
 cout << stack.top() << " ";  
 stack.pop();  
 }  
 cout << endl;  
   
 return 0;  
}

# Chapter 8: Standard Template Library (STL) (Part 2)

## 8.2.5 Queue

std::queue is a container adapter that gives the functionality of a first-in, first-out (FIFO) data structure. Like std::stack, it is built on top of other STL containers, with std::deque being the default.

### Key Features

* **FIFO (First-In, First-Out)**: Elements are inserted at the back and removed from the front
* **Container Adapter**: Built on top of other containers (typically deque)
* **Limited Interface**: Only allows operations appropriate for queues
* **No iterators**: Can’t traverse all elements

### Basic Queue Operations

#include <iostream>  
#include <queue>  
#include <string>  
using namespace std;  
  
int main() {  
 // Create a queue with default container (deque)  
 queue<int> numbers;  
   
 // Add elements to the queue (at the back)  
 numbers.push(10);  
 numbers.push(20);  
 numbers.push(30);  
 numbers.push(40);  
   
 cout << "Queue size: " << numbers.size() << endl;  
 cout << "Front element: " << numbers.front() << endl;  
 cout << "Back element: " << numbers.back() << endl;  
   
 // Remove elements from the queue (from the front)  
 cout << "Processing queue: ";  
 while (!numbers.empty()) {  
 cout << numbers.front() << " ";  
 numbers.pop();  
 }  
 cout << endl;  
   
 cout << "Queue empty? " << (numbers.empty() ? "Yes" : "No") << endl;  
   
 // Creating a queue with a different underlying container  
 queue<string, vector<string>> strQueue;  
   
 strQueue.push("First");  
 strQueue.push("Second");  
 strQueue.push("Third");  
   
 cout << "String queue front: " << strQueue.front() << endl;  
 cout << "String queue back: " << strQueue.back() << endl;  
   
 return 0;  
}

### Real-World Queue Applications

#include <iostream>  
#include <queue>  
#include <string>  
#include <thread>  
#include <chrono>  
using namespace std;  
  
// Example 1: Print queue simulation  
class PrintQueue {  
private:  
 queue<string> documents;  
   
public:  
 void addDocument(const string& doc) {  
 documents.push(doc);  
 cout << "Added to print queue: " << doc << endl;  
 }  
   
 void processDocuments() {  
 cout << "\nProcessing documents..." << endl;  
 while (!documents.empty()) {  
 string doc = documents.front();  
 cout << "Printing: " << doc << endl;  
   
 // Simulate printing time  
 this\_thread::sleep\_for(chrono::milliseconds(500));  
   
 documents.pop();  
 cout << "Finished printing: " << doc << endl;  
 }  
 }  
   
 int queueSize() const {  
 return documents.size();  
 }  
};  
  
// Example 2: BFS algorithm using queue for level-order traversal  
struct TreeNode {  
 int value;  
 TreeNode\* left;  
 TreeNode\* right;  
   
 TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}  
};  
  
void levelOrderTraversal(TreeNode\* root) {  
 if (!root) return;  
   
 queue<TreeNode\*> q;  
 q.push(root);  
   
 cout << "Level order traversal: ";  
 while (!q.empty()) {  
 TreeNode\* node = q.front();  
 q.pop();  
   
 cout << node->value << " ";  
   
 if (node->left) {  
 q.push(node->left);  
 }  
   
 if (node->right) {  
 q.push(node->right);  
 }  
 }  
 cout << endl;  
}  
  
int main() {  
 // Example 1: Print queue  
 PrintQueue printer;  
 printer.addDocument("report.pdf");  
 printer.addDocument("letter.doc");  
 printer.addDocument("image.jpg");  
   
 cout << "Queue size: " << printer.queueSize() << endl;  
 printer.processDocuments();  
   
 // Example 2: Level-order traversal of a binary tree using a queue  
 TreeNode\* root = new TreeNode(1);  
 root->left = new TreeNode(2);  
 root->right = new TreeNode(3);  
 root->left->left = new TreeNode(4);  
 root->left->right = new TreeNode(5);  
 root->right->left = new TreeNode(6);  
 root->right->right = new TreeNode(7);  
   
 levelOrderTraversal(root);  
   
 // Clean up memory  
 delete root->left->left;  
 delete root->left->right;  
 delete root->right->left;  
 delete root->right->right;  
 delete root->left;  
 delete root->right;  
 delete root;  
   
 return 0;  
}

### When to Use Queue

* When you need FIFO (First-In, First-Out) behavior
* For level-order traversals in trees/graphs (BFS - Breadth-First Search)
* For implementing buffers (e.g., print queue, task queue)
* For scheduling processes or tasks
* When order of processing is important

## 8.2.6 Priority Queue

std::priority\_queue is a container adapter that provides constant time lookup of the largest (by default) element. It’s implemented using a heap structure, which allows for efficient insertion and extraction of the highest priority element.

### Key Features

* **Priority-based**: Elements are retrieved according to priority, not insertion order
* **Container Adapter**: Built on top of other containers (typically vector)
* **Heap Structure**: Implemented as a max-heap by default (highest value has highest priority)
* **Fast Access to Highest Priority**: Constant time access to the top element
* **No iterators**: Can’t traverse all elements

### Basic Priority Queue Operations

#include <iostream>  
#include <queue>  
#include <string>  
using namespace std;  
  
int main() {  
 // Default priority queue (max-heap)  
 priority\_queue<int> pq;  
   
 // Insert elements  
 pq.push(10);  
 pq.push(30);  
 pq.push(20);  
 pq.push(5);  
   
 cout << "Priority queue size: " << pq.size() << endl;  
 cout << "Top element: " << pq.top() << endl; // Highest value is on top  
   
 // Process elements (highest priority first)  
 cout << "Elements in priority order: ";  
 while (!pq.empty()) {  
 cout << pq.top() << " "; // Access highest priority element  
 pq.pop(); // Remove highest priority element  
 }  
 cout << endl;  
   
 // Min-heap using custom comparator  
 priority\_queue<int, vector<int>, greater<int>> minPQ;  
   
 minPQ.push(10);  
 minPQ.push(30);  
 minPQ.push(20);  
 minPQ.push(5);  
   
 cout << "\nMin priority queue top: " << minPQ.top() << endl; // Lowest value on top  
   
 cout << "Elements in ascending order: ";  
 while (!minPQ.empty()) {  
 cout << minPQ.top() << " ";  
 minPQ.pop();  
 }  
 cout << endl;  
   
 return 0;  
}

### Custom Priority and Complex Types

#include <iostream>  
#include <queue>  
#include <string>  
#include <functional> // For std::greater  
using namespace std;  
  
// Define a custom type  
struct Task {  
 string name;  
 int priority; // Higher number = higher priority  
   
 Task(const string& n, int p) : name(n), priority(p) {}  
   
 // Operator overloading method  
 bool operator<(const Task& other) const {  
 return priority < other.priority; // For max-heap (highest priority on top)  
 }  
};  
  
// Alternative approach: Custom comparator class  
struct TaskComparator {  
 bool operator()(const Task& a, const Task& b) const {  
 return a.priority < b.priority; // For max-heap  
 }  
};  
  
int main() {  
 // Method 1: Using operator overloading  
 priority\_queue<Task> taskQueue;  
   
 // Add tasks  
 taskQueue.push(Task("Read emails", 10));  
 taskQueue.push(Task("Write report", 50));  
 taskQueue.push(Task("Meet client", 100));  
 taskQueue.push(Task("Coffee break", 5));  
   
 cout << "Processing tasks (using operator overloading):" << endl;  
 while (!taskQueue.empty()) {  
 Task currentTask = taskQueue.top();  
 cout << "Processing: " << currentTask.name << " (Priority: " << currentTask.priority << ")" << endl;  
 taskQueue.pop();  
 }  
   
 // Method 2: Using custom comparator  
 priority\_queue<Task, vector<Task>, TaskComparator> taskQueue2;  
   
 // Add tasks  
 taskQueue2.push(Task("Read emails", 10));  
 taskQueue2.push(Task("Write report", 50));  
 taskQueue2.push(Task("Meet client", 100));  
 taskQueue2.push(Task("Coffee break", 5));  
   
 cout << "\nProcessing tasks (using custom comparator):" << endl;  
 while (!taskQueue2.empty()) {  
 Task currentTask = taskQueue2.top();  
 cout << "Processing: " << currentTask.name << " (Priority: " << currentTask.priority << ")" << endl;  
 taskQueue2.pop();  
 }  
   
 // Method 3: Lambda function as comparator (C++11)  
 auto cmp = [](const Task& a, const Task& b) {   
 return a.priority < b.priority; // For max-heap  
 };  
 priority\_queue<Task, vector<Task>, decltype(cmp)> taskQueue3(cmp);  
   
 // Add tasks  
 taskQueue3.push(Task("Read emails", 10));  
 taskQueue3.push(Task("Write report", 50));  
 taskQueue3.push(Task("Meet client", 100));  
 taskQueue3.push(Task("Coffee break", 5));  
   
 cout << "\nProcessing tasks (using lambda comparator):" << endl;  
 while (!taskQueue3.empty()) {  
 Task currentTask = taskQueue3.top();  
 cout << "Processing: " << currentTask.name << " (Priority: " << currentTask.priority << ")" << endl;  
 taskQueue3.pop();  
 }  
   
 return 0;  
}

### Real-World Priority Queue Applications

#include <iostream>  
#include <queue>  
#include <vector>  
#include <string>  
#include <ctime>  
using namespace std;  
  
// Example 1: Emergency Room Triage System  
class Patient {  
private:  
 string name;  
 int severity; // 1-10, 10 being most severe  
 time\_t arrivalTime;  
   
public:  
 Patient(const string& n, int sev)   
 : name(n), severity(sev), arrivalTime(time(nullptr)) {}  
   
 string getName() const { return name; }  
 int getSeverity() const { return severity; }  
 time\_t getArrivalTime() const { return arrivalTime; }  
   
 // For display  
 string getTimeString() const {  
 char buffer[26];  
 struct tm\* timeinfo = localtime(&arrivalTime);  
 strftime(buffer, 26, "%H:%M:%S", timeinfo);  
 return string(buffer);  
 }  
};  
  
// Comparator for patients - higher severity first, then earlier arrival  
struct PatientComparator {  
 bool operator()(const Patient& a, const Patient& b) const {  
 if (a.getSeverity() != b.getSeverity()) {  
 return a.getSeverity() < b.getSeverity(); // Higher severity first  
 }  
 return a.getArrivalTime() > b.getArrivalTime(); // Earlier arrival time for equal severity  
 }  
};  
  
// Example 2: Dijkstra's Algorithm  
class Graph {  
private:  
 int V; // Number of vertices  
 vector<vector<pair<int, int>>> adj; // Adjacency list: (vertex, weight)  
   
public:  
 Graph(int vertices) : V(vertices) {  
 adj.resize(V);  
 }  
   
 void addEdge(int u, int v, int weight) {  
 adj[u].push\_back(make\_pair(v, weight));  
 adj[v].push\_back(make\_pair(u, weight)); // For undirected graph  
 }  
   
 // Dijkstra's algorithm using priority queue  
 void shortestPath(int src) {  
 // Min-heap priority queue  
 // Pair: (distance, vertex)  
 priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;  
   
 // Distance array  
 vector<int> dist(V, INT\_MAX);  
   
 // Insert source with distance 0  
 pq.push(make\_pair(0, src));  
 dist[src] = 0;  
   
 cout << "Dijkstra's Algorithm from vertex " << src << ":" << endl;  
   
 while (!pq.empty()) {  
 // Extract minimum distance vertex  
 int u = pq.top().second;  
 pq.pop();  
   
 // Visit all adjacent vertices  
 for (const auto& edge : adj[u]) {  
 int v = edge.first;  
 int weight = edge.second;  
   
 // If there's a shorter path to v through u  
 if (dist[v] > dist[u] + weight) {  
 dist[v] = dist[u] + weight;  
 pq.push(make\_pair(dist[v], v));  
 }  
 }  
 }  
   
 // Print shortest distances  
 cout << "Shortest distances from source:" << endl;  
 for (int i = 0; i < V; ++i) {  
 cout << "Vertex " << i << ": " << dist[i] << endl;  
 }  
 }  
};  
  
int main() {  
 // Example 1: Emergency Room Triage  
 priority\_queue<Patient, vector<Patient>, PatientComparator> emergencyRoom;  
   
 emergencyRoom.push(Patient("John Doe", 3));  
 this\_thread::sleep\_for(chrono::seconds(1));  
 emergencyRoom.push(Patient("Jane Smith", 8)); // Higher priority  
 this\_thread::sleep\_for(chrono::seconds(1));  
 emergencyRoom.push(Patient("Bob Johnson", 2));  
 this\_thread::sleep\_for(chrono::seconds(1));  
 emergencyRoom.push(Patient("Alice Brown", 8)); // Same priority as Jane, but later arrival  
 this\_thread::sleep\_for(chrono::seconds(1));  
 emergencyRoom.push(Patient("Charlie Davis", 10)); // Highest priority  
   
 cout << "Emergency Room - Treating patients in priority order:" << endl;  
 while (!emergencyRoom.empty()) {  
 const Patient& patient = emergencyRoom.top();  
 cout << "Treating: " << patient.getName()   
 << " (Severity: " << patient.getSeverity()   
 << ", Arrived: " << patient.getTimeString() << ")" << endl;  
 emergencyRoom.pop();  
 }  
   
 // Example 2: Dijkstra's Algorithm  
 cout << "\nDijkstra's Algorithm Example:" << endl;  
 Graph g(6); // Graph with 6 vertices  
   
 // Adding edges (u, v, weight)  
 g.addEdge(0, 1, 4);  
 g.addEdge(0, 2, 3);  
 g.addEdge(1, 2, 1);  
 g.addEdge(1, 3, 2);  
 g.addEdge(2, 3, 4);  
 g.addEdge(3, 4, 2);  
 g.addEdge(4, 5, 6);  
   
 g.shortestPath(0); // Find shortest paths from vertex 0  
   
 return 0;  
}

### When to Use Priority Queue

* When you need to process elements based on priority rather than arrival order
* For algorithms like Dijkstra’s shortest path, Prim’s MST, Huffman coding, etc.
* For scheduling systems (e.g., process schedulers, task management)
* For simulations where events need to be processed in order of importance
* For implementing a dynamic median finder

## 8.2.7 Set & Multiset

std::set and std::multiset are associative containers that store elements in a sorted order. The main difference is that set contains only unique elements, while multiset allows duplicates.

### Key Features

#### Common Features

* **Ordered**: Elements are always sorted according to a comparison function
* **Tree-Based**: Typically implemented as a Red-Black Tree (a type of self-balancing binary search tree)
* **No Direct Element Modification**: Elements are const once inserted (to maintain order)
* **Logarithmic Operations**: Most operations are O(log n)

#### Set-Specific Features

* **Unique Elements**: No duplicates allowed
* **Key is Value**: The element value itself is the key

#### Multiset-Specific Features

* **Allows Duplicates**: The same value can appear multiple times
* **Consistent Ordering**: Equal elements appear in the order they were inserted

### Basic Set Operations

#include <iostream>  
#include <set>  
#include <string>  
using namespace std;  
  
int main() {  
 // Create a set of integers  
 set<int> numbers;  
   
 // Insert elements  
 numbers.insert(30);  
 numbers.insert(10);  
 numbers.insert(50);  
 numbers.insert(20);  
 numbers.insert(10); // Duplicate, will be ignored  
   
 // Display set size  
 cout << "Set size: " << numbers.size() << endl;  
   
 // Check if an element exists  
 if (numbers.find(20) != numbers.end()) {  
 cout << "20 is in the set" << endl;  
 }  
   
 if (numbers.count(25) > 0) { // Another way to check  
 cout << "25 is in the set" << endl;  
 } else {  
 cout << "25 is not in the set" << endl;  
 }  
   
 // Iterate through the set (will be in sorted order)  
 cout << "Set elements: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Remove elements  
 numbers.erase(20);  
   
 // Check if removal was successful  
 if (numbers.find(20) == numbers.end()) {  
 cout << "20 is no longer in the set" << endl;  
 }  
   
 // Different ways to insert  
 auto result = numbers.insert(40);  
 if (result.second) {  
 cout << "40 was inserted successfully" << endl;  
 }  
   
 numbers.insert({5, 15, 25}); // Insert multiple elements (C++11)  
   
 // Set operations with iterators  
 auto it = numbers.begin();  
 advance(it, 2); // Move iterator forward by 2 positions  
   
 cout << "Third element in sorted set: " << \*it << endl;  
   
 cout << "Final set: ";  
 for (auto it = numbers.begin(); it != numbers.end(); ++it) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### Multiset Operations

#include <iostream>  
#include <set>  
#include <string>  
using namespace std;  
  
int main() {  
 // Create a multiset of integers  
 multiset<int> numbers;  
   
 // Insert elements  
 numbers.insert(30);  
 numbers.insert(10);  
 numbers.insert(50);  
 numbers.insert(20);  
 numbers.insert(10); // Duplicate allowed in multiset  
 numbers.insert(10); // Another duplicate  
   
 // Display multiset size  
 cout << "Multiset size: " << numbers.size() << endl;  
   
 // Count occurrences of an element  
 cout << "Count of 10: " << numbers.count(10) << endl;  
   
 // Iterate through the multiset (will be in sorted order)  
 cout << "Multiset elements: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Find elements  
 auto range = numbers.equal\_range(10); // Get range of elements equal to 10  
 cout << "Elements equal to 10: ";  
 for (auto it = range.first; it != range.second; ++it) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 // Erase elements  
 int eraseCount = numbers.erase(10); // Erase all elements with value 10  
 cout << "Erased " << eraseCount << " occurrences of 10" << endl;  
   
 // Erase a single element  
 auto it = numbers.find(30);  
 if (it != numbers.end()) {  
 numbers.erase(it); // Erase just one occurrence  
 }  
   
 cout << "Multiset after erasing: ";  
 for (const auto& num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### Custom Comparators

#include <iostream>  
#include <set>  
#include <string>  
using namespace std;  
  
// Custom comparison function for strings (case insensitive)  
struct CaseInsensitiveCompare {  
 bool operator()(const string& a, const string& b) const {  
 // Convert to lowercase for comparison  
 string aLower = a;  
 string bLower = b;  
 transform(aLower.begin(), aLower.end(), aLower.begin(), ::tolower);  
 transform(bLower.begin(), bLower.end(), bLower.begin(), ::tolower);  
 return aLower < bLower;  
 }  
};  
  
// Custom struct with custom comparison  
struct Person {  
 string name;  
 int age;  
   
 Person(const string& n, int a) : name(n), age(a) {}  
   
 // For displaying  
 friend ostream& operator<<(ostream& os, const Person& p) {  
 return os << p.name << " (" << p.age << ")";  
 }  
};  
  
// Compare persons by age  
struct CompareByAge {  
 bool operator()(const Person& a, const Person& b) const {  
 return a.age < b.age;  
 }  
};  
  
int main() {  
 // Set with custom comparator  
 set<string, CaseInsensitiveCompare> names;  
   
 names.insert("Apple");  
 names.insert("banana");  
 names.insert("Cherry");  
 names.insert("apple"); // Will be considered duplicate of "Apple"  
   
 cout << "Names set size: " << names.size() << endl;  
 cout << "Names set elements: ";  
 for (const auto& name : names) {  
 cout << name << " ";  
 }  
 cout << endl;  
   
 // Set of custom objects with custom comparison  
 set<Person, CompareByAge> people;  
   
 people.insert(Person("Alice", 30));  
 people.insert(Person("Bob", 25));  
 people.insert(Person("Charlie", 40));  
 people.insert(Person("David", 25)); // Same age as Bob, both will be included  
   
 cout << "\nPeople sorted by age: " << endl;  
 for (const auto& person : people) {  
 cout << " " << person << endl;  
 }  
   
 // Using a lambda as comparator (C++14 and later)  
 auto nameLengthComp = [](const string& a, const string& b) {  
 return a.length() < b.length() || (a.length() == b.length() && a < b);  
 };  
   
 set<string, decltype(nameLengthComp)> namesByLength(nameLengthComp);  
   
 namesByLength.insert("Apple");  
 namesByLength.insert("Banana");  
 namesByLength.insert("Cat");  
 namesByLength.insert("Dog");  
 namesByLength.insert("Elephant");  
   
 cout << "\nNames sorted by length: ";  
 for (const auto& name : namesByLength) {  
 cout << name << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### Set Operations (Union, Intersection, Difference)

#include <iostream>  
#include <set>  
#include <algorithm> // For set\_intersection, set\_union, set\_difference  
#include <vector>  
using namespace std;  
  
// Helper function to print any container  
template <typename Container>  
void printContainer(const Container& c, const string& name) {  
 cout << name << ": ";  
 for (const auto& item : c) {  
 cout << item << " ";  
 }  
 cout << endl;  
}  
  
int main() {  
 // Create sets  
 set<int> set1 = {1, 2, 3, 5, 7, 9};  
 set<int> set2 = {2, 4, 5, 6, 8};  
   
 printContainer(set1, "Set 1");  
 printContainer(set2, "Set 2");  
   
 // Union: Elements in either set1 or set2  
 vector<int> unionSet;  
 set\_union(set1.begin(), set1.end(),   
 set2.begin(), set2.end(),  
 back\_inserter(unionSet));  
   
 printContainer(unionSet, "Union");  
   
 // Intersection: Elements in both set1 and set2  
 vector<int> intersectionSet;  
 set\_intersection(set1.begin(), set1.end(),  
 set2.begin(), set2.end(),  
 back\_inserter(intersectionSet));  
   
 printContainer(intersectionSet, "Intersection");  
   
 // Difference: Elements in set1 but not in set2  
 vector<int> differenceSet;  
 set\_difference(set1.begin(), set1.end(),  
 set2.begin(), set2.end(),  
 back\_inserter(differenceSet));  
   
 printContainer(differenceSet, "Set1 - Set2");  
   
 // Symmetric difference: Elements in either set but not in both  
 vector<int> symmetricDiff;  
 set\_symmetric\_difference(set1.begin(), set1.end(),  
 set2.begin(), set2.end(),  
 back\_inserter(symmetricDiff));  
   
 printContainer(symmetricDiff, "Symmetric Difference");  
   
 // You can also do these operations manually  
 set<int> manualUnion;  
 manualUnion.insert(set1.begin(), set1.end());  
 manualUnion.insert(set2.begin(), set2.end());  
   
 printContainer(manualUnion, "Manual Union");  
   
 return 0;  
}

### Real-World Applications of Set and Multiset

#include <iostream>  
#include <set>  
#include <string>  
#include <vector>  
using namespace std;  
  
// Example 1: Word uniqueness checker  
void uniqueWordsChecker(const string& text) {  
 // Split text into words  
 vector<string> words;  
 string word;  
 for (char c : text) {  
 if (isalpha(c)) {  
 word += tolower(c);  
 } else if (!word.empty()) {  
 words.push\_back(word);  
 word.clear();  
 }  
 }  
 if (!word.empty()) {  
 words.push\_back(word);  
 }  
   
 // Use a set to find unique words  
 set<string> uniqueWords;  
 multiset<string> allWords;  
   
 for (const auto& w : words) {  
 uniqueWords.insert(w);  
 allWords.insert(w);  
 }  
   
 cout << "Text analysis:" << endl;  
 cout << "Total word count: " << words.size() << endl;  
 cout << "Unique word count: " << uniqueWords.size() << endl;  
   
 // Find duplicate words  
 cout << "\nWords that appear more than once:" << endl;  
 for (const auto& w : uniqueWords) {  
 int count = allWords.count(w);  
 if (count > 1) {  
 cout << " \"" << w << "\" appears " << count << " times" << endl;  
 }  
 }  
}  
  
// Example 2: Room scheduling system  
class TimeSlot {  
private:  
 int startHour; // 0-23  
 int endHour; // 0-23  
   
public:  
 TimeSlot(int start, int end) : startHour(start), endHour(end) {}  
   
 int getStart() const { return startHour; }  
 int getEnd() const { return endHour; }  
   
 bool operator<(const TimeSlot& other) const {  
 return startHour < other.startHour;  
 }  
   
 friend ostream& operator<<(ostream& os, const TimeSlot& ts) {  
 return os << ts.startHour << ":00-" << ts.endHour << ":00";  
 }  
};  
  
class RoomScheduler {  
private:  
 set<TimeSlot> bookedTimeSlots;  
   
public:  
 bool bookRoom(int startHour, int endHour) {  
 if (startHour >= endHour || startHour < 0 || endHour > 24) {  
 cout << "Invalid time slot" << endl;  
 return false;  
 }  
   
 TimeSlot newSlot(startHour, endHour);  
   
 // Check for conflicts  
 for (const auto& slot : bookedTimeSlots) {  
 if (!(endHour <= slot.getStart() || startHour >= slot.getEnd())) {  
 cout << "Conflict with existing booking: " << slot << endl;  
 return false;  
 }  
 }  
   
 // No conflicts, book the room  
 bookedTimeSlots.insert(newSlot);  
 cout << "Room booked for " << newSlot << endl;  
 return true;  
 }  
   
 void displaySchedule() {  
 if (bookedTimeSlots.empty()) {  
 cout << "No bookings yet" << endl;  
 return;  
 }  
   
 cout << "Current room schedule:" << endl;  
 for (const auto& slot : bookedTimeSlots) {  
 cout << " " << slot << endl;  
 }  
 }  
   
 vector<TimeSlot> getAvailableSlots(int minDuration) {  
 vector<TimeSlot> availableSlots;  
   
 // Start with full day availability  
 vector<TimeSlot> freeTime;  
   
 if (bookedTimeSlots.empty()) {  
 freeTime.push\_back(TimeSlot(0, 24));  
 } else {  
 // Add start of day to first booking  
 auto firstBooking = bookedTimeSlots.begin();  
 if (firstBooking->getStart() > 0) {  
 freeTime.push\_back(TimeSlot(0, firstBooking->getStart()));  
 }  
   
 // Add gaps between bookings  
 auto it = bookedTimeSlots.begin();  
 auto nextIt = it;  
 ++nextIt;  
   
 while (nextIt != bookedTimeSlots.end()) {  
 if (nextIt->getStart() > it->getEnd()) {  
 freeTime.push\_back(TimeSlot(it->getEnd(), nextIt->getStart()));  
 }  
 it = nextIt;  
 ++nextIt;  
 }  
   
 // Add end of last booking to end of day  
 auto lastBooking = --bookedTimeSlots.end();  
 if (lastBooking->getEnd() < 24) {  
 freeTime.push\_back(TimeSlot(lastBooking->getEnd(), 24));  
 }  
 }  
   
 // Filter by minimum duration  
 for (const auto& slot : freeTime) {  
 if (slot.getEnd() - slot.getStart() >= minDuration) {  
 availableSlots.push\_back(slot);  
 }  
 }  
   
 return availableSlots;  
 }  
};  
  
int main() {  
 // Example 1: Word uniqueness checker  
 string text = "The quick brown fox jumps over the lazy dog. The dog was not amused.";  
 uniqueWordsChecker(text);  
   
 // Example 2: Room scheduling system  
 cout << "\nRoom Scheduling Example:" << endl;  
 RoomScheduler scheduler;  
   
 scheduler.bookRoom(9, 10); // 9:00-10:00  
 scheduler.bookRoom(11, 13); // 11:00-13:00  
 scheduler.bookRoom(14, 16); // 14:00-16:00  
   
 // Try to book a conflicting slot  
 scheduler.bookRoom(12, 14); // Conflicts with 11:00-13:00  
   
 // Display schedule  
 scheduler.displaySchedule();  
   
 // Get available slots with minimum duration 2 hours  
 cout << "\nAvailable slots (2+ hours duration):" << endl;  
 auto availableSlots = scheduler.getAvailableSlots(2);  
 for (const auto& slot : availableSlots) {  
 cout << " " << slot << endl;  
 }  
   
 return 0;  
}

### When to Use Set and Multiset

**Use Set When:** - You need to maintain a collection of unique elements in sorted order - You frequently check for element existence - You need to iterate through elements in sorted order - Set operations like union, intersection, or difference are needed

**Use Multiset When:** - You need to maintain a sorted collection that allows duplicates - You need to count occurrences of elements - You want to maintain insertion order for equal elements

## 8.2.8 Map & Multimap

std::map and std::multimap are associative containers that store key-value pairs in a sorted order according to the key. Like with set/multiset, the main difference is that map contains unique keys, while multimap allows duplicate keys.

### Key Features

#### Common Features

* **Key-Value Storage**: Each element is a pair of key and value
* **Ordered by Key**: Elements are sorted by key using a comparison function
* **Tree-Based**: Typically implemented as a Red-Black Tree
* **Logarithmic Operations**: Most operations are O(log n)

#### Map-Specific Features

* **Unique Keys**: Each key can only appear once
* **Direct Value Access**: Values can be accessed using operator[] with the key

#### Multimap-Specific Features

* **Allows Duplicate Keys**: Multiple values can be associated with the same key
* **No operator[]**: Since keys aren’t unique, direct access via operator[] isn’t provided

### Basic Map Operations

#include <iostream>  
#include <map>  
#include <string>  
using namespace std;  
  
int main() {  
 // Create a map (string keys, int values)  
 map<string, int> ages;  
   
 // Insert elements (multiple ways)  
 ages["Alice"] = 30; // Using operator[]  
 ages.insert(pair<string, int>("Bob", 25)); // Using insert() with pair  
 ages.insert(make\_pair("Charlie", 35)); // Using make\_pair  
 ages.insert({"David", 28}); // Using initializer list (C++11)  
   
 // Display map size  
 cout << "Map size: " << ages.size() << endl;  
   
 // Access elements  
 cout << "Alice's age: " << ages["Alice"] << endl;  
 cout << "Bob's age: " << ages.at("Bob") << endl; // at() throws exception if key not found  
   
 // Checking if a key exists  
 if (ages.find("Eve") != ages.end()) {  
 cout << "Eve's age: " << ages["Eve"] << endl;  
 } else {  
 cout << "Eve is not in the map" << endl;  
 }  
   
 // Using operator[] with a new key inserts a default-constructed value  
 cout << "Eve's age (default): " << ages["Eve"] << endl; // Creates entry with value 0  
 cout << "Map size after accessing Eve: " << ages.size() << endl;  
   
 // Iterate through the map (will be in sorted order by key)  
 cout << "\nAll entries:" << endl;  
 for (const auto& entry : ages) {  
 cout << entry.first << ": " << entry.second << endl;  
 }  
   
 // Update a value  
 ages["Alice"] = 31;  
   
 // Another way to insert/update  
 ages.insert\_or\_assign("Frank", 45); // C++17  
   
 // Erase an entry  
 ages.erase("Eve");  
   
 // Check if removal was successful  
 if (ages.find("Eve") == ages.end()) {  
 cout << "\nEve was removed from the map" << endl;  
 }  
   
 // Another way to iterate  
 cout << "\nAll entries after modifications:" << endl;  
 for (auto it = ages.begin(); it != ages.end(); ++it) {  
 cout << it->first << ": " << it->second << endl;  
 }  
   
 return 0;  
}

### Multimap Operations

#include <iostream>  
#include <map>  
#include <string>  
using namespace std;  
  
int main() {  
 // Create a multimap (string keys, int values)  
 multimap<string, int> scores;  
   
 // Insert elements  
 scores.insert(make\_pair("Alice", 85));  
 scores.insert(make\_pair("Bob", 92));  
 scores.insert(make\_pair("Charlie", 78));  
 scores.insert(make\_pair("Alice", 90)); // Duplicate key  
 scores.insert(make\_pair("Alice", 88)); // Another duplicate  
   
 // Display multimap size  
 cout << "Multimap size: " << scores.size() << endl;  
   
 // Count entries with a specific key  
 cout << "Alice's entry count: " << scores.count("Alice") << endl;  
   
 // Find entries with a specific key  
 cout << "\nAlice's scores:" << endl;  
 auto range = scores.equal\_range("Alice");  
 for (auto it = range.first; it != range.second; ++it) {  
 cout << it->first << ": " << it->second << endl;  
 }  
   
 // Iterate through all entries  
 cout << "\nAll scores:" << endl;  
 for (const auto& entry : scores) {  
 cout << entry.first << ": " << entry.second << endl;  
 }  
   
 // Erase entries with a specific key  
 int removed = scores.erase("Alice"); // Removes all entries with key "Alice"  
 cout << "\nRemoved " << removed << " entries for Alice" << endl;  
   
 // Erase a single entry  
 auto it = scores.find("Bob");  
 if (it != scores.end()) {  
 scores.erase(it); // Erase just this entry  
 cout << "Removed one entry for Bob" << endl;  
 }  
   
 // Check remaining entries  
 cout << "\nRemaining entries:" << endl;  
 for (const auto& entry : scores) {  
 cout << entry.first << ": " << entry.second << endl;  
 }  
   
 return 0;  
}

### Maps with Custom Keys

#include <iostream>  
#include <map>  
#include <string>  
using namespace std;  
  
// Custom key type  
struct Person {  
 string firstName;  
 string lastName;  
   
 Person(const string& first, const string& last)   
 : firstName(first), lastName(last) {}  
   
 // Required for map: operator< for comparison  
 bool operator<(const Person& other) const {  
 // Sort by last name, then first name  
 if (lastName != other.lastName) {  
 return lastName < other.lastName;  
 }  
 return firstName < other.firstName;  
 }  
   
 // For display  
 friend ostream& operator<<(ostream& os, const Person& p) {  
 return os << p.firstName << " " << p.lastName;  
 }  
};  
  
// Custom comparator approach  
struct CaseInsensitiveCompare {  
 bool operator()(const string& a, const string& b) const {  
 string aLower = a;  
 string bLower = b;  
 transform(aLower.begin(), aLower.end(), aLower.begin(), ::tolower);  
 transform(bLower.begin(), bLower.end(), bLower.begin(), ::tolower);  
 return aLower < bLower;  
 }  
};  
  
int main() {  
 // Map with custom key type  
 map<Person, string> phoneBook;  
   
 // Insert entries  
 phoneBook[Person("John", "Doe")] = "555-1234";  
 phoneBook[Person("Jane", "Doe")] = "555-2345";  
 phoneBook[Person("John", "Smith")] = "555-3456";  
   
 // Display entries (sorted by last name, then first name)  
 cout << "Phone book entries:" << endl;  
 for (const auto& entry : phoneBook) {  
 cout << entry.first << ": " << entry.second << endl;  
 }  
   
 // Map with custom comparator  
 map<string, int, CaseInsensitiveCompare> caseInsensitiveMap;  
   
 caseInsensitiveMap["Apple"] = 1;  
 caseInsensitiveMap["banana"] = 2;  
 caseInsensitiveMap["Cherry"] = 3;  
   
 // "apple" is treated as the same key as "Apple" due to case-insensitive comparison  
 caseInsensitiveMap["apple"] = 4; // This will update the value for "Apple"  
   
 cout << "\nCase-insensitive map:" << endl;  
 for (const auto& entry : caseInsensitiveMap) {  
 cout << entry.first << ": " << entry.second << endl;  
 }  
   
 return 0;  
}

### Real-World Applications of Map and Multimap

#include <iostream>  
#include <map>  
#include <string>  
#include <vector>  
#include <fstream>  
#include <sstream>  
using namespace std;  
  
// Example 1: Word frequency counter  
map<string, int> countWordFrequency(const string& text) {  
 map<string, int> frequency;  
   
 istringstream iss(text);  
 string word;  
   
 while (iss >> word) {  
 // Remove punctuation  
 word.erase(remove\_if(word.begin(), word.end(),   
 [](char c) { return !isalpha(c); }),  
 word.end());  
   
 // Convert to lowercase  
 transform(word.begin(), word.end(), word.begin(), ::tolower);  
   
 // Count only non-empty words  
 if (!word.empty()) {  
 frequency[word]++;  
 }  
 }  
   
 return frequency;  
}  
  
// Example 2: Student grade tracker  
class GradeTracker {  
private:  
 // Map: student name -> map of subject -> grades  
 map<string, map<string, vector<int>>> studentGrades;  
   
public:  
 // Add a grade for a student in a subject  
 void addGrade(const string& student, const string& subject, int grade) {  
 studentGrades[student][subject].push\_back(grade);  
 }  
   
 // Get average grade for a student in a subject  
 double getAverage(const string& student, const string& subject) {  
 if (studentGrades.find(student) == studentGrades.end() ||  
 studentGrades[student].find(subject) == studentGrades[student].end() ||  
 studentGrades[student][subject].empty()) {  
 return 0.0;  
 }  
   
 const vector<int>& grades = studentGrades[student][subject];  
 int sum = 0;  
 for (int grade : grades) {  
 sum += grade;  
 }  
   
 return static\_cast<double>(sum) / grades.size();  
 }  
   
 // Get overall average for a student  
 double getOverallAverage(const string& student) {  
 if (studentGrades.find(student) == studentGrades.end()) {  
 return 0.0;  
 }  
   
 int totalSum = 0;  
 int totalCount = 0;  
   
 for (const auto& subjectPair : studentGrades[student]) {  
 for (int grade : subjectPair.second) {  
 totalSum += grade;  
 totalCount++;  
 }  
 }  
   
 return totalCount > 0 ? static\_cast<double>(totalSum) / totalCount : 0.0;  
 }  
   
 // Display all grades  
 void displayAllGrades() {  
 for (const auto& studentPair : studentGrades) {  
 cout << "Student: " << studentPair.first << endl;  
   
 for (const auto& subjectPair : studentPair.second) {  
 cout << " Subject: " << subjectPair.first << ", Grades: ";  
   
 for (int grade : subjectPair.second) {  
 cout << grade << " ";  
 }  
   
 cout << "(Avg: " << getAverage(studentPair.first, subjectPair.first) << ")" << endl;  
 }  
   
 cout << " Overall Average: " << getOverallAverage(studentPair.first) << endl;  
 }  
 }  
};  
  
// Example 3: Movie database with multimap  
class MovieDatabase {  
private:  
 // Multimap: genre -> movie title  
 multimap<string, string> moviesByGenre;  
   
 // Map: movie title -> year  
 map<string, int> movieYears;  
   
public:  
 // Add a movie to the database  
 void addMovie(const string& title, int year, const vector<string>& genres) {  
 movieYears[title] = year;  
   
 for (const string& genre : genres) {  
 moviesByGenre.insert({genre, title});  
 }  
 }  
   
 // Find movies by genre  
 vector<pair<string, int>> findMoviesByGenre(const string& genre) {  
 vector<pair<string, int>> result;  
   
 auto range = moviesByGenre.equal\_range(genre);  
 for (auto it = range.first; it != range.second; ++it) {  
 const string& title = it->second;  
 int year = movieYears[title];  
 result.push\_back({title, year});  
 }  
   
 return result;  
 }  
   
 // Display all genres and their movies  
 void displayByGenre() {  
 string currentGenre;  
   
 for (const auto& entry : moviesByGenre) {  
 if (entry.first != currentGenre) {  
 currentGenre = entry.first;  
 cout << "\nGenre: " << currentGenre << endl;  
 }  
   
 cout << " " << entry.second << " (" << movieYears[entry.second] << ")" << endl;  
 }  
 }  
   
 // Count movies per genre  
 map<string, int> countByGenre() {  
 map<string, int> counts;  
   
 for (const auto& entry : moviesByGenre) {  
 counts[entry.first]++;  
 }  
   
 return counts;  
 }  
};  
  
int main() {  
 // Example 1: Word frequency counter  
 string text = "The quick brown fox jumps over the lazy dog. The dog was not amused. Quick thinking saved the day.";  
   
 map<string, int> wordFreq = countWordFrequency(text);  
   
 cout << "Word Frequency:" << endl;  
 for (const auto& entry : wordFreq) {  
 cout << entry.first << ": " << entry.second << endl;  
 }  
   
 // Example 2: Student grade tracker  
 cout << "\nGrade Tracker Example:" << endl;  
 GradeTracker tracker;  
   
 tracker.addGrade("Alice", "Math", 95);  
 tracker.addGrade("Alice", "Math", 88);  
 tracker.addGrade("Alice", "English", 92);  
 tracker.addGrade("Bob", "Math", 78);  
 tracker.addGrade("Bob", "Physics", 85);  
 tracker.addGrade("Bob", "Physics", 90);  
   
 tracker.displayAllGrades();  
   
 // Example 3: Movie database  
 cout << "\nMovie Database Example:" << endl;  
 MovieDatabase movieDB;  
   
 movieDB.addMovie("The Shawshank Redemption", 1994, {"Drama"});  
 movieDB.addMovie("The Godfather", 1972, {"Crime", "Drama"});  
 movieDB.addMovie("Pulp Fiction", 1994, {"Crime", "Drama"});  
 movieDB.addMovie("The Dark Knight", 2008, {"Action", "Crime", "Drama"});  
 movieDB.addMovie("Inception", 2010, {"Action", "Sci-Fi", "Thriller"});  
   
 cout << "Movies by genre:" << endl;  
 movieDB.displayByGenre();  
   
 cout << "\nMovies in Drama genre:" << endl;  
 auto dramaMovies = movieDB.findMoviesByGenre("Drama");  
 for (const auto& movie : dramaMovies) {  
 cout << " " << movie.first << " (" << movie.second << ")" << endl;  
 }  
   
 cout << "\nMovie counts by genre:" << endl;  
 auto genreCounts = movieDB.countByGenre();  
 for (const auto& entry : genreCounts) {  
 cout << " " << entry.first << ": " << entry.second << " movies" << endl;  
 }  
   
 return 0;  
}

### When to Use Map and Multimap

**Use Map When:** - You need to associate values with unique keys - You need fast key-based lookups - You need to maintain a sorted collection of key-value pairs - You need to update values associated with keys

**Use Multimap When:** - You need to associate multiple values with the same key - You need to maintain key-value pairs in sorted order - You need to find all values associated with a specific key - You’re implementing a dictionary with multiple definitions per word

### Performance Considerations

1. **Memory Overhead**: Maps and sets use more memory than unordered versions due to tree structure
2. **Insertion/Deletion/Lookup**: O(log n) complexity for map, set, multimap, multiset
3. **Iteration**: Very efficient, proportional to the number of elements

### Choosing Between Container Types

| Need | Recommended Container |
| --- | --- |
| Fast lookup by key with unique keys | unordered\_map (if order doesn’t matter), map (if order matters) |
| Fast lookup by key with duplicate keys | unordered\_multimap (if order doesn’t matter), multimap (if order matters) |
| Collection of unique items | unordered\_set (if order doesn’t matter), set (if order matters) |
| Collection with duplicates | unordered\_multiset (if order doesn’t matter), multiset (if order matters) |
| Fast insertions/deletions at both ends | deque |
| Fast random access and dynamic size | vector |
| Fast insertions/deletions anywhere | list |

# Chapter 8: Standard Template Library (STL) (Part 3)

## 8.3 Iterators

Iterators are one of the most important concepts in the STL. They act as the “glue” between containers and algorithms, providing a uniform way to access elements in containers regardless of their internal structure.

### What Are Iterators?

Iterators are objects that behave like pointers, providing a way to traverse through the elements in a container. They abstract the specifics of how a container stores its elements, allowing algorithms to work with any container that provides the required iterators.

### Iterator Categories

STL defines several categories of iterators, each with different capabilities:

1. **Input Iterators**: Read-only, single-pass traversal (e.g., istream\_iterator)
2. **Output Iterators**: Write-only, single-pass traversal (e.g., ostream\_iterator)
3. **Forward Iterators**: Read/write, single-direction traversal, multi-pass (e.g., forward\_list iterators)
4. **Bidirectional Iterators**: Read/write, bidirectional traversal (e.g., list, set iterators)
5. **Random Access Iterators**: Read/write, arbitrary access (e.g., vector, deque iterators)
6. **Contiguous Iterators** (C++17): Random access with contiguous memory guarantee (e.g., vector iterators)

Each category is a superset of the capabilities of the previous categories.

Here’s a table showing which containers provide which types of iterators:

| Container | Iterator Type |
| --- | --- |
| vector | Random Access (Contiguous) |
| array | Random Access (Contiguous) |
| deque | Random Access |
| list | Bidirectional |
| forward\_list | Forward |
| set/multiset | Bidirectional |
| map/multimap | Bidirectional |
| unordered\_\* | Forward |

### Basic Iterator Operations

All iterators support these basic operations:

* Dereferencing: \*iter (access element)
* Increment: ++iter (move to next element)
* Copy/assignment: iter1 = iter2
* Comparison: iter1 == iter2, iter1 != iter2

More advanced iterators support additional operations:

* Bidirectional: --iter (move to previous element)
* Random Access: iter + n, iter - n, iter[n], iter1 - iter2, iter1 < iter2

#include <iostream>  
#include <vector>  
#include <list>  
#include <set>  
using namespace std;  
  
int main() {  
 // Using iterators with vector (random access)  
 vector<int> nums = {10, 20, 30, 40, 50};  
   
 cout << "Vector using iterators:" << endl;  
 for (auto it = nums.begin(); it != nums.end(); ++it) {  
 cout << \*it << " "; // Dereferencing to get value  
 }  
 cout << endl;  
   
 // Random access operations (only for random access iterators)  
 auto it = nums.begin();  
 cout << "Vector - Third element using random access: " << \*(it + 2) << endl;  
 cout << "Vector - Second element using subscript: " << it[1] << endl;  
   
 // Using iterators with list (bidirectional)  
 list<int> myList = {5, 15, 25, 35, 45};  
   
 cout << "\nList using iterators:" << endl;  
 for (auto it = myList.begin(); it != myList.end(); ++it) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 // Bidirectional iteration - going backwards  
 cout << "List in reverse:" << endl;  
 auto listEnd = myList.end();  
 auto listIt = myList.end();  
 do {  
 --listIt;  
 cout << \*listIt << " ";  
 } while (listIt != myList.begin());  
 cout << endl;  
   
 // Set with bidirectional iterators  
 set<char> chars = {'a', 'b', 'c', 'd', 'e'};  
   
 cout << "\nSet using iterators:" << endl;  
 for (auto it = chars.begin(); it != chars.end(); ++it) {  
 cout << \*it << " ";  
 // \*it = 'z'; // Error - set iterators are const  
 }  
 cout << endl;  
   
 return 0;  
}

### Iterator Functions and Utilities

STL provides several utility functions for working with iterators:

#include <iostream>  
#include <vector>  
#include <iterator> // For iterator utilities  
using namespace std;  
  
int main() {  
 vector<int> nums = {10, 20, 30, 40, 50, 60, 70};  
   
 // advance() - moves iterator forward or backward  
 auto it1 = nums.begin();  
 advance(it1, 3); // Move forward 3 positions  
 cout << "After advance(3): " << \*it1 << endl; // 40  
   
 // next(), prev() - returns iterator to next/previous positions  
 auto it2 = next(it1, 2); // Iterator 2 positions after it1  
 cout << "next(it1, 2): " << \*it2 << endl; // 60  
   
 auto it3 = prev(it1, 1); // Iterator 1 position before it1  
 cout << "prev(it1, 1): " << \*it3 << endl; // 30  
   
 // distance() - returns distance between iterators  
 cout << "Distance from begin to it1: "   
 << distance(nums.begin(), it1) << endl; // 3  
   
 // Iterate with specific step  
 cout << "Iterating with step 2: ";  
 for (auto it = nums.begin(); it != nums.end(); advance(it, 2)) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### Special Iterator Types

STL provides several special iterator types for specific purposes:

#### Reverse Iterators

Reverse iterators traverse a container in reverse order. They’re created using rbegin() and rend() container methods:

#include <iostream>  
#include <vector>  
using namespace std;  
  
int main() {  
 vector<int> nums = {10, 20, 30, 40, 50};  
   
 cout << "Forward iteration: ";  
 for (auto it = nums.begin(); it != nums.end(); ++it) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 cout << "Reverse iteration with reverse\_iterator: ";  
 for (auto rit = nums.rbegin(); rit != nums.rend(); ++rit) {  
 cout << \*rit << " "; // 50 40 30 20 10  
 }  
 cout << endl;  
   
 // Converting between normal and reverse iterators  
 auto it = nums.begin() + 2; // Points to 30  
 vector<int>::reverse\_iterator rit(it);  
 cout << "it points to: " << \*it << endl; // 30  
 cout << "rit points to: " << \*rit << endl; // 20 (previous element)  
 cout << "rit.base() points to: " << \*rit.base() << endl; // 30  
   
 return 0;  
}

#### Const Iterators

Const iterators prevent modification of the elements they point to:

#include <iostream>  
#include <vector>  
using namespace std;  
  
int main() {  
 vector<int> nums = {10, 20, 30, 40, 50};  
   
 // Non-const iterator - can modify elements  
 auto it = nums.begin();  
 \*it = 100; // OK - modifies the first element  
   
 // Const iterator - can't modify elements  
 auto cit = nums.cbegin(); // or const\_iterator cit = nums.begin();  
 // \*cit = 200; // Error: assignment of read-only location  
   
 cout << "Vector after modification: ";  
 for (const auto& num : nums) {  
 cout << num << " "; // 100 20 30 40 50  
 }  
 cout << endl;  
   
 return 0;  
}

#### Insert Iterators

Insert iterators convert assignment operations into insertions:

#include <iostream>  
#include <vector>  
#include <list>  
#include <algorithm> // For copy  
#include <iterator> // For insert iterators  
using namespace std;  
  
int main() {  
 vector<int> source = {10, 20, 30, 40, 50};  
   
 // back\_inserter - inserts at the end using push\_back  
 vector<int> dest1;  
 copy(source.begin(), source.end(), back\_inserter(dest1));  
   
 // front\_inserter - inserts at the beginning using push\_front (not available for vector)  
 list<int> dest2;  
 copy(source.begin(), source.end(), front\_inserter(dest2));  
   
 // inserter - inserts at a specific position  
 vector<int> dest3 = {1, 2, 3};  
 copy(source.begin(), source.end(), inserter(dest3, dest3.begin() + 1));  
   
 // Display results  
 cout << "Source: ";  
 for (int n : source) cout << n << " ";  
 cout << endl;  
   
 cout << "dest1 (back\_inserter): ";  
 for (int n : dest1) cout << n << " ";  
 cout << endl;  
   
 cout << "dest2 (front\_inserter): ";  
 for (int n : dest2) cout << n << " ";  
 cout << endl;  
   
 cout << "dest3 (inserter): ";  
 for (int n : dest3) cout << n << " ";  
 cout << endl;  
   
 return 0;  
}

#### Stream Iterators

Stream iterators allow STL algorithms to work directly with input/output streams:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <iterator> // For stream iterators  
using namespace std;  
  
int main() {  
 // Writing to output stream  
 vector<int> nums = {10, 20, 30, 40, 50};  
   
 cout << "Vector contents: ";  
 copy(nums.begin(), nums.end(),   
 ostream\_iterator<int>(cout, " "));  
 cout << endl;  
   
 // Reading from input stream  
 cout << "Enter integers (Ctrl+Z or Ctrl+D to end): ";  
 vector<int> input;  
 copy(istream\_iterator<int>(cin), istream\_iterator<int>(),  
 back\_inserter(input));  
   
 cout << "You entered: ";  
 copy(input.begin(), input.end(),  
 ostream\_iterator<int>(cout, " "));  
 cout << endl;  
   
 // Using stream iterators for calculations  
 cout << "Sum: " << accumulate(input.begin(), input.end(), 0) << endl;  
   
 return 0;  
}

### Iterator Invalidation

Iterator invalidation occurs when a container operation makes existing iterators invalid. Understanding when this happens is crucial for safe code:

#include <iostream>  
#include <vector>  
#include <list>  
using namespace std;  
  
int main() {  
 // Vector iterator invalidation  
 vector<int> vec = {10, 20, 30, 40, 50};  
 auto vecIt = vec.begin() + 2; // Points to 30  
   
 cout << "Before insertion: \*vecIt = " << \*vecIt << endl;  
   
 // This insertion might invalidate all iterators if reallocation occurs  
 vec.insert(vec.begin(), 5);  
   
 // Using vecIt after insertion could cause undefined behavior  
 // cout << "After insertion: \*vecIt = " << \*vecIt << endl; // Dangerous!  
   
 // Instead, get a new iterator  
 vecIt = vec.begin() + 3; // Points to 30 again  
 cout << "After reacquiring iterator: \*vecIt = " << \*vecIt << endl;  
   
 // List iterators are more stable - only invalidated when their actual element is removed  
 list<int> lst = {10, 20, 30, 40, 50};  
 auto lstIt = next(lst.begin(), 2); // Points to 30  
   
 cout << "\nList before insertion: \*lstIt = " << \*lstIt << endl;  
   
 // This doesn't invalidate lstIt since the element it points to isn't affected  
 lst.insert(lst.begin(), 5);  
   
 cout << "List after insertion: \*lstIt = " << \*lstIt << endl; // Still valid!  
   
 // Removing the element pointed to by lstIt would invalidate it  
 auto toRemove = lstIt;  
 ++lstIt; // Move to next element before removing  
 lst.erase(toRemove);  
   
 cout << "List after erasing 30, lstIt now points to: " << \*lstIt << endl;  
   
 return 0;  
}

### Writing Iterator-Safe Code

To write iterator-safe code, follow these guidelines:

1. **Refresh iterators after container modifications**:

* auto it = container.begin();  
  container.insert(/\* ... \*/);  
  it = container.begin(); // Refresh iterator

1. **Store indices instead of iterators** when heavy modification is expected:

* size\_t index = 3;  
  container.insert(/\* ... \*/);  
  auto it = container.begin() + index; // Recalculate iterator

1. **Use returned iterators** from modifying operations:

* // erase() returns iterator to element after the erased one  
  it = container.erase(it); // No need to increment

1. **Be aware of container-specific behaviors**:
   * vector/deque: Insertions/removals invalidate iterators to elements after the modification point
   * list/forward\_list: Insertions/removals only invalidate iterators pointing to affected elements
   * Associative containers: Insertions/removals don’t invalidate iterators except to the erased elements

### Custom Iterator Implementation

Understanding how iterators work internally can be valuable. Here’s a simple example of implementing your own iterator:

#include <iostream>  
#include <iterator>  
using namespace std;  
  
// Simple container with custom iterator  
class IntArray {  
private:  
 int\* data;  
 size\_t size;  
   
public:  
 // Constructor  
 IntArray(size\_t n) : size(n) {  
 data = new int[size](); // Initialize with zeros  
 }  
   
 // Destructor  
 ~IntArray() {  
 delete[] data;  
 }  
   
 // Get value at index  
 int& at(size\_t index) {  
 if (index >= size) throw out\_of\_range("Index out of bounds");  
 return data[index];  
 }  
   
 // Get size  
 size\_t getSize() const {  
 return size;  
 }  
   
 // Custom iterator class  
 class Iterator {  
 private:  
 int\* ptr;  
   
 public:  
 // Iterator traits  
 using iterator\_category = std::random\_access\_iterator\_tag;  
 using value\_type = int;  
 using difference\_type = ptrdiff\_t;  
 using pointer = int\*;  
 using reference = int&;  
   
 // Constructor  
 Iterator(int\* p) : ptr(p) {}  
   
 // Dereference  
 int& operator\*() const {  
 return \*ptr;  
 }  
   
 // Increment (prefix)  
 Iterator& operator++() {  
 ++ptr;  
 return \*this;  
 }  
   
 // Increment (postfix)  
 Iterator operator++(int) {  
 Iterator temp = \*this;  
 ++(\*this);  
 return temp;  
 }  
   
 // Decrement (prefix)  
 Iterator& operator--() {  
 --ptr;  
 return \*this;  
 }  
   
 // Decrement (postfix)  
 Iterator operator--(int) {  
 Iterator temp = \*this;  
 --(\*this);  
 return temp;  
 }  
   
 // Addition  
 Iterator operator+(difference\_type n) const {  
 return Iterator(ptr + n);  
 }  
   
 // Subtraction  
 Iterator operator-(difference\_type n) const {  
 return Iterator(ptr - n);  
 }  
   
 // Distance between iterators  
 difference\_type operator-(const Iterator& other) const {  
 return ptr - other.ptr;  
 }  
   
 // Equality comparison  
 bool operator==(const Iterator& other) const {  
 return ptr == other.ptr;  
 }  
   
 // Inequality comparison  
 bool operator!=(const Iterator& other) const {  
 return ptr != other.ptr;  
 }  
 };  
   
 // Iterator factory methods  
 Iterator begin() {  
 return Iterator(data);  
 }  
   
 Iterator end() {  
 return Iterator(data + size);  
 }  
};  
  
int main() {  
 IntArray arr(5);  
   
 // Initialize with values  
 for (size\_t i = 0; i < arr.getSize(); ++i) {  
 arr.at(i) = i \* 10;  
 }  
   
 // Use our custom iterator  
 cout << "Array elements: ";  
 for (auto it = arr.begin(); it != arr.end(); ++it) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 // Test random access  
 auto it = arr.begin();  
 it = it + 3;  
 cout << "Element at position 3: " << \*it << endl;  
   
 // Test range-based for loop (requires begin/end)  
 cout << "Using range-based for loop: ";  
 for (int value : arr) {  
 cout << value << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### Best Practices for Using Iterators

1. **Prefer high-level iteration** when possible:

* // Modern, safer approach  
  for (const auto& item : container) { /\* ... \*/ }  
    
  // Traditional, more explicit approach  
  for (auto it = container.begin(); it != container.end(); ++it) { /\* ... \*/ }

1. **Use appropriate iterator types**:
   * Use const\_iterator/cbegin() when you don’t need to modify elements
   * Use reverse iterators (rbegin()/rend()) for backwards traversal
2. **Check validity before dereferencing**:

* auto it = container.find(value);  
  if (it != container.end()) {  
   // Only dereference valid iterators  
   process(\*it);  
  }

1. **Update iterators after container modifications**
2. **Understand the iterator requirements of algorithms**:
   * Some algorithms require only input/forward iterators
   * Others need bidirectional or random access
3. **Use iterator utilities** like advance(), next(), prev(), and distance()
4. **Be mindful of iterator invalidation rules** for each container

## 8.4 Algorithms (sort, find, count, etc.)

The STL algorithms library provides a large collection of functions that operate on ranges of elements defined by iterators. These algorithms are both efficient and generic, working with any container type that provides the required iterators.

### Algorithm Categories

STL algorithms can be broadly categorized as:

1. **Non-modifying sequence operations**: Don’t change the elements or their order
2. **Modifying sequence operations**: Change elements or their order
3. **Sorting and related operations**: Sort, merge, partition, etc.
4. **Binary search operations**: Search in sorted sequences
5. **Set operations**: Operations on sorted ranges
6. **Numeric operations**: Mathematical operations

### Non-Modifying Algorithms

These algorithms don’t modify the elements in the container:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <numeric>  
using namespace std;  
  
int main() {  
 vector<int> nums = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};  
   
 // find - returns iterator to first occurrence of value  
 auto it = find(nums.begin(), nums.end(), 5);  
 if (it != nums.end()) {  
 cout << "Found 5 at index: " << distance(nums.begin(), it) << endl;  
 }  
   
 // find\_if - finds element satisfying predicate  
 auto even = find\_if(nums.begin(), nums.end(),   
 [](int n){ return n % 2 == 0; });  
 if (even != nums.end()) {  
 cout << "First even number: " << \*even << endl;  
 }  
   
 // count - counts occurrences of value  
 int count\_5 = count(nums.begin(), nums.end(), 5);  
 cout << "Number 5 appears " << count\_5 << " times" << endl;  
   
 // count\_if - counts elements satisfying predicate  
 int num\_even = count\_if(nums.begin(), nums.end(),   
 [](int n){ return n % 2 == 0; });  
 cout << "Number of even values: " << num\_even << endl;  
   
 // all\_of, any\_of, none\_of - check conditions  
 bool all\_positive = all\_of(nums.begin(), nums.end(),   
 [](int n){ return n > 0; });  
 cout << "All positive: " << boolalpha << all\_positive << endl;  
   
 bool any\_greater\_than\_5 = any\_of(nums.begin(), nums.end(),  
 [](int n){ return n > 5; });  
 cout << "Any greater than 5: " << any\_greater\_than\_5 << endl;  
   
 bool none\_negative = none\_of(nums.begin(), nums.end(),   
 [](int n){ return n < 0; });  
 cout << "None negative: " << none\_negative << endl;  
   
 // for\_each - applies function to each element  
 cout << "Elements: ";  
 for\_each(nums.begin(), nums.end(),   
 [](int n){ cout << n << " "; });  
 cout << endl;  
   
 // min\_element, max\_element - finds min/max element  
 auto min\_it = min\_element(nums.begin(), nums.end());  
 auto max\_it = max\_element(nums.begin(), nums.end());  
 cout << "Min: " << \*min\_it << ", Max: " << \*max\_it << endl;  
   
 // minmax\_element - finds both min and max  
 auto [min\_it2, max\_it2] = minmax\_element(nums.begin(), nums.end());  
 cout << "Min: " << \*min\_it2 << ", Max: " << \*max\_it2 << endl;  
   
 // accumulate - computes sum or applies binary operation  
 int sum = accumulate(nums.begin(), nums.end(), 0);  
 cout << "Sum: " << sum << endl;  
   
 int product = accumulate(nums.begin(), nums.end(), 1,   
 [](int a, int b){ return a \* b; });  
 cout << "Product: " << product << endl;  
   
 // adjacent\_find - finds equal adjacent elements  
 auto adjacent = adjacent\_find(nums.begin(), nums.end());  
 if (adjacent != nums.end()) {  
 cout << "First adjacent equal elements: " << \*adjacent   
 << " and " << \*(adjacent + 1) << endl;  
 }  
   
 return 0;  
}

### Modifying Algorithms

These algorithms modify container elements or their order:

#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
template<typename T>  
void print\_vector(const vector<T>& v, const string& label) {  
 cout << label << ": ";  
 for (const auto& item : v) {  
 cout << item << " ";  
 }  
 cout << endl;  
}  
  
int main() {  
 vector<int> nums = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};  
 print\_vector(nums, "Original");  
   
 // copy - copies elements to another container  
 vector<int> copy\_dest(nums.size());  
 copy(nums.begin(), nums.end(), copy\_dest.begin());  
 print\_vector(copy\_dest, "Copy");  
   
 // copy\_if - copies elements satisfying predicate  
 vector<int> even\_numbers;  
 copy\_if(nums.begin(), nums.end(), back\_inserter(even\_numbers),  
 [](int n){ return n % 2 == 0; });  
 print\_vector(even\_numbers, "Even numbers");  
   
 // transform - applies function to each element  
 vector<int> transformed(nums.size());  
 transform(nums.begin(), nums.end(), transformed.begin(),  
 [](int n){ return n \* n; });  
 print\_vector(transformed, "Squared");  
   
 // transform with two input sequences  
 vector<int> vec1 = {1, 2, 3, 4, 5};  
 vector<int> vec2 = {10, 20, 30, 40, 50};  
 vector<int> sum\_vec(vec1.size());  
   
 transform(vec1.begin(), vec1.end(), vec2.begin(),   
 sum\_vec.begin(), plus<int>());  
 print\_vector(sum\_vec, "vec1 + vec2");  
   
 // fill - fills range with value  
 vector<int> filled(5);  
 fill(filled.begin(), filled.end(), 42);  
 print\_vector(filled, "Filled with 42");  
   
 // fill\_n - fills n elements with value  
 vector<int> partial\_fill(10);  
 fill\_n(partial\_fill.begin(), 5, 99);  
 print\_vector(partial\_fill, "First 5 elements filled with 99");  
   
 // generate - fills range with values from generator function  
 vector<int> generated(5);  
 int value = 1;  
 generate(generated.begin(), generated.end(),   
 [&value]() { return value \*= 2; });  
 print\_vector(generated, "Generated powers of 2");  
   
 // replace - replaces values  
 vector<int> replaced = nums;  
 replace(replaced.begin(), replaced.end(), 1, 99);  
 print\_vector(replaced, "Replaced 1 with 99");  
   
 // replace\_if - replaces values satisfying predicate  
 vector<int> replaced\_if = nums;  
 replace\_if(replaced\_if.begin(), replaced\_if.end(),  
 [](int n){ return n % 2 == 0; }, 0);  
 print\_vector(replaced\_if, "Replaced even numbers with 0");  
   
 // remove and erase-remove idiom  
 vector<int> to\_remove = nums;  
 print\_vector(to\_remove, "Before remove");  
   
 // Remove doesn't change vector size, it moves unwanted elements to end  
 auto new\_end = remove(to\_remove.begin(), to\_remove.end(), 1);  
 to\_remove.erase(new\_end, to\_remove.end()); // Actually remove  
 print\_vector(to\_remove, "After removing 1");  
   
 // remove\_if - removes elements satisfying predicate  
 vector<int> remove\_if\_vec = nums;  
 remove\_if\_vec.erase(  
 remove\_if(remove\_if\_vec.begin(), remove\_if\_vec.end(),  
 [](int n){ return n < 3; }),  
 remove\_if\_vec.end()  
 );  
 print\_vector(remove\_if\_vec, "After removing elements < 3");  
   
 // unique - removes consecutive duplicates  
 vector<int> with\_dups = {1, 1, 2, 2, 2, 3, 3, 1, 1, 4};  
 print\_vector(with\_dups, "With duplicates");  
   
 with\_dups.erase(unique(with\_dups.begin(), with\_dups.end()),  
 with\_dups.end());  
 print\_vector(with\_dups, "After removing consecutive duplicates");  
   
 // reverse - reverses element order  
 vector<int> to\_reverse = {1, 2, 3, 4, 5};  
 reverse(to\_reverse.begin(), to\_reverse.end());  
 print\_vector(to\_reverse, "Reversed");  
   
 // rotate - rotates elements  
 vector<int> to\_rotate = {1, 2, 3, 4, 5, 6, 7};  
 rotate(to\_rotate.begin(), to\_rotate.begin() + 3, to\_rotate.end());  
 print\_vector(to\_rotate, "Rotated (moved first 3 elements to end)");  
   
 // shuffle - randomly shuffles elements  
 vector<int> to\_shuffle = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
 random\_device rd;  
 mt19937 g(rd()); // Random generator  
 shuffle(to\_shuffle.begin(), to\_shuffle.end(), g);  
 print\_vector(to\_shuffle, "Shuffled");  
   
 // swap\_ranges - swaps elements between ranges  
 vector<int> v1 = {1, 2, 3, 4, 5};  
 vector<int> v2 = {6, 7, 8, 9, 10};  
 swap\_ranges(v1.begin(), v1.end(), v2.begin());  
 print\_vector(v1, "v1 after swap");  
 print\_vector(v2, "v2 after swap");  
   
 return 0;  
}

### Sorting and Related Algorithms

Sorting algorithms arrange elements in a specific order:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <functional> // For greater<>  
using namespace std;  
  
template<typename T>  
void print\_vector(const vector<T>& v, const string& label) {  
 cout << label << ": ";  
 for (const auto& item : v) {  
 cout << item << " ";  
 }  
 cout << endl;  
}  
  
int main() {  
 vector<int> nums = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};  
 print\_vector(nums, "Original");  
   
 // sort - sorts elements (quicksort-like algorithm)  
 vector<int> sorted = nums;  
 sort(sorted.begin(), sorted.end());  
 print\_vector(sorted, "Sorted ascending");  
   
 // sort in descending order  
 vector<int> sorted\_desc = nums;  
 sort(sorted\_desc.begin(), sorted\_desc.end(), greater<int>());  
 print\_vector(sorted\_desc, "Sorted descending");  
   
 // Custom sort comparator  
 vector<int> custom\_sort = nums;  
 sort(custom\_sort.begin(), custom\_sort.end(),  
 [](int a, int b) { return abs(a - 5) < abs(b - 5); });  
 print\_vector(custom\_sort, "Sorted by distance from 5");  
   
 // stable\_sort - preserves relative order of equal elements  
 vector<pair<int, char>> pairs = {  
 {2, 'a'}, {1, 'b'}, {2, 'c'}, {1, 'd'}, {3, 'e'}  
 };  
   
 stable\_sort(pairs.begin(), pairs.end(),  
 [](auto& p1, auto& p2) { return p1.first < p2.first; });  
   
 cout << "Stable sorted pairs: ";  
 for (const auto& p : pairs) {  
 cout << "(" << p.first << "," << p.second << ") ";  
 }  
 cout << endl;  
   
 // partial\_sort - sorts part of range  
 vector<int> partial = nums;  
 partial\_sort(partial.begin(), partial.begin() + 4, partial.end());  
 print\_vector(partial, "First 4 elements sorted");  
   
 // nth\_element - partially sorts so that nth element is in correct position  
 vector<int> nth = nums;  
 auto middle = nth.begin() + nth.size() / 2;  
 nth\_element(nth.begin(), middle, nth.end());  
 print\_vector(nth, "After nth\_element (median)");  
 cout << "Median: " << \*middle << endl;  
   
 // partition - separates elements that satisfy predicate  
 vector<int> to\_partition = nums;  
 auto partition\_point = partition(to\_partition.begin(), to\_partition.end(),  
 [](int n) { return n % 2 == 0; });  
   
 cout << "Even elements: ";  
 for (auto it = to\_partition.begin(); it != partition\_point; ++it) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 cout << "Odd elements: ";  
 for (auto it = partition\_point; it != to\_partition.end(); ++it) {  
 cout << \*it << " ";  
 }  
 cout << endl;  
   
 // is\_sorted - checks if range is sorted  
 bool sorted\_check = is\_sorted(sorted.begin(), sorted.end());  
 cout << "Is 'sorted' sorted? " << boolalpha << sorted\_check << endl;  
   
 // merge - merges two sorted ranges  
 vector<int> v1 = {1, 3, 5, 7, 9};  
 vector<int> v2 = {2, 4, 6, 8, 10};  
 vector<int> merged(v1.size() + v2.size());  
   
 merge(v1.begin(), v1.end(), v2.begin(), v2.end(), merged.begin());  
 print\_vector(merged, "Merged sorted ranges");  
   
 // inplace\_merge - merges two consecutive sorted ranges in-place  
 vector<int> to\_inplace\_merge = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};  
 auto middle\_point = to\_inplace\_merge.begin() + 5;  
 inplace\_merge(to\_inplace\_merge.begin(), middle\_point, to\_inplace\_merge.end());  
 print\_vector(to\_inplace\_merge, "After inplace\_merge");  
   
 return 0;  
}

### Binary Search Algorithms

These algorithms work efficiently on sorted ranges:

#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
int main() {  
 // Create a sorted vector  
 vector<int> sorted = {10, 20, 30, 30, 30, 40, 50, 60, 70};  
   
 // binary\_search - checks if element exists (returns bool)  
 bool has30 = binary\_search(sorted.begin(), sorted.end(), 30);  
 bool has35 = binary\_search(sorted.begin(), sorted.end(), 35);  
   
 cout << "Contains 30: " << boolalpha << has30 << endl;  
 cout << "Contains 35: " << has35 << endl;  
   
 // lower\_bound - returns iterator to first element >= value  
 auto lb = lower\_bound(sorted.begin(), sorted.end(), 30);  
 cout << "Lower bound of 30 is at index: "   
 << distance(sorted.begin(), lb) << endl;  
   
 // upper\_bound - returns iterator to first element > value  
 auto ub = upper\_bound(sorted.begin(), sorted.end(), 30);  
 cout << "Upper bound of 30 is at index: "   
 << distance(sorted.begin(), ub) << endl;  
   
 // equal\_range - returns pair of iterators defining the range of equal elements  
 auto [first, last] = equal\_range(sorted.begin(), sorted.end(), 30);  
 cout << "Range of 30s is from index "   
 << distance(sorted.begin(), first) << " to "   
 << distance(sorted.begin(), last) << endl;  
   
 cout << "Count of 30s: " << distance(first, last) << endl;  
   
 // Binary search with custom comparison  
 vector<pair<string, int>> scores = {  
 {"Alice", 95}, {"Bob", 87}, {"Charlie", 95}, {"David", 80}  
 };  
   
 // Sort by score descending, then by name ascending  
 sort(scores.begin(), scores.end(),   
 [](const auto& a, const auto& b) {  
 if (a.second != b.second) {  
 return a.second > b.second; // Higher score first  
 }  
 return a.first < b.first; // Same score, alphabetical by name  
 });  
   
 cout << "\nScores (sorted):" << endl;  
 for (const auto& [name, score] : scores) {  
 cout << name << ": " << score << endl;  
 }  
   
 // Search for score >= 90  
 auto it = lower\_bound(scores.begin(), scores.end(), 90,  
 [](const auto& a, int value) {  
 return a.second > value; // Reversed comparison  
 });  
   
 cout << "\nStudents with score >= 90:" << endl;  
 while (it != scores.end() && it->second >= 90) {  
 cout << it->first << ": " << it->second << endl;  
 ++it;  
 }  
   
 return 0;  
}

### Set Operations on Sorted Ranges

These algorithms perform set operations on sorted ranges:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <iterator>  
using namespace std;  
  
template<typename Container>  
void print\_container(const Container& c, const string& label) {  
 cout << label << ": ";  
 for (const auto& item : c) {  
 cout << item << " ";  
 }  
 cout << endl;  
}  
  
int main() {  
 // Two sorted sets  
 vector<int> set1 = {1, 2, 3, 4, 5, 5, 6};  
 vector<int> set2 = {4, 5, 5, 6, 7, 8};  
   
 print\_container(set1, "Set 1");  
 print\_container(set2, "Set 2");  
   
 // includes - checks if one range includes another  
 bool set2\_in\_set1 = includes(set1.begin(), set1.end(),   
 set2.begin(), set2.end());  
 bool fourToSix\_in\_set1 = includes(set1.begin(), set1.end(),   
 set2.begin(), set2.begin() + 3);  
   
 cout << "Set 2 is contained in Set 1: " << boolalpha << set2\_in\_set1 << endl;  
 cout << "Range [4,5,5] is contained in Set 1: " << fourToSix\_in\_set1 << endl;  
   
 // set\_union - elements in either set  
 vector<int> set\_union;  
 set\_union(set1.begin(), set1.end(),   
 set2.begin(), set2.end(),  
 back\_inserter(set\_union));  
 print\_container(set\_union, "Union");  
   
 // set\_intersection - elements in both sets  
 vector<int> set\_intersection;  
 set\_intersection(set1.begin(), set1.end(),  
 set2.begin(), set2.end(),  
 back\_inserter(set\_intersection));  
 print\_container(set\_intersection, "Intersection");  
   
 // set\_difference - elements in first set but not in second  
 vector<int> set\_difference1;  
 set\_difference(set1.begin(), set1.end(),  
 set2.begin(), set2.end(),  
 back\_inserter(set\_difference1));  
 print\_container(set\_difference1, "Set1 - Set2");  
   
 vector<int> set\_difference2;  
 set\_difference(set2.begin(), set2.end(),  
 set1.begin(), set1.end(),  
 back\_inserter(set\_difference2));  
 print\_container(set\_difference2, "Set2 - Set1");  
   
 // set\_symmetric\_difference - elements in either set but not in both  
 vector<int> sym\_diff;  
 set\_symmetric\_difference(set1.begin(), set1.end(),  
 set2.begin(), set2.end(),  
 back\_inserter(sym\_diff));  
 print\_container(sym\_diff, "Symmetric Difference");  
   
 return 0;  
}

### Numeric Algorithms

Numeric algorithms are specialized for mathematical operations:

#include <iostream>  
#include <vector>  
#include <numeric> // For numeric algorithms  
#include <algorithm>  
using namespace std;  
  
int main() {  
 vector<int> nums = {1, 2, 3, 4, 5};  
   
 // accumulate - computes sum or applies binary operation  
 int sum = accumulate(nums.begin(), nums.end(), 0);  
 cout << "Sum: " << sum << endl;  
   
 // Accumulate with custom binary operation  
 int product = accumulate(nums.begin(), nums.end(), 1,   
 multiplies<int>());  
 cout << "Product: " << product << endl;  
   
 // inner\_product - computes dot product or custom combination  
 vector<int> weights = {2, 1, 3, 2, 1};  
 int dot\_product = inner\_product(nums.begin(), nums.end(),  
 weights.begin(), 0);  
 cout << "Weighted sum: " << dot\_product << endl;  
   
 // Custom inner product (sum of absolute differences)  
 int abs\_diff\_sum = inner\_product(nums.begin(), nums.end(),  
 weights.begin(), 0,  
 plus<int>(),  
 [](int a, int b) { return abs(a - b); });  
 cout << "Sum of absolute differences: " << abs\_diff\_sum << endl;  
   
 // adjacent\_difference - computes difference between adjacent elements  
 vector<int> diffs(nums.size());  
 adjacent\_difference(nums.begin(), nums.end(), diffs.begin());  
   
 cout << "Adjacent differences: ";  
 for (int diff : diffs) {  
 cout << diff << " ";  
 }  
 cout << endl;  
   
 // Custom adjacent difference (sum of adjacent elements)  
 vector<int> adjacent\_sums(nums.size());  
 adjacent\_difference(nums.begin(), nums.end(), adjacent\_sums.begin(),  
 plus<int>());  
   
 cout << "Adjacent sums: ";  
 for (int sum : adjacent\_sums) {  
 cout << sum << " ";  
 }  
 cout << endl;  
   
 // partial\_sum - computes running sum  
 vector<int> running\_sum(nums.size());  
 partial\_sum(nums.begin(), nums.end(), running\_sum.begin());  
   
 cout << "Running sum: ";  
 for (int sum : running\_sum) {  
 cout << sum << " ";  
 }  
 cout << endl;  
   
 // Custom partial sum (running product)  
 vector<int> running\_product(nums.size());  
 partial\_sum(nums.begin(), nums.end(), running\_product.begin(),  
 multiplies<int>());  
   
 cout << "Running product: ";  
 for (int prod : running\_product) {  
 cout << prod << " ";  
 }  
 cout << endl;  
   
 // iota - fills range with sequential values  
 vector<int> seq(10);  
 iota(seq.begin(), seq.end(), 1); // Start from 1  
   
 cout << "Sequence: ";  
 for (int n : seq) {  
 cout << n << " ";  
 }  
 cout << endl;  
   
 // reduce - parallel version of accumulate (C++17)  
 // int parallel\_sum = reduce(execution::par, nums.begin(), nums.end(), 0);  
 // cout << "Parallel sum: " << parallel\_sum << endl;  
   
 return 0;  
}

### Algorithm Complexity and Performance

Understanding algorithm time complexity helps choose the right one for your task:

| Algorithm | Time Complexity | Space Complexity | Notes |
| --- | --- | --- | --- |
| find, count | O(n) | O(1) | Linear search |
| binary\_search | O(log n) | O(1) | Requires sorted input |
| sort | O(n log n) | O(log n) | Typically QuickSort/IntroSort |
| stable\_sort | O(n log n) | O(n) | Typically MergeSort |
| partial\_sort | O(n log k) | O(k) | k = position of element |
| nth\_element | O(n) | O(1) | Linear average time |
| min\_element/max\_element | O(n) | O(1) | Single pass |
| accumulate | O(n) | O(1) | Linear traversal |

### Parallel Algorithms (C++17)

C++17 introduced parallel versions of many algorithms:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <chrono>  
#include <execution> // For parallel execution policies  
using namespace std;  
using namespace chrono;  
  
int main() {  
 // Create a large vector for testing  
 vector<int> large\_vec(10'000'000);  
 iota(large\_vec.begin(), large\_vec.end(), 0); // Fill with 0...n  
 shuffle(large\_vec.begin(), large\_vec.end(), mt19937{random\_device{}()});  
   
 cout << "Testing sort performance with 10 million elements:" << endl;  
   
 // Sequential sort  
 auto seq\_vec = large\_vec;  
 auto start = high\_resolution\_clock::now();  
 sort(execution::seq, seq\_vec.begin(), seq\_vec.end());  
 auto end = high\_resolution\_clock::now();  
 auto seq\_time = duration\_cast<milliseconds>(end - start).count();  
 cout << "Sequential sort: " << seq\_time << " ms" << endl;  
   
 // Parallel sort  
 auto par\_vec = large\_vec;  
 start = high\_resolution\_clock::now();  
 sort(execution::par, par\_vec.begin(), par\_vec.end());  
 end = high\_resolution\_clock::now();  
 auto par\_time = duration\_cast<milliseconds>(end - start).count();  
 cout << "Parallel sort: " << par\_time << " ms" << endl;  
   
 // Parallel unsequenced sort  
 auto par\_unseq\_vec = large\_vec;  
 start = high\_resolution\_clock::now();  
 sort(execution::par\_unseq, par\_unseq\_vec.begin(), par\_unseq\_vec.end());  
 end = high\_resolution\_clock::now();  
 auto par\_unseq\_time = duration\_cast<milliseconds>(end - start).count();  
 cout << "Parallel unsequenced sort: " << par\_unseq\_time << " ms" << endl;  
   
 // Verify all results are the same  
 bool same\_results = equal(seq\_vec.begin(), seq\_vec.end(),   
 par\_vec.begin()) &&  
 equal(seq\_vec.begin(), seq\_vec.end(),   
 par\_unseq\_vec.begin());  
 cout << "All sorted results match: " << boolalpha << same\_results << endl;  
   
 return 0;  
}

### Real-World Algorithm Examples

#### Example 1: Finding Most Frequent Words

#include <iostream>  
#include <vector>  
#include <string>  
#include <algorithm>  
#include <map>  
#include <cctype> // for isalpha  
using namespace std;  
  
// Function to count word frequencies and return top k most frequent  
vector<pair<string, int>> top\_k\_frequent\_words(const string& text, int k) {  
 // Tokenize text into words  
 vector<string> words;  
 string word;  
 for (char c : text) {  
 if (isalpha(c)) {  
 word += tolower(c);  
 } else if (!word.empty()) {  
 words.push\_back(word);  
 word.clear();  
 }  
 }  
 if (!word.empty()) {  
 words.push\_back(word);  
 }  
   
 // Count frequencies  
 map<string, int> freq;  
 for (const string& w : words) {  
 freq[w]++;  
 }  
   
 // Convert to vector of pairs for sorting  
 vector<pair<string, int>> word\_counts(freq.begin(), freq.end());  
   
 // Sort by frequency (descending) and alphabetically for ties  
 sort(word\_counts.begin(), word\_counts.end(),  
 [](const auto& a, const auto& b) {  
 return a.second > b.second ||   
 (a.second == b.second && a.first < b.first);  
 });  
   
 // Return top k  
 if (word\_counts.size() > k) {  
 word\_counts.resize(k);  
 }  
   
 return word\_counts;  
}  
  
int main() {  
 string text = "To be, or not to be, that is the question: "  
 "Whether 'tis nobler in the mind to suffer "  
 "The slings and arrows of outrageous fortune, "  
 "Or to take Arms against a Sea of troubles, "  
 "And by opposing end them.";  
   
 int k = 5;  
 auto result = top\_k\_frequent\_words(text, k);  
   
 cout << "Top " << k << " most frequent words:" << endl;  
 for (const auto& [word, count] : result) {  
 cout << word << ": " << count << endl;  
 }  
   
 return 0;  
}

#### Example 2: Custom Sorting Algorithm

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <string>  
using namespace std;  
  
struct Student {  
 string name;  
 double gpa;  
 int age;  
   
 // For display  
 friend ostream& operator<<(ostream& os, const Student& s) {  
 return os << s.name << " (GPA: " << s.gpa << ", Age: " << s.age << ")";  
 }  
};  
  
int main() {  
 vector<Student> students = {  
 {"Alice", 3.9, 22},  
 {"Bob", 3.7, 21},  
 {"Charlie", 3.9, 23},  
 {"David", 3.5, 22},  
 {"Eve", 3.7, 20},  
 };  
   
 // Display original order  
 cout << "Original students:" << endl;  
 for (const auto& student : students) {  
 cout << student << endl;  
 }  
   
 // Sort by GPA (descending), then by age (ascending) for ties  
 sort(students.begin(), students.end(),  
 [](const Student& a, const Student& b) {  
 if (a.gpa != b.gpa) {  
 return a.gpa > b.gpa; // Higher GPA first  
 }  
 return a.age < b.age; // Same GPA, younger first  
 });  
   
 // Display sorted order  
 cout << "\nStudents sorted by GPA (desc) and age (asc):" << endl;  
 for (const auto& student : students) {  
 cout << student << endl;  
 }  
   
 // Find students with GPA >= 3.8  
 auto high\_gpa = partition\_point(students.begin(), students.end(),  
 [](const Student& s) { return s.gpa >= 3.8; });  
   
 cout << "\nStudents with GPA >= 3.8:" << endl;  
 for (auto it = students.begin(); it != high\_gpa; ++it) {  
 cout << \*it << endl;  
 }  
   
 // Find student with exact name  
 auto it = find\_if(students.begin(), students.end(),  
 [](const Student& s) { return s.name == "Bob"; });  
   
 if (it != students.end()) {  
 cout << "\nFound student: " << \*it << endl;  
 }  
   
 return 0;  
}

#### Example 3: Algorithm Composition

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <numeric>  
#include <cmath>  
using namespace std;  
  
// Function to calculate mean of a vector  
double mean(const vector<double>& v) {  
 return accumulate(v.begin(), v.end(), 0.0) / v.size();  
}  
  
// Function to calculate standard deviation  
double standard\_deviation(const vector<double>& v) {  
 double m = mean(v);  
 vector<double> diff(v.size());  
   
 // Calculate differences from mean  
 transform(v.begin(), v.end(), diff.begin(),  
 [m](double x) { return x - m; });  
   
 // Square the differences  
 transform(diff.begin(), diff.end(), diff.begin(),  
 [](double x) { return x \* x; });  
   
 // Calculate mean of squares  
 double sq\_sum = accumulate(diff.begin(), diff.end(), 0.0);  
   
 return sqrt(sq\_sum / v.size());  
}  
  
// Function to normalize data (z-score)  
vector<double> normalize(const vector<double>& v) {  
 double m = mean(v);  
 double sd = standard\_deviation(v);  
   
 vector<double> result(v.size());  
 transform(v.begin(), v.end(), result.begin(),  
 [m, sd](double x) { return (x - m) / sd; });  
   
 return result;  
}  
  
int main() {  
 vector<double> data = {12.5, 19.8, 16.7, 18.2, 15.9, 21.3, 17.5};  
   
 cout << "Original data:" << endl;  
 for (double val : data) {  
 cout << val << " ";  
 }  
 cout << endl;  
   
 // Calculate statistics  
 double m = mean(data);  
 double sd = standard\_deviation(data);  
   
 cout << "Mean: " << m << endl;  
 cout << "Standard Deviation: " << sd << endl;  
   
 // Normalize data  
 auto normalized = normalize(data);  
   
 cout << "\nNormalized data (z-scores):" << endl;  
 for (double val : normalized) {  
 cout << val << " ";  
 }  
 cout << endl;  
   
 // Verify normalization worked correctly  
 cout << "Mean of normalized data: " << mean(normalized) << endl;  
 cout << "SD of normalized data: " << standard\_deviation(normalized) << endl;  
   
 return 0;  
}

### Best Practices for STL Algorithms

1. **Choose the Right Algorithm**:
   * Understand what each algorithm does and its requirements
   * Consider time and space complexity
   * Check if input must be sorted
2. **Use Algorithm Compositions**:
   * Chain algorithms together for complex tasks
   * Use the output of one algorithm as input to another
3. **Make Predicate Functions Clear and Focused**:
   * Keep lambda functions short and readable
   * Use named function objects for complex predicates
4. **Consider Using Execution Policies** (C++17 and later):
   * std::execution::seq - sequential execution
   * std::execution::par - parallel execution
   * std::execution::par\_unseq - parallel and vectorized execution
5. **Be Careful with Mutating Algorithms**:
   * Some algorithms modify containers in-place
   * Others require destination iterators
6. **Understand Iterator Requirements**:
   * Some algorithms need random access iterators
   * Others work with any iterator category
7. **Use Algorithm Adapters When Appropriate**:
   * Use iterator adapters like back\_inserter for dynamic growth
   * Consider views (C++20) for non-copying transformations
8. **Avoid Reinventing the Wheel**:
   * There’s probably an STL algorithm for your task
   * Custom loops are often less efficient and more error-prone
9. **Watch for Edge Cases**:
   * Handle empty ranges properly
   * Consider what happens with elements that compare equal
10. **Leverage Range-Based Algorithms** (C++20):
    * Modern C++ provides more intuitive range-based versions of algorithms
    * Consider using ranges when available

# Chapter 8: Standard Template Library (STL) (Part 4)

## 8.5 Function Objects (Functors)

Function objects, often called functors, are objects that can be called like a function. They are implemented as classes that overload the function call operator operator().

### 8.5.1 Basic Concept of Functors

Functors are essentially objects that behave like functions. They’re created by defining classes that implement the function call operator ().

#include <iostream>  
using namespace std;  
  
// Simple functor that adds a specified value  
class Adder {  
private:  
 int addValue;  
   
public:  
 // Constructor that initializes the value to add  
 Adder(int val) : addValue(val) {}  
   
 // Function call operator  
 int operator()(int x) const {  
 return x + addValue;  
 }  
};  
  
int main() {  
 // Create functor objects  
 Adder add5(5);  
 Adder add10(10);  
   
 // Use the functors like functions  
 cout << "add5(10) = " << add5(10) << endl; // Outputs: 15  
 cout << "add10(10) = " << add10(10) << endl; // Outputs: 20  
   
 // Can be used on multiple values  
 cout << "add5(20) = " << add5(20) << endl; // Outputs: 25  
 cout << "add5(25) = " << add5(25) << endl; // Outputs: 30  
   
 return 0;  
}

### 8.5.2 Advantages of Functors

Functors have several advantages over regular functions:

1. **State Preservation**: They can maintain state between calls
2. **Type Safety**: They are objects with defined types
3. **Inline Optimization**: Compilers can often inline functor calls
4. **Parameterization**: They can be customized at instantiation
5. **STL Integration**: They integrate seamlessly with STL algorithms

#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
// Counter functor that tracks how many times it's called  
class Counter {  
private:  
 int count = 0;  
 int threshold;  
   
public:  
 Counter(int t = 0) : threshold(t) {}  
   
 // Function call operator that counts calls and checks threshold  
 bool operator()(int x) {  
 ++count;  
 return x > threshold;  
 }  
   
 // Method to get current count  
 int getCount() const {  
 return count;  
 }  
};  
  
int main() {  
 vector<int> nums = {5, 2, 8, 1, 9, 3, 7, 4, 6};  
   
 // Count how many elements are greater than 5  
 Counter counter(5);  
 int greaterThan5 = count\_if(nums.begin(), nums.end(), counter);  
   
 cout << "Numbers greater than 5: " << greaterThan5 << endl;  
 cout << "Function was called " << counter.getCount() << " times" << endl;  
   
 return 0;  
}

### 8.5.3 Standard Library Functors

The C++ Standard Library provides many built-in functors in the <functional> header:

#### Arithmetic Functors

#include <iostream>  
#include <functional>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
int main() {  
 // Using arithmetic functors  
 plus<int> add;  
 minus<int> subtract;  
 multiplies<int> multiply;  
 divides<double> divide;  
 modulus<int> mod;  
 negate<int> negate;  
   
 cout << "plus: 5 + 3 = " << add(5, 3) << endl;  
 cout << "minus: 5 - 3 = " << subtract(5, 3) << endl;  
 cout << "multiplies: 5 \* 3 = " << multiply(5, 3) << endl;  
 cout << "divides: 5 / 3 = " << divide(5, 3) << endl;  
 cout << "modulus: 5 % 3 = " << mod(5, 3) << endl;  
 cout << "negate: -5 = " << negate(5) << endl;  
   
 // Using functors with algorithms  
 vector<int> nums = {1, 2, 3, 4, 5};  
 vector<int> result(nums.size());  
   
 // Transform each element by multiplying it by 2  
 transform(nums.begin(), nums.end(), result.begin(),   
 bind(multiply, placeholders::\_1, 2));  
   
 cout << "After multiplying each element by 2: ";  
 for (int n : result) {  
 cout << n << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

#### Comparison Functors

#include <iostream>  
#include <functional>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
int main() {  
 // Using comparison functors  
 equal\_to<int> eq;  
 not\_equal\_to<int> neq;  
 greater<int> gt;  
 less<int> lt;  
 greater\_equal<int> ge;  
 less\_equal<int> le;  
   
 cout << "equal\_to: 5 == 3 is " << boolalpha << eq(5, 3) << endl;  
 cout << "not\_equal\_to: 5 != 3 is " << neq(5, 3) << endl;  
 cout << "greater: 5 > 3 is " << gt(5, 3) << endl;  
 cout << "less: 5 < 3 is " << lt(5, 3) << endl;  
 cout << "greater\_equal: 5 >= 5 is " << ge(5, 5) << endl;  
 cout << "less\_equal: 3 <= 5 is " << le(3, 5) << endl;  
   
 // Sorting with comparison functors  
 vector<int> nums = {5, 2, 8, 1, 9, 3, 7, 4, 6};  
   
 // Sort in descending order  
 sort(nums.begin(), nums.end(), greater<int>());  
   
 cout << "Sorted in descending order: ";  
 for (int n : nums) {  
 cout << n << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

#### Logical Functors

#include <iostream>  
#include <functional>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
int main() {  
 // Using logical functors  
 logical\_and<bool> land;  
 logical\_or<bool> lor;  
 logical\_not<bool> lnot;  
   
 cout << "logical\_and: true && false is " << boolalpha << land(true, false) << endl;  
 cout << "logical\_or: true || false is " << lor(true, false) << endl;  
 cout << "logical\_not: !true is " << lnot(true) << endl;  
   
 // Using with algorithms  
 vector<bool> b1 = {true, false, true};  
 vector<bool> b2 = {false, true, true};  
 vector<bool> result(b1.size());  
   
 // Element-wise AND operation  
 transform(b1.begin(), b1.end(), b2.begin(), result.begin(), logical\_and<bool>());  
   
 cout << "Result of element-wise AND: ";  
 for (bool b : result) {  
 cout << b << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### 8.5.4 Functors with STL Algorithms

Functors integrate seamlessly with STL algorithms, making code more readable and flexible:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <functional>  
using namespace std;  
  
// Custom functor that checks if a number is divisible by another  
class DivisibleBy {  
private:  
 int divisor;  
   
public:  
 DivisibleBy(int d) : divisor(d) {}  
   
 bool operator()(int value) const {  
 return value % divisor == 0;  
 }  
};  
  
int main() {  
 vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};  
   
 // Count numbers divisible by 3  
 int count3 = count\_if(nums.begin(), nums.end(), DivisibleBy(3));  
 cout << "Numbers divisible by 3: " << count3 << endl;  
   
 // Count numbers divisible by 2  
 int count2 = count\_if(nums.begin(), nums.end(), DivisibleBy(2));  
 cout << "Numbers divisible by 2: " << count2 << endl;  
   
 // Find first number divisible by both 2 and 3 (i.e., 6)  
 auto it = find\_if(nums.begin(), nums.end(),   
 [](int n) { return n % 2 == 0 && n % 3 == 0; });  
   
 if (it != nums.end()) {  
 cout << "First number divisible by both 2 and 3: " << \*it << endl;  
 }  
   
 // Remove all numbers divisible by 4  
 nums.erase(  
 remove\_if(nums.begin(), nums.end(), DivisibleBy(4)),   
 nums.end()  
 );  
   
 cout << "After removing numbers divisible by 4: ";  
 for (int n : nums) {  
 cout << n << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### 8.5.5 Function Adapters

Function adapters modify the behavior of existing functors:

#include <iostream>  
#include <functional>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
bool isEven(int n) {  
 return n % 2 == 0;  
}  
  
int main() {  
 vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
   
 // Using not1 to negate a functor  
 auto notDivisible3 = not1(bind(modulus<int>(), placeholders::\_1, 3));  
   
 cout << "Numbers not divisible by 3: ";  
 for\_each(nums.begin(), nums.end(),   
 [&notDivisible3](int n) {  
 if (notDivisible3(n)) {  
 cout << n << " ";  
 }  
 });  
 cout << endl;  
   
 // Using negators  
 auto isOdd = not1(ptr\_fun(isEven)); // Negate isEven function  
   
 cout << "Odd numbers: ";  
 for\_each(nums.begin(), nums.end(),  
 [&isOdd](int n) {  
 if (isOdd(n)) {  
 cout << n << " ";  
 }  
 });  
 cout << endl;  
   
 // Using bind (C++11) to create new functors from existing ones  
 auto isMultipleOf3 = bind(modulus<int>(), placeholders::\_1, 3);  
 auto isNotMultipleOf3 = bind(not\_equal\_to<int>(),   
 bind(modulus<int>(), placeholders::\_1, 3), 0);  
   
 cout << "Is 9 multiple of 3? " << boolalpha << isMultipleOf3(9) << endl;  
 cout << "Is 10 not multiple of 3? " << isNotMultipleOf3(10) << endl;  
   
 return 0;  
}

### 8.5.6 Creating Complex Functors

More complex functors can contain multiple functions and store state:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <numeric>  
using namespace std;  
  
// Statistical functor that computes mean, min, max and count  
class Statistics {  
private:  
 int count = 0;  
 double sum = 0.0;  
 double min = numeric\_limits<double>::max();  
 double max = numeric\_limits<double>::lowest();  
   
public:  
 // Function call operator to process each value  
 void operator()(double value) {  
 ++count;  
 sum += value;  
   
 if (value < min) min = value;  
 if (value > max) max = value;  
 }  
   
 // Functions to retrieve statistics  
 int getCount() const { return count; }  
 double getSum() const { return sum; }  
 double getMean() const { return count > 0 ? sum / count : 0; }  
 double getMin() const { return min; }  
 double getMax() const { return max; }  
 double getRange() const { return max - min; }  
};  
  
// Functor that applies multiple operations based on rules  
class NumberClassifier {  
private:  
 vector<int> primes;  
 vector<int> evens;  
 vector<int> odds;  
   
public:  
 void operator()(int value) {  
 // Check if even or odd  
 if (value % 2 == 0) {  
 evens.push\_back(value);  
 } else {  
 odds.push\_back(value);  
 }  
   
 // Check if prime  
 if (value <= 1) return;  
 bool isPrime = true;  
 for (int i = 2; i \* i <= value; ++i) {  
 if (value % i == 0) {  
 isPrime = false;  
 break;  
 }  
 }  
 if (isPrime) {  
 primes.push\_back(value);  
 }  
 }  
   
 const vector<int>& getPrimes() const { return primes; }  
 const vector<int>& getEvens() const { return evens; }  
 const vector<int>& getOdds() const { return odds; }  
};  
  
int main() {  
 vector<double> values = {1.5, 3.7, 2.8, 9.2, 4.3, 5.1, 2.9, 8.4};  
   
 // Using the Statistics functor  
 Statistics stats = for\_each(values.begin(), values.end(), Statistics());  
   
 cout << "Number of values: " << stats.getCount() << endl;  
 cout << "Sum: " << stats.getSum() << endl;  
 cout << "Mean: " << stats.getMean() << endl;  
 cout << "Minimum: " << stats.getMin() << endl;  
 cout << "Maximum: " << stats.getMax() << endl;  
 cout << "Range: " << stats.getRange() << endl;  
   
 // Using the NumberClassifier functor  
 vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};  
 NumberClassifier classifier = for\_each(numbers.begin(), numbers.end(), NumberClassifier());  
   
 cout << "\nPrime numbers: ";  
 for (int prime : classifier.getPrimes()) {  
 cout << prime << " ";  
 }  
 cout << endl;  
   
 cout << "Even numbers: ";  
 for (int even : classifier.getEvens()) {  
 cout << even << " ";  
 }  
 cout << endl;  
   
 cout << "Odd numbers: ";  
 for (int odd : classifier.getOdds()) {  
 cout << odd << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### 8.5.7 Functors vs Lambdas

Modern C++ provides lambdas as a simpler alternative to functors in many cases:

#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
// Traditional functor approach  
class Multiplier {  
private:  
 int factor;  
   
public:  
 Multiplier(int f) : factor(f) {}  
   
 int operator()(int value) const {  
 return value \* factor;  
 }  
};  
  
int main() {  
 vector<int> nums = {1, 2, 3, 4, 5};  
 vector<int> result1(nums.size());  
 vector<int> result2(nums.size());  
 vector<int> result3(nums.size());  
   
 // Approach 1: Using functor  
 transform(nums.begin(), nums.end(), result1.begin(), Multiplier(3));  
   
 // Approach 2: Using lambda  
 transform(nums.begin(), nums.end(), result2.begin(),   
 [factor = 3](int value) { return value \* factor; });  
   
 // Approach 3: Using std::bind  
 transform(nums.begin(), nums.end(), result3.begin(),  
 bind(multiplies<int>(), placeholders::\_1, 3));  
   
 // Output results  
 cout << "Using functor: ";  
 for (int n : result1) cout << n << " ";  
 cout << endl;  
   
 cout << "Using lambda: ";  
 for (int n : result2) cout << n << " ";  
 cout << endl;  
   
 cout << "Using bind: ";  
 for (int n : result3) cout << n << " ";  
 cout << endl;  
   
 // When to use functors over lambdas:  
 // 1. When you need a reusable function object  
 // 2. When you need complex state management  
 // 3. When your functor is used in multiple places  
 // 4. When you need custom template parameters  
   
 return 0;  
}

### 8.5.8 Best Practices for Functors

1. **Keep functors simple and focused** - Each functor should have a single responsibility
2. **Use meaningful names** - Names should reflect what the functor does
3. **Make functors const-correct** - Mark the function call operator as const when it doesn’t modify the functor’s state
4. **Consider lambdas for simple cases** - Use lambdas for short, one-off operations
5. **Consider std::function for polymorphic function wrappers** - When you need to store different callable objects
6. **Use STL functors when possible** - Leverage existing functors before creating your own

## 8.6 Custom Comparators

Custom comparators allow you to define custom ordering rules for containers and algorithms. They’re essential when working with complex data types or when you need specific ordering behaviors.

### 8.6.1 Basic Concept of Comparators

A comparator is a function or function object that takes two arguments and returns a boolean indicating whether the first argument should be considered “less than” the second.

#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
// Custom comparator function  
bool descending(int a, int b) {  
 return a > b; // Sort in descending order  
}  
  
int main() {  
 vector<int> numbers = {5, 2, 8, 1, 9, 3, 7, 4, 6};  
   
 // Sort using custom comparator function  
 sort(numbers.begin(), numbers.end(), descending);  
   
 cout << "Sorted in descending order: ";  
 for (int num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Sort using lambda comparator (ascending)  
 sort(numbers.begin(), numbers.end(),   
 [](int a, int b) { return a < b; });  
   
 cout << "Sorted in ascending order: ";  
 for (int num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### 8.6.2 Comparator Classes

Comparators can also be implemented as classes, which is useful when they need to maintain state or be parameterized:

#include <iostream>  
#include <vector>  
#include <algorithm>  
using namespace std;  
  
// Custom comparator class  
class CustomCompare {  
private:  
 bool ascending;  
   
public:  
 CustomCompare(bool asc = true) : ascending(asc) {}  
   
 bool operator()(int a, int b) const {  
 return ascending ? (a < b) : (a > b);  
 }  
};  
  
int main() {  
 vector<int> numbers = {5, 2, 8, 1, 9, 3, 7, 4, 6};  
   
 // Sort using comparator object (descending)  
 sort(numbers.begin(), numbers.end(), CustomCompare(false));  
   
 cout << "Sorted in descending order: ";  
 for (int num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 // Sort using same comparator class but ascending  
 sort(numbers.begin(), numbers.end(), CustomCompare(true));  
   
 cout << "Sorted in ascending order: ";  
 for (int num : numbers) {  
 cout << num << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### 8.6.3 Comparators for Custom Types

Custom comparators are particularly useful for custom types:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <string>  
using namespace std;  
  
// Student class  
class Student {  
public:  
 string name;  
 double gpa;  
 int id;  
   
 Student(string n, double g, int i) : name(n), gpa(g), id(i) {}  
   
 // For easy output  
 friend ostream& operator<<(ostream& os, const Student& s) {  
 return os << s.name << " (ID: " << s.id << ", GPA: " << s.gpa << ")";  
 }  
};  
  
// Comparator by name  
struct CompareByName {  
 bool operator()(const Student& a, const Student& b) const {  
 return a.name < b.name;  
 }  
};  
  
// Comparator by GPA  
struct CompareByGPA {  
 bool operator()(const Student& a, const Student& b) const {  
 return a.gpa > b.gpa; // Note: Higher GPA first (descending)  
 }  
};  
  
// Comparator by ID  
struct CompareByID {  
 bool operator()(const Student& a, const Student& b) const {  
 return a.id < b.id;  
 }  
};  
  
int main() {  
 vector<Student> students = {  
 {"Alice", 3.8, 103},  
 {"Bob", 3.6, 101},  
 {"Charlie", 4.0, 105},  
 {"David", 3.9, 102},  
 {"Eva", 3.7, 104}  
 };  
   
 // Sort by name  
 sort(students.begin(), students.end(), CompareByName());  
   
 cout << "Sorted by name:\n";  
 for (const auto& student : students) {  
 cout << student << endl;  
 }  
   
 // Sort by GPA (descending)  
 sort(students.begin(), students.end(), CompareByGPA());  
   
 cout << "\nSorted by GPA (highest first):\n";  
 for (const auto& student : students) {  
 cout << student << endl;  
 }  
   
 // Sort by ID  
 sort(students.begin(), students.end(), CompareByID());  
   
 cout << "\nSorted by ID:\n";  
 for (const auto& student : students) {  
 cout << student << endl;  
 }  
   
 // Using lambda for multi-criteria sort (GPA then name)  
 sort(students.begin(), students.end(),   
 [](const Student& a, const Student& b) {  
 if (a.gpa != b.gpa) {  
 return a.gpa > b.gpa; // First by GPA (descending)  
 }  
 return a.name < b.name; // Then by name (ascending)  
 });  
   
 cout << "\nSorted by GPA (then by name for ties):\n";  
 for (const auto& student : students) {  
 cout << student << endl;  
 }  
   
 return 0;  
}

### 8.6.4 Comparators with Associative Containers

Custom comparators are essential for associative containers like set and map, where they define the ordering of elements:

#include <iostream>  
#include <set>  
#include <map>  
#include <string>  
using namespace std;  
  
// Case-insensitive string comparator  
struct CaseInsensitiveCompare {  
 bool operator()(const string& a, const string& b) const {  
 // Convert both strings to lowercase for comparison  
 string a\_lower = a;  
 string b\_lower = b;  
   
 transform(a\_lower.begin(), a\_lower.end(), a\_lower.begin(), ::tolower);  
 transform(b\_lower.begin(), b\_lower.end(), b\_lower.begin(), ::tolower);  
   
 return a\_lower < b\_lower;  
 }  
};  
  
int main() {  
 // Set with custom comparator  
 set<string, CaseInsensitiveCompare> names;  
   
 // Inserting names  
 names.insert("John");  
 names.insert("alice");  
 names.insert("Bob");  
 names.insert("CHARLIE");  
 names.insert("Alice"); // Will not be inserted since "alice" is already present  
   
 cout << "Case-insensitive set of names:" << endl;  
 for (const auto& name : names) {  
 cout << name << endl;  
 }  
   
 // Map with custom comparator  
 map<string, int, CaseInsensitiveCompare> nameCounts;  
   
 // Counting names regardless of case  
 vector<string> words = {"Apple", "banana", "APPLE", "orange", "Banana", "ORANGE"};  
   
 for (const auto& word : words) {  
 ++nameCounts[word];  
 }  
   
 cout << "\nCase-insensitive word counts:" << endl;  
 for (const auto& [word, count] : nameCounts) {  
 cout << word << ": " << count << endl;  
 }  
   
 return 0;  
}

### 8.6.5 Comparators with STL Algorithms

Custom comparators can be used with many STL algorithms:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <functional>  
using namespace std;  
  
// Person structure  
struct Person {  
 string name;  
 int age;  
   
 Person(string n, int a) : name(n), age(a) {}  
   
 friend ostream& operator<<(ostream& os, const Person& p) {  
 return os << p.name << " (age: " << p.age << ")";  
 }  
};  
  
int main() {  
 vector<Person> people = {  
 {"Alice", 25},  
 {"Bob", 30},  
 {"Charlie", 22},  
 {"David", 35},  
 {"Eva", 28}  
 };  
   
 // Find the oldest person  
 auto oldest = max\_element(people.begin(), people.end(),  
 [](const Person& a, const Person& b) {  
 return a.age < b.age;  
 });  
   
 cout << "Oldest person: " << \*oldest << endl;  
   
 // Find the youngest person  
 auto youngest = min\_element(people.begin(), people.end(),  
 [](const Person& a, const Person& b) {  
 return a.age < b.age;  
 });  
   
 cout << "Youngest person: " << \*youngest << endl;  
   
 // Sort by name  
 sort(people.begin(), people.end(),  
 [](const Person& a, const Person& b) {  
 return a.name < b.name;  
 });  
   
 cout << "\nPeople sorted by name:" << endl;  
 for (const auto& person : people) {  
 cout << person << endl;  
 }  
   
 // Partial sort - get 3 youngest people  
 partial\_sort(people.begin(), people.begin() + 3, people.end(),  
 [](const Person& a, const Person& b) {  
 return a.age < b.age;  
 });  
   
 cout << "\n3 youngest people:" << endl;  
 for (auto it = people.begin(); it != people.begin() + 3; ++it) {  
 cout << \*it << endl;  
 }  
   
 // Binary search with custom comparator  
 vector<int> numbers = {10, 20, 30, 40, 50, 60, 70, 80, 90};  
   
 // Find element closest to 45  
 auto closest = lower\_bound(numbers.begin(), numbers.end(), 45);  
   
 if (closest != numbers.end()) {  
 cout << "\nClosest number to 45 is: " << \*closest << endl;  
 }  
   
 return 0;  
}

### 8.6.6 Complex Custom Comparators

More complex sorting scenarios often require sophisticated comparators:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <string>  
#include <map>  
using namespace std;  
  
// Product class  
class Product {  
public:  
 string name;  
 double price;  
 int stock;  
 string category;  
   
 Product(string n, double p, int s, string c)  
 : name(n), price(p), stock(s), category(c) {}  
   
 friend ostream& operator<<(ostream& os, const Product& p) {  
 return os << p.name << " - $" << p.price  
 << " (Stock: " << p.stock  
 << ", Category: " << p.category << ")";  
 }  
};  
  
// Multi-criteria comparator class  
class ProductComparator {  
private:  
 enum class SortCriterion { Name, Price, Stock, Category };  
 enum class SortDirection { Ascending, Descending };  
   
 vector<pair<SortCriterion, SortDirection>> criteria;  
   
public:  
 // Add a sorting criterion  
 void addCriterion(SortCriterion crit, SortDirection dir = SortDirection::Ascending) {  
 criteria.push\_back({crit, dir});  
 }  
   
 // Clear all criteria  
 void clearCriteria() {  
 criteria.clear();  
 }  
   
 // Comparison operator  
 bool operator()(const Product& a, const Product& b) const {  
 for (const auto& [criterion, direction] : criteria) {  
 bool result;  
   
 switch (criterion) {  
 case SortCriterion::Name:  
 if (a.name != b.name) {  
 result = a.name < b.name;  
 break;  
 }  
 continue; // Move to next criterion  
   
 case SortCriterion::Price:  
 if (a.price != b.price) {  
 result = a.price < b.price;  
 break;  
 }  
 continue; // Move to next criterion  
   
 case SortCriterion::Stock:  
 if (a.stock != b.stock) {  
 result = a.stock < b.stock;  
 break;  
 }  
 continue; // Move to next criterion  
   
 case SortCriterion::Category:  
 if (a.category != b.category) {  
 result = a.category < b.category;  
 break;  
 }  
 continue; // Move to next criterion  
 }  
   
 // Apply sort direction  
 return direction == SortDirection::Ascending ? result : !result;  
 }  
   
 // If all criteria match, maintain stable sort  
 return false;  
 }  
   
 // Factory methods for common comparators  
 static ProductComparator byNameAsc() {  
 ProductComparator comp;  
 comp.addCriterion(SortCriterion::Name, SortDirection::Ascending);  
 return comp;  
 }  
   
 static ProductComparator byPriceDesc() {  
 ProductComparator comp;  
 comp.addCriterion(SortCriterion::Price, SortDirection::Descending);  
 return comp;  
 }  
   
 static ProductComparator byCategoryThenPrice() {  
 ProductComparator comp;  
 comp.addCriterion(SortCriterion::Category, SortDirection::Ascending);  
 comp.addCriterion(SortCriterion::Price, SortDirection::Ascending);  
 return comp;  
 }  
};  
  
int main() {  
 vector<Product> products = {  
 {"Laptop", 999.99, 15, "Electronics"},  
 {"Smartphone", 699.99, 25, "Electronics"},  
 {"Desk", 249.99, 5, "Furniture"},  
 {"Chair", 149.99, 12, "Furniture"},  
 {"Mouse", 29.99, 50, "Electronics"},  
 {"Keyboard", 89.99, 30, "Electronics"},  
 {"Bookshelf", 199.99, 8, "Furniture"},  
 {"Monitor", 249.99, 20, "Electronics"}  
 };  
   
 // Sort by name (ascending)  
 sort(products.begin(), products.end(), ProductComparator::byNameAsc());  
   
 cout << "Products sorted by name (ascending):\n";  
 for (const auto& product : products) {  
 cout << product << endl;  
 }  
   
 // Sort by price (descending)  
 sort(products.begin(), products.end(), ProductComparator::byPriceDesc());  
   
 cout << "\nProducts sorted by price (descending):\n";  
 for (const auto& product : products) {  
 cout << product << endl;  
 }  
   
 // Sort by category, then by price  
 sort(products.begin(), products.end(), ProductComparator::byCategoryThenPrice());  
   
 cout << "\nProducts sorted by category, then by price:\n";  
 for (const auto& product : products) {  
 cout << product << endl;  
 }  
   
 // Custom multi-level sorting using a configurable comparator  
 ProductComparator customComp;  
 customComp.addCriterion(ProductComparator::SortCriterion::Stock,   
 ProductComparator::SortDirection::Ascending);  
 customComp.addCriterion(ProductComparator::SortCriterion::Name,   
 ProductComparator::SortDirection::Descending);  
   
 sort(products.begin(), products.end(), customComp);  
   
 cout << "\nProducts sorted by stock (ascending), then by name (descending):\n";  
 for (const auto& product : products) {  
 cout << product << endl;  
 }  
   
 return 0;  
}

### 8.6.7 Using Comparators with priority\_queue

The priority\_queue container adapter requires a special form of comparator:

#include <iostream>  
#include <queue>  
#include <vector>  
#include <string>  
using namespace std;  
  
// Task class  
class Task {  
public:  
 string description;  
 int priority; // Higher number means higher priority  
   
 Task(string desc, int prio) : description(desc), priority(prio) {}  
   
 // For easy output  
 friend ostream& operator<<(ostream& os, const Task& task) {  
 return os << task.description << " (Priority: " << task.priority << ")";  
 }  
};  
  
// Comparator for Task priority queue  
struct TaskCompare {  
 // Note: priority\_queue uses this to determine if a is "less" priority than b  
 // For a max-heap (highest priority first), we return true if a should come after b  
 bool operator()(const Task& a, const Task& b) const {  
 return a.priority < b.priority; // This creates a max-heap  
 }  
};  
  
int main() {  
 // Create a priority queue of tasks  
 priority\_queue<Task, vector<Task>, TaskCompare> taskQueue;  
   
 // Add tasks  
 taskQueue.push(Task("Check emails", 2));  
 taskQueue.push(Task("Fix critical bug", 5));  
 taskQueue.push(Task("Write documentation", 1));  
 taskQueue.push(Task("Code review", 3));  
 taskQueue.push(Task("Client meeting", 4));  
   
 cout << "Tasks in priority order:\n";  
 while (!taskQueue.top()) {  
 cout << taskQueue.top() << endl;  
 taskQueue.pop();  
 }  
   
 // Using a min-heap with lambda (lowest priority first)  
 auto minCompare = [](const Task& a, const Task& b) {  
 return a.priority > b.priority; // Return true if a should come after b  
 };  
   
 priority\_queue<Task, vector<Task>, decltype(minCompare)> minTaskQueue(minCompare);  
   
 // Add tasks to min-heap  
 minTaskQueue.push(Task("Check emails", 2));  
 minTaskQueue.push(Task("Fix critical bug", 5));  
 minTaskQueue.push(Task("Write documentation", 1));  
 minTaskQueue.push(Task("Code review", 3));  
 minTaskQueue.push(Task("Client meeting", 4));  
   
 cout << "\nTasks in reverse priority order:\n";  
 while (!minTaskQueue.empty()) {  
 cout << minTaskQueue.top() << endl;  
 minTaskQueue.pop();  
 }  
   
 return 0;  
}

### 8.6.8 Stable and Unstable Sorting

Using comparators with stable and unstable sorting algorithms:

#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <string>  
using namespace std;  
  
struct Person {  
 string name;  
 int age;  
   
 Person(string n, int a) : name(n), age(a) {}  
   
 friend ostream& operator<<(ostream& os, const Person& p) {  
 return os << p.name << " (age: " << p.age << ")";  
 }  
};  
  
int main() {  
 vector<Person> people = {  
 {"Alice", 30}, {"Bob", 25}, {"Charlie", 30},  
 {"David", 25}, {"Eva", 30}, {"Frank", 25}  
 };  
   
 // Make a copy for stable\_sort  
 vector<Person> stablePeople = people;  
   
 // Unstable sort by age (may reorder people of the same age)  
 sort(people.begin(), people.end(),  
 [](const Person& a, const Person& b) {  
 return a.age < b.age;  
 });  
   
 cout << "After regular sort by age (unstable):\n";  
 for (const auto& person : people) {  
 cout << person << endl;  
 }  
   
 // Stable sort by age (preserves original order of people with same age)  
 stable\_sort(stablePeople.begin(), stablePeople.end(),  
 [](const Person& a, const Person& b) {  
 return a.age < b.age;  
 });  
   
 cout << "\nAfter stable sort by age:\n";  
 for (const auto& person : stablePeople) {  
 cout << person << endl;  
 }  
   
 // Custom multi-criteria comparator preserving stability  
 cout << "\nMulti-criteria stable sort (age then name):\n";  
   
 // Sort by name first  
 stable\_sort(stablePeople.begin(), stablePeople.end(),  
 [](const Person& a, const Person& b) {  
 return a.name < b.name;  
 });  
   
 // Then sort by age (will preserve name order for same age)  
 stable\_sort(stablePeople.begin(), stablePeople.end(),  
 [](const Person& a, const Person& b) {  
 return a.age < b.age;  
 });  
   
 for (const auto& person : stablePeople) {  
 cout << person << endl;  
 }  
   
 return 0;  
}

### 8.6.9 Best Practices for Custom Comparators

1. **Follow the strict weak ordering requirements**:
   * If a < b is true, then b < a must be false
   * If a < b is false and b < a is false, then a and b are equivalent
   * If a < b and b < c, then a < c (transitivity)
2. **Make your comparators const-correct**:

* bool operator()(const Type& a, const Type& b) const { ... }

1. **Keep comparators simple and focused**:
   * Each comparator should have a clear sorting criterion
   * Use multiple comparators for complex sorting scenarios
2. **Use lambdas for simple, one-off comparisons**:

* sort(v.begin(), v.end(), [](auto& a, auto& b) { return a.value < b.value; });

1. **Use function objects for reusable or stateful comparators**:

* struct MyComparator {  
   bool operator()(const Type& a, const Type& b) const { ... }  
  };

1. **Be consistent with the meaning of the comparison**:
   * Convention: return true if a should come before b
   * Exception: priority\_queue interprets the comparison differently
2. **Be mindful of stability when needed**:
   * Use stable\_sort when the original order matters for equivalent elements
   * Use multi-pass sorting for complex criteria while preserving order
3. **Avoid expensive operations in comparators**:
   * Comparison functions are called many times during sorting
   * Cache results or precompute values when appropriate
4. **Test edge cases**:
   * Empty strings, zero/negative values, duplicate elements
   * Very large or very small values
   * Make sure your comparator behaves correctly in all situations

# Chapter 9: File Handling in C++ (Part 1)

## Introduction to File Handling

File handling is one of the most important aspects of programming, allowing applications to store and retrieve data persistently. C++ provides robust file handling capabilities through its stream classes, which offer a consistent and type-safe approach to dealing with files.

In this chapter, we’ll explore how to perform file operations in C++, starting with the fundamental stream concepts and progressing to practical file reading and writing techniques.

## 9.1 Streams (ifstream, ofstream, fstream)

### Stream Hierarchy in C++

C++ file handling is built around the concept of streams. A stream is an abstraction that represents a sequence of bytes, whether they’re coming from a file, keyboard, network, or any other source or destination.

The C++ stream classes are organized in a hierarchy:

ios\_base  
 |  
 v  
 ios  
 /\  
 / \  
 / \  
 / \  
 v v  
 istream ostream  
 / | | \  
 / | | \  
 v v v v  
ifstream iostream ofstream  
 |  
 v  
 fstream

* **ios\_base**: The base class that provides basic functionality common to all stream classes
* **ios**: Adds formatting capabilities to the streams
* **istream**: Base class for input streams
* **ostream**: Base class for output streams
* **iostream**: Combined input and output capabilities
* **ifstream**: Input file stream - for reading from files
* **ofstream**: Output file stream - for writing to files
* **fstream**: File stream with both input and output capabilities

### Header Files for File Handling

#include <fstream> // For file streams (ifstream, ofstream, fstream)  
#include <iostream> // For standard streams (cin, cout)  
#include <sstream> // For string streams (stringstream)

### ifstream (Input File Stream)

An ifstream object is used to read data from a file:

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
int main() {  
 // Create an input file stream  
 ifstream inputFile;  
   
 // Open file  
 inputFile.open("example.txt");  
   
 // Alternative way to open file (constructor method)  
 // ifstream inputFile("example.txt");  
   
 // Check if file opened successfully  
 if (!inputFile.is\_open()) {  
 cerr << "Error opening file!" << endl;  
 return 1;  
 }  
   
 // Read data from file  
 string line;  
 while (getline(inputFile, line)) {  
 cout << line << endl;  
 }  
   
 // Close the file when done  
 inputFile.close();  
   
 return 0;  
}

### ofstream (Output File Stream)

An ofstream object is used to write data to a file:

#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main() {  
 // Create an output file stream  
 ofstream outputFile;  
   
 // Open file (will create if doesn't exist, or truncate if it does)  
 outputFile.open("output.txt");  
   
 // Alternative way to open file (constructor method)  
 // ofstream outputFile("output.txt");  
   
 // Check if file opened successfully  
 if (!outputFile.is\_open()) {  
 cerr << "Error opening file for writing!" << endl;  
 return 1;  
 }  
   
 // Write data to file  
 outputFile << "Hello, File I/O in C++!" << endl;  
 outputFile << "This is a second line of text." << endl;  
   
 // Close the file when done  
 outputFile.close();  
   
 cout << "Data written to file successfully." << endl;  
   
 return 0;  
}

### fstream (File Stream)

An fstream object can be used for both reading and writing:

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
int main() {  
 // Create a file stream  
 fstream fileStream;  
   
 // Open for both reading and writing  
 fileStream.open("data.txt", ios::in | ios::out | ios::app);  
   
 // Check if file opened successfully  
 if (!fileStream.is\_open()) {  
 cerr << "Error opening file!" << endl;  
 return 1;  
 }  
   
 // Write to file  
 fileStream << "Adding new content to the file." << endl;  
   
 // Reset file position to beginning for reading  
 fileStream.seekg(0, ios::beg);  
   
 // Read file content  
 string line;  
 cout << "File content:" << endl;  
 while (getline(fileStream, line)) {  
 cout << line << endl;  
 }  
   
 // Close the file  
 fileStream.close();  
   
 return 0;  
}

### File Opening Modes

When opening a file, you can specify a mode that determines how the file should be treated:

| Mode Flag | Description |
| --- | --- |
| ios::in | Open for input operations (reading) |
| ios::out | Open for output operations (writing) - truncates existing file by default |
| ios::app | Append mode - all output is appended to the end of the file |
| ios::ate | Open and seek to the end of file immediately after opening |
| ios::trunc | Truncate the file to zero length if it exists (default for ios::out) |
| ios::binary | Open in binary mode rather than text mode |

You can combine these flags using the bitwise OR operator (|):

// Open for reading and writing without truncating  
fstream file("data.txt", ios::in | ios::out);  
  
// Open in append mode  
ofstream outFile("log.txt", ios::app);  
  
// Open in binary mode for reading  
ifstream binFile("image.png", ios::binary);  
  
// Open for output, create if doesn't exist, and go to the end  
ofstream logFile("server.log", ios::out | ios::app);

### Stream States and Error Handling

Stream objects maintain state flags that indicate their current condition:

* **good()**: Returns true if no errors have occurred and the stream is ready for I/O
* **eof()**: Returns true if end-of-file has been reached
* **fail()**: Returns true if a formatting error has occurred
* **bad()**: Returns true if an unrecoverable error has occurred

#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main() {  
 ifstream file("nonexistent.txt");  
   
 if (file.good()) {  
 cout << "File opened successfully." << endl;  
 } else {  
 cout << "File state flags:" << endl;  
 cout << "- EOF flag: " << (file.eof() ? "Set" : "Not set") << endl;  
 cout << "- Fail flag: " << (file.fail() ? "Set" : "Not set") << endl;  
 cout << "- Bad flag: " << (file.bad() ? "Set" : "Not set") << endl;  
 }  
   
 // Alternative way to check if file opened successfully  
 if (!file) {  
 cerr << "Error opening file!" << endl;  
 }  
   
 // Clear error state flags  
 file.clear();  
   
 // Try to open a different file  
 file.open("existing.txt");  
   
 // Reset error state for specific error  
 if (file.fail()) {  
 file.clear(ios::failbit);  
 }  
   
 return 0;  
}

### File Position Indicators

Stream objects maintain pointers to the current position in the file:

* **tellg()**: Returns the current position of the “get” pointer (input)
* **seekg()**: Sets the position of the “get” pointer (input)
* **tellp()**: Returns the current position of the “put” pointer (output)
* **seekp()**: Sets the position of the “put” pointer (output)

The seek functions take two parameters: the position and the seek direction:

#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main() {  
 fstream file("data.txt", ios::in | ios::out);  
   
 if (!file) {  
 cerr << "Error opening file!" << endl;  
 return 1;  
 }  
   
 // Get current position  
 streampos initialPos = file.tellg();  
 cout << "Initial position: " << initialPos << endl;  
   
 // Move to the 10th byte of the file  
 file.seekg(10, ios::beg);  
   
 // Move 5 bytes forward from current position  
 file.seekg(5, ios::cur);  
   
 // Move 10 bytes before the end of file  
 file.seekg(-10, ios::end);  
   
 // Return to beginning of file  
 file.seekg(0, ios::beg);  
   
 // Go to end of file  
 file.seekg(0, ios::end);  
 streampos fileSize = file.tellg();  
 cout << "File size: " << fileSize << " bytes" << endl;  
   
 file.close();  
 return 0;  
}

## 9.2 Reading and Writing Text Files

### Opening and Closing Files

**Opening Files**:

There are two primary ways to open a file:

1. Using the constructor:

ifstream inFile("input.txt");  
ofstream outFile("output.txt");  
fstream file("data.txt", ios::in | ios::out);

1. Using the open() method:

ifstream inFile;  
inFile.open("input.txt");  
  
ofstream outFile;  
outFile.open("output.txt", ios::out);  
  
fstream file;  
file.open("data.txt", ios::in | ios::out | ios::app);

**Closing Files**:

Files should be closed when they’re no longer needed:

inFile.close();  
outFile.close();  
file.close();

When a file stream object goes out of scope, its destructor automatically closes the file, but it’s good practice to explicitly close files when you’re done with them.

### Writing to Text Files

**Basic Writing**:

#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main() {  
 ofstream outFile("sample.txt");  
   
 if (!outFile) {  
 cerr << "Error opening file for writing!" << endl;  
 return 1;  
 }  
   
 // Write basic data types  
 int id = 101;  
 string name = "John Doe";  
 double salary = 50000.50;  
   
 outFile << "ID: " << id << endl;  
 outFile << "Name: " << name << endl;  
 outFile << "Salary: " << salary << endl;  
   
 // Write formatted data  
 outFile << fixed << setprecision(2);  
 outFile << "Formatted salary: $" << salary << endl;  
   
 outFile.close();  
 cout << "Data written successfully!" << endl;  
   
 return 0;  
}

**Append Mode**:

ofstream logFile("application.log", ios::app);  
logFile << "Log entry at " << getCurrentTime() << ": System started" << endl;

**Writing Character by Character**:

ofstream outFile("char\_output.txt");  
string message = "Character by character";  
  
for (char c : message) {  
 outFile.put(c);  
}  
  
// Or using another method  
for (size\_t i = 0; i < message.length(); ++i) {  
 outFile.write(&message[i], 1);  
}

**Writing Binary Data to Text File**:

// Not recommended, but possible  
ofstream outFile("mixed.txt");  
int num = 12345;  
outFile.write(reinterpret\_cast<const char\*>(&num), sizeof(num));

### Reading from Text Files

**Reading Line by Line**:

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
int main() {  
 ifstream inFile("sample.txt");  
   
 if (!inFile) {  
 cerr << "Error opening file for reading!" << endl;  
 return 1;  
 }  
   
 string line;  
   
 // Read file line by line  
 cout << "File content:" << endl;  
 while (getline(inFile, line)) {  
 cout << line << endl;  
 }  
   
 inFile.close();  
 return 0;  
}

**Reading Word by Word**:

ifstream inFile("text.txt");  
string word;  
  
while (inFile >> word) {  
 cout << word << endl;  
}

**Reading Character by Character**:

ifstream inFile("char\_input.txt");  
char ch;  
  
while (inFile.get(ch)) {  
 cout << ch;  
}

**Reading Formatted Data**:

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
int main() {  
 ifstream inFile("employee.txt");  
   
 if (!inFile) {  
 cerr << "Error opening file!" << endl;  
 return 1;  
 }  
   
 int id;  
 string name;  
 double salary;  
   
 // Format of file is assumed to be:  
 // 101 John\_Doe 50000.50  
 // 102 Jane\_Smith 55000.75  
   
 cout << "Employee data:" << endl;  
 cout << "--------------" << endl;  
   
 while (inFile >> id >> name >> salary) {  
 // Replace underscores with spaces  
 replace(name.begin(), name.end(), '\_', ' ');  
   
 cout << "ID: " << id << endl;  
 cout << "Name: " << name << endl;  
 cout << "Salary: $" << fixed << setprecision(2) << salary << endl;  
 cout << endl;  
 }  
   
 inFile.close();  
 return 0;  
}

**Checking for End of File**:

ifstream inFile("data.txt");  
string line;  
  
while (!inFile.eof()) {  
 getline(inFile, line);  
   
 // Important: check if read was successful before processing  
 if (!inFile.fail()) {  
 cout << line << endl;  
 }  
}  
  
// Better approach to avoid duplicating the last line  
while (getline(inFile, line)) {  
 cout << line << endl;  
}

### Reading and Writing Mixed Data

#include <iostream>  
#include <fstream>  
#include <vector>  
#include <string>  
using namespace std;  
  
struct Student {  
 int id;  
 string name;  
 double gpa;  
};  
  
void saveStudents(const vector<Student>& students, const string& filename) {  
 ofstream outFile(filename);  
   
 if (!outFile) {  
 cerr << "Error opening file for writing!" << endl;  
 return;  
 }  
   
 for (const auto& student : students) {  
 // Replace spaces with underscores for easier parsing  
 string nameNoSpaces = student.name;  
 replace(nameNoSpaces.begin(), nameNoSpaces.end(), ' ', '\_');  
   
 outFile << student.id << " "  
 << nameNoSpaces << " "  
 << student.gpa << endl;  
 }  
   
 outFile.close();  
}  
  
vector<Student> loadStudents(const string& filename) {  
 vector<Student> students;  
 ifstream inFile(filename);  
   
 if (!inFile) {  
 cerr << "Error opening file for reading!" << endl;  
 return students;  
 }  
   
 Student temp;  
 string nameWithUnderscores;  
   
 while (inFile >> temp.id >> nameWithUnderscores >> temp.gpa) {  
 // Replace underscores with spaces  
 replace(nameWithUnderscores.begin(), nameWithUnderscores.end(), '\_', ' ');  
 temp.name = nameWithUnderscores;  
   
 students.push\_back(temp);  
 }  
   
 inFile.close();  
 return students;  
}  
  
int main() {  
 // Create some student records  
 vector<Student> students = {  
 {1001, "John Doe", 3.75},  
 {1002, "Jane Smith", 3.90},  
 {1003, "Bob Johnson", 3.45}  
 };  
   
 // Save to file  
 saveStudents(students, "students.txt");  
   
 // Load from file  
 vector<Student> loadedStudents = loadStudents("students.txt");  
   
 // Display loaded students  
 cout << "Loaded students:" << endl;  
 cout << "-----------------" << endl;  
   
 for (const auto& student : loadedStudents) {  
 cout << "ID: " << student.id << endl;  
 cout << "Name: " << student.name << endl;  
 cout << "GPA: " << student.gpa << endl;  
 cout << endl;  
 }  
   
 return 0;  
}

### Error Handling and File Management

**Proper Error Checking**:

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
bool processFile(const string& filename) {  
 ifstream inFile(filename);  
   
 // Check if file opened successfully  
 if (!inFile) {  
 cerr << "Error: Could not open file " << filename << endl;  
 return false;  
 }  
   
 string line;  
 int lineCount = 0;  
   
 while (getline(inFile, line)) {  
 try {  
 ++lineCount;  
 // Process line here  
 cout << "Line " << lineCount << ": " << line << endl;  
   
 // Example of detecting a format error  
 if (line.empty() && lineCount == 1) {  
 throw runtime\_error("First line cannot be empty");  
 }  
 }  
 catch (const exception& e) {  
 cerr << "Error processing line " << lineCount << ": " << e.what() << endl;  
 inFile.close();  
 return false;  
 }  
 }  
   
 // Check if any errors occurred during reading  
 if (inFile.bad()) {  
 cerr << "Error: I/O error while reading file" << endl;  
 inFile.close();  
 return false;  
 }  
 else if (inFile.fail() && !inFile.eof()) {  
 cerr << "Error: Non-integer data encountered" << endl;  
 inFile.close();  
 return false;  
 }  
   
 inFile.close();  
 cout << "Successfully processed " << lineCount << " lines" << endl;  
 return true;  
}  
  
int main() {  
 if (processFile("example.txt")) {  
 cout << "File processing succeeded" << endl;  
 }  
 else {  
 cout << "File processing failed" << endl;  
 }  
   
 return 0;  
}

**Using RAII for File Handling**:

Resource Acquisition Is Initialization (RAII) is a programming idiom where resource management is tied to object lifetime:

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
class FileHandler {  
private:  
 ifstream file;  
   
public:  
 FileHandler(const string& filename) : file(filename) {  
 if (!file) {  
 throw runtime\_error("Could not open file: " + filename);  
 }  
 }  
   
 // No need to explicitly close the file - destructor will handle it  
 ~FileHandler() {  
 if (file.is\_open()) {  
 file.close();  
 }  
 }  
   
 bool getLine(string& line) {  
 return static\_cast<bool>(getline(file, line));  
 }  
   
 bool isGood() const {  
 return file.good();  
 }  
};  
  
int main() {  
 try {  
 FileHandler handler("example.txt");  
   
 string line;  
 while (handler.getLine(line)) {  
 cout << line << endl;  
 }  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 return 1;  
 }  
   
 return 0;  
}

### Best Practices for File Handling

1. **Always check if file operations succeeded** before proceeding with reading or writing
2. **Close files explicitly** when you’re done with them
3. **Use appropriate opening modes** for your specific needs
4. **Handle errors gracefully** with proper error messages
5. **Use RAII principles** for automatic resource management
6. **Avoid mixing binary and text operations** on the same file
7. **Be careful with file paths** - consider platform differences
8. **Use stringstreams for string parsing and formatting** before file operations
9. **Implement proper error recovery** when file operations fail
10. **Consider file permissions** when accessing files

### Practical Example: A Simple CSV Parser

#include <iostream>  
#include <fstream>  
#include <sstream>  
#include <vector>  
#include <string>  
using namespace std;  
  
vector<vector<string>> readCSV(const string& filename, char delimiter = ',') {  
 vector<vector<string>> data;  
 ifstream file(filename);  
   
 if (!file.is\_open()) {  
 cerr << "Error opening file: " << filename << endl;  
 return data;  
 }  
   
 string line;  
 while (getline(file, line)) {  
 vector<string> row;  
 stringstream ss(line);  
 string cell;  
   
 while (getline(ss, cell, delimiter)) {  
 row.push\_back(cell);  
 }  
   
 data.push\_back(row);  
 }  
   
 file.close();  
 return data;  
}  
  
void writeCSV(const string& filename, const vector<vector<string>>& data, char delimiter = ',') {  
 ofstream file(filename);  
   
 if (!file.is\_open()) {  
 cerr << "Error opening file for writing: " << filename << endl;  
 return;  
 }  
   
 for (const auto& row : data) {  
 for (size\_t i = 0; i < row.size(); ++i) {  
 file << row[i];  
 if (i < row.size() - 1) {  
 file << delimiter;  
 }  
 }  
 file << endl;  
 }  
   
 file.close();  
}  
  
int main() {  
 // Example: Reading a CSV file  
 vector<vector<string>> csvData = readCSV("employees.csv");  
   
 cout << "CSV data:" << endl;  
 for (const auto& row : csvData) {  
 for (const auto& cell : row) {  
 cout << cell << "\t";  
 }  
 cout << endl;  
 }  
   
 // Example: Modifying and writing a CSV file  
 vector<vector<string>> outputData = {  
 {"Name", "Department", "Salary"},  
 {"John Doe", "Engineering", "75000"},  
 {"Jane Smith", "Marketing", "65000"},  
 {"Bob Johnson", "Finance", "85000"}  
 };  
   
 writeCSV("output.csv", outputData);  
 cout << "Output CSV file written successfully!" << endl;  
   
 return 0;  
}

### File Handling with Exception Safety

#include <iostream>  
#include <fstream>  
#include <stdexcept>  
using namespace std;  
  
string readEntireFile(const string& filename) {  
 ifstream file(filename);  
   
 if (!file) {  
 throw runtime\_error("Could not open file: " + filename);  
 }  
   
 // Read entire file content  
 string content((istreambuf\_iterator<char>(file)),   
 istreambuf\_iterator<char>());  
   
 if (file.bad()) {  
 throw runtime\_error("I/O error while reading file: " + filename);  
 }  
   
 return content;  
}  
  
void writeStringToFile(const string& filename, const string& content) {  
 ofstream file(filename);  
   
 if (!file) {  
 throw runtime\_error("Could not open file for writing: " + filename);  
 }  
   
 file << content;  
   
 if (!file) {  
 throw runtime\_error("Error writing to file: " + filename);  
 }  
}  
  
int main() {  
 try {  
 // Read a file  
 string content = readEntireFile("input.txt");  
   
 // Process the content  
 content += "\nAdditional text added programmatically.";  
   
 // Write to a new file  
 writeStringToFile("output\_modified.txt", content);  
   
 cout << "File processing completed successfully!" << endl;  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 return 1;  
 }  
   
 return 0;  
}

### Using String Streams for File Processing

String streams are particularly useful when you need to parse or format data in memory before file operations:

#include <iostream>  
#include <fstream>  
#include <sstream>  
#include <vector>  
#include <string>  
using namespace std;  
  
// Parse a line from a file into structured data  
struct Record {  
 int id;  
 string name;  
 double value;  
};  
  
Record parseLine(const string& line) {  
 Record rec;  
 stringstream ss(line);  
   
 ss >> rec.id;  
 ss >> rec.name;  
 ss >> rec.value;  
   
 if (ss.fail()) {  
 throw runtime\_error("Failed to parse line: " + line);  
 }  
   
 return rec;  
}  
  
// Create formatted output for a file  
string formatRecord(const Record& rec) {  
 stringstream ss;  
 ss << rec.id << " "   
 << rec.name << " "   
 << fixed << setprecision(2) << rec.value;  
 return ss.str();  
}  
  
int main() {  
 // Read file into records  
 vector<Record> records;  
 ifstream inFile("records.txt");  
   
 if (!inFile) {  
 cerr << "Error opening input file" << endl;  
 return 1;  
 }  
   
 string line;  
 while (getline(inFile, line)) {  
 try {  
 records.push\_back(parseLine(line));  
 }  
 catch (const exception& e) {  
 cerr << "Warning: " << e.what() << endl;  
 // Continue processing next line  
 }  
 }  
   
 inFile.close();  
   
 // Process records  
 for (auto& rec : records) {  
 rec.value \*= 1.1; // Apply 10% increase  
 }  
   
 // Write modified records to output file  
 ofstream outFile("records\_updated.txt");  
   
 if (!outFile) {  
 cerr << "Error opening output file" << endl;  
 return 1;  
 }  
   
 for (const auto& rec : records) {  
 outFile << formatRecord(rec) << endl;  
 }  
   
 outFile.close();  
   
 cout << "Successfully processed " << records.size() << " records" << endl;  
 return 0;  
}

### Platform Independence in File Handling

Different operating systems use different conventions for line endings: - Windows: \r\n (carriage return + line feed) - Unix/Linux/macOS: \n (line feed)

Modern C++ file streams handle these differences automatically when opened in text mode, but there are still some considerations:

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
// Platform-independent file path handling  
string getFilePath(const string& directory, const string& filename) {  
#ifdef \_WIN32  
 return directory + "\\" + filename;  
#else  
 return directory + "/" + filename;  
#endif  
}  
  
int main() {  
 string path = getFilePath("data", "records.txt");  
 cout << "Platform-specific path: " << path << endl;  
   
 // Open file in text mode (default)  
 ifstream textFile(path);  
   
 // Open file in binary mode (no line ending translation)  
 ifstream binaryFile(path, ios::binary);  
   
 // Check if file exists using portable code  
 ifstream checkFile(path);  
 if (!checkFile) {  
 cout << "File does not exist or cannot be opened" << endl;  
 }  
 else {  
 cout << "File exists and can be opened" << endl;  
 }  
   
 return 0;  
}

In conclusion, C++ provides powerful file handling capabilities through its stream classes. By understanding the stream hierarchy, file operations, error handling, and best practices, you can effectively work with files in your C++ applications. Remember to always check for errors, close files when you’re done with them, and use the appropriate stream types and modes for your specific requirements.

# Chapter 9: File Handling in C++ (Part 2)

## 9.3 Binary File I/O

Binary file I/O is a powerful feature in C++ that allows us to read and write data in its raw memory representation without any conversion. Unlike text files, binary files store data in the same format as it exists in memory, providing efficiency and precision for many applications.

### 9.3.1 Text vs Binary Mode

Understanding the difference between text and binary modes is crucial before diving into binary file operations:

| Feature | Text Mode | Binary Mode |
| --- | --- | --- |
| Data Representation | Characters with potential conversions | Raw bytes with no conversions |
| Line Endings | Platform-specific conversions (e.g., \n to \r\n on Windows) | No conversions |
| End-of-File Detection | Special handling of EOF character | No special handling |
| Null Characters | May cause issues (treated as string terminators) | Handled correctly as ordinary bytes |
| Use Cases | Human-readable text, simple data | Images, audio, structured data, custom formats |

### 9.3.2 Opening Files in Binary Mode

To work with binary files, you need to include the ios::binary flag when opening a file:

#include <fstream>  
using namespace std;  
  
int main() {  
 // Opening a file for binary reading  
 ifstream inFile("data.bin", ios::binary);  
   
 // Opening a file for binary writing  
 ofstream outFile("output.bin", ios::binary);  
   
 // Opening a file for both binary reading and writing  
 fstream file("data.bin", ios::binary | ios::in | ios::out);  
   
 return 0;  
}

### 9.3.3 Writing Binary Data

The primary method for binary writing is write(), which takes a pointer to memory and the number of bytes to write:

#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main() {  
 // Open a binary file for writing  
 ofstream outFile("numbers.bin", ios::binary);  
   
 if (!outFile) {  
 cerr << "Error opening file for writing!" << endl;  
 return 1;  
 }  
   
 // Writing a single integer  
 int value = 42;  
 outFile.write(reinterpret\_cast<const char\*>(&value), sizeof(value));  
   
 // Writing an array of integers  
 int numbers[] = {10, 20, 30, 40, 50};  
 outFile.write(reinterpret\_cast<const char\*>(numbers), sizeof(numbers));  
   
 // Writing floating-point data  
 double pi = 3.14159265359;  
 outFile.write(reinterpret\_cast<const char\*>(&pi), sizeof(double));  
   
 outFile.close();  
 cout << "Binary data written successfully." << endl;  
   
 return 0;  
}

Key points about binary writing:

1. **The reinterpret\_cast**: Converts any pointer type to const char\* which represents raw bytes
2. **sizeof operator**: Determines the exact number of bytes to write
3. **No formatting**: Data is written exactly as it appears in memory
4. **No automatic terminators**: Unlike text mode, no extra characters are added

### 9.3.4 Reading Binary Data

To read binary data, we use the read() method that corresponds to write():

#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main() {  
 // Open the binary file for reading  
 ifstream inFile("numbers.bin", ios::binary);  
   
 if (!inFile) {  
 cerr << "Error opening file for reading!" << endl;  
 return 1;  
 }  
   
 // Reading a single integer  
 int value;  
 inFile.read(reinterpret\_cast<char\*>(&value), sizeof(value));  
 cout << "Read integer value: " << value << endl;  
   
 // Reading an array of integers  
 int numbers[5];  
 inFile.read(reinterpret\_cast<char\*>(numbers), sizeof(numbers));  
   
 cout << "Read integer array: ";  
 for (int i = 0; i < 5; i++) {  
 cout << numbers[i] << " ";  
 }  
 cout << endl;  
   
 // Reading a double  
 double pi;  
 inFile.read(reinterpret\_cast<char\*>(&pi), sizeof(double));  
 cout << "Read double value: " << pi << endl;  
   
 // Check how many bytes were actually read  
 cout << "Last read operation read " << inFile.gcount() << " bytes." << endl;  
   
 // Check for errors  
 if (inFile.fail() && !inFile.eof()) {  
 cerr << "Error reading from file!" << endl;  
 }  
   
 inFile.close();  
 return 0;  
}

Important aspects of binary reading:

1. **Error checking**: Always verify that read operations completed successfully
2. **gcount()**: Returns the number of bytes read by the last unformatted read operation
3. **Memory allocation**: Ensure your variables or buffers are properly allocated before reading

### 9.3.5 Working with Complex Data Structures

Binary I/O is particularly useful for working with structures and classes:

#include <iostream>  
#include <fstream>  
#include <vector>  
#include <string>  
using namespace std;  
  
// A fixed-size record structure suitable for binary I/O  
struct Person {  
 char name[50]; // Fixed-size array for binary compatibility  
 int age;  
 double salary;  
};  
  
// Initialize a Person struct  
void initPerson(Person& p, const string& name, int age, double salary) {  
 // Copy name (ensuring it fits in the buffer)  
 strncpy(p.name, name.c\_str(), sizeof(p.name) - 1);  
 p.name[sizeof(p.name) - 1] = '\0'; // Ensure null termination  
   
 p.age = age;  
 p.salary = salary;  
}  
  
// Display a Person struct  
void displayPerson(const Person& p) {  
 cout << "Name: " << p.name << ", Age: " << p.age << ", Salary: $" << p.salary << endl;  
}  
  
int main() {  
 // Create sample data  
 vector<Person> people;  
   
 Person p1;  
 initPerson(p1, "Alice Johnson", 28, 75000.00);  
   
 Person p2;  
 initPerson(p2, "Bob Smith", 35, 85000.50);  
   
 Person p3;  
 initPerson(p3, "Carol Williams", 42, 95000.75);  
   
 people.push\_back(p1);  
 people.push\_back(p2);  
 people.push\_back(p3);  
   
 // Write to binary file  
 ofstream outFile("personnel.dat", ios::binary);  
   
 if (!outFile) {  
 cerr << "Failed to open file for writing!" << endl;  
 return 1;  
 }  
   
 // Write number of records first  
 int count = people.size();  
 outFile.write(reinterpret\_cast<const char\*>(&count), sizeof(count));  
   
 // Then write all records at once  
 outFile.write(reinterpret\_cast<const char\*>(people.data()),   
 people.size() \* sizeof(Person));  
   
 outFile.close();  
   
 cout << "Wrote " << count << " records to personnel.dat" << endl;  
   
 // Read from binary file  
 ifstream inFile("personnel.dat", ios::binary);  
   
 if (!inFile) {  
 cerr << "Failed to open file for reading!" << endl;  
 return 1;  
 }  
   
 // Read record count  
 int readCount;  
 inFile.read(reinterpret\_cast<char\*>(&readCount), sizeof(readCount));  
   
 cout << "Reading " << readCount << " records from file..." << endl;  
   
 // Read all records at once  
 vector<Person> readPeople(readCount);  
 inFile.read(reinterpret\_cast<char\*>(readPeople.data()),   
 readCount \* sizeof(Person));  
   
 inFile.close();  
   
 // Display the read records  
 for (const auto& person : readPeople) {  
 displayPerson(person);  
 }  
   
 return 0;  
}

Important considerations when working with structures:

1. **Fixed-size members**: For reliable binary I/O, use fixed-size data members like arrays instead of std::string
2. **Structure packing**: Be aware of structure padding that may vary across compilers
3. **Data alignment**: Different platforms may have different alignment requirements

### 9.3.6 Handling Variable-Length Data

For variable-length data such as strings, we need to use additional strategies:

#include <iostream>  
#include <fstream>  
#include <string>  
#include <vector>  
using namespace std;  
  
struct VariableLengthRecord {  
 int id;  
 string name; // Variable length  
 vector<double> scores; // Variable length  
};  
  
// Write a single string with length prefix  
void writeString(ofstream& outFile, const string& str) {  
 // Write length first  
 size\_t length = str.length();  
 outFile.write(reinterpret\_cast<const char\*>(&length), sizeof(length));  
   
 // Write string data (without null terminator)  
 outFile.write(str.c\_str(), length);  
}  
  
// Read a single string with length prefix  
string readString(ifstream& inFile) {  
 // Read string length  
 size\_t length;  
 inFile.read(reinterpret\_cast<char\*>(&length), sizeof(length));  
   
 // Read string data  
 vector<char> buffer(length);  
 inFile.read(buffer.data(), length);  
   
 // Return as string  
 return string(buffer.data(), length);  
}  
  
// Write a vector of doubles  
void writeVector(ofstream& outFile, const vector<double>& vec) {  
 // Write vector size  
 size\_t size = vec.size();  
 outFile.write(reinterpret\_cast<const char\*>(&size), sizeof(size));  
   
 // Write all elements at once  
 outFile.write(reinterpret\_cast<const char\*>(vec.data()),   
 size \* sizeof(double));  
}  
  
// Read a vector of doubles  
vector<double> readVector(ifstream& inFile) {  
 // Read vector size  
 size\_t size;  
 inFile.read(reinterpret\_cast<char\*>(&size), sizeof(size));  
   
 // Create vector and read all elements  
 vector<double> vec(size);  
 inFile.read(reinterpret\_cast<char\*>(vec.data()),   
 size \* sizeof(double));  
   
 return vec;  
}  
  
int main() {  
 // Create a test record  
 VariableLengthRecord student;  
 student.id = 12345;  
 student.name = "Jane Doe Smith";  
 student.scores = {98.5, 87.0, 92.5, 95.0};  
   
 // Write to binary file  
 ofstream outFile("student.bin", ios::binary);  
 if (!outFile) {  
 cerr << "Failed to open file for writing!" << endl;  
 return 1;  
 }  
   
 // Write the record components  
 outFile.write(reinterpret\_cast<const char\*>(&student.id), sizeof(student.id));  
 writeString(outFile, student.name);  
 writeVector(outFile, student.scores);  
   
 outFile.close();  
 cout << "Record written successfully." << endl;  
   
 // Read from binary file  
 ifstream inFile("student.bin", ios::binary);  
 if (!inFile) {  
 cerr << "Failed to open file for reading!" << endl;  
 return 1;  
 }  
   
 // Read the record components  
 VariableLengthRecord readStudent;  
 inFile.read(reinterpret\_cast<char\*>(&readStudent.id), sizeof(readStudent.id));  
 readStudent.name = readString(inFile);  
 readStudent.scores = readVector(inFile);  
   
 inFile.close();  
   
 // Display the read record  
 cout << "\nRead record:" << endl;  
 cout << "ID: " << readStudent.id << endl;  
 cout << "Name: " << readStudent.name << endl;  
 cout << "Scores: ";  
 for (double score : readStudent.scores) {  
 cout << score << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

Strategies for variable-length data:

1. **Length prefixing**: Store the length before the actual data
2. **Serialization**: Convert complex objects to a sequence of bytes
3. **Delimiting**: Use special markers to separate variable-length fields (less common)

### 9.3.7 Portability Considerations

Binary files face several portability issues across different platforms:

1. **Endianness**: Different systems store multi-byte values in different byte orders
   * Little-endian: least significant byte first (x86, x64)
   * Big-endian: most significant byte first (some network protocols, older systems)
2. **Data Alignment**: Different platforms may insert different padding between structure members
3. **Type Sizes**: The size of types like int and long can vary across platforms

Here’s how to handle these issues:

#include <iostream>  
#include <fstream>  
#include <cstdint> // For fixed-width integer types  
using namespace std;  
  
// Check system endianness  
bool isLittleEndian() {  
 uint16\_t value = 0x0001;  
 return \*reinterpret\_cast<uint8\_t\*>(&value) == 0x01;  
}  
  
// Write a 32-bit integer in portable (little-endian) format  
void writeInt32(ofstream& outFile, int32\_t value) {  
 uint8\_t bytes[4];  
 bytes[0] = value & 0xFF;  
 bytes[1] = (value >> 8) & 0xFF;  
 bytes[2] = (value >> 16) & 0xFF;  
 bytes[3] = (value >> 24) & 0xFF;  
   
 outFile.write(reinterpret\_cast<const char\*>(bytes), 4);  
}  
  
// Read a 32-bit integer in portable (little-endian) format  
int32\_t readInt32(ifstream& inFile) {  
 uint8\_t bytes[4];  
 inFile.read(reinterpret\_cast<char\*>(bytes), 4);  
   
 return bytes[0] |   
 (bytes[1] << 8) |   
 (bytes[2] << 16) |   
 (bytes[3] << 24);  
}  
  
// Write a 32-bit floating point in portable format  
void writeFloat(ofstream& outFile, float value) {  
 // This approach works but is still platform-dependent for the float representation  
 uint8\_t\* bytes = reinterpret\_cast<uint8\_t\*>(&value);  
   
 // For complete portability, we would need to handle the IEEE-754 format explicitly  
 outFile.write(reinterpret\_cast<const char\*>(bytes), sizeof(float));  
}  
  
// A portable record structure with explicit field sizes  
#pragma pack(push, 1) // Disable padding  
struct PortableRecord {  
 int32\_t id; // Always 32 bits  
 uint16\_t age; // Always 16 bits  
 uint8\_t flags; // Always 8 bits  
};  
#pragma pack(pop) // Restore default padding  
  
int main() {  
 cout << "System is " << (isLittleEndian() ? "little" : "big") << "-endian." << endl;  
   
 // Write portable binary data  
 ofstream outFile("portable.bin", ios::binary);  
 if (!outFile) {  
 cerr << "Failed to open file for writing!" << endl;  
 return 1;  
 }  
   
 // Write integers in a portable format  
 writeInt32(outFile, 12345);  
 writeInt32(outFile, -67890);  
   
 // Write a float value  
 float pi = 3.14159f;  
 writeFloat(outFile, pi);  
   
 // Write a portable structure  
 PortableRecord rec = {1001, 25, 0x03};  
 outFile.write(reinterpret\_cast<const char\*>(&rec), sizeof(rec));  
   
 outFile.close();  
 cout << "Portable data written successfully." << endl;  
   
 // Read the portable data back  
 ifstream inFile("portable.bin", ios::binary);  
 if (!inFile) {  
 cerr << "Failed to open file for reading!" << endl;  
 return 1;  
 }  
   
 // Read the integers  
 int32\_t val1 = readInt32(inFile);  
 int32\_t val2 = readInt32(inFile);  
   
 cout << "Read integers: " << val1 << ", " << val2 << endl;  
   
 // Read the float value  
 float readPi;  
 inFile.read(reinterpret\_cast<char\*>(&readPi), sizeof(float));  
 cout << "Read float: " << readPi << endl;  
   
 // Read the structure  
 PortableRecord readRec;  
 inFile.read(reinterpret\_cast<char\*>(&readRec), sizeof(readRec));  
   
 cout << "Read record: id=" << readRec.id   
 << ", age=" << readRec.age   
 << ", flags=0x" << hex << static\_cast<int>(readRec.flags) << dec << endl;  
   
 inFile.close();  
   
 return 0;  
}

Best practices for portable binary files:

1. **Use fixed-width types** like int32\_t instead of int
2. **Handle endianness explicitly** when necessary
3. **Minimize or control structure padding** using pragmas or careful member ordering
4. **Include format versions** in file headers to help with future compatibility
5. **Consider using standardized binary formats** (like Protocol Buffers or MessagePack) for critical applications

### 9.3.8 Binary I/O for Image Files

A practical application of binary I/O is working with image files. Here’s an example with BMP files:

#include <iostream>  
#include <fstream>  
#include <vector>  
using namespace std;  
  
#pragma pack(push, 1) // Disable padding for these structures  
  
// BMP file header (14 bytes)  
struct BMPFileHeader {  
 uint16\_t signature; // 'BM' signature (0x4D42)  
 uint32\_t fileSize; // Size of the entire file  
 uint16\_t reserved1; // Reserved (0)  
 uint16\_t reserved2; // Reserved (0)  
 uint32\_t dataOffset; // Offset to the start of pixel data  
};  
  
// BMP info header (40 bytes)  
struct BMPInfoHeader {  
 uint32\_t headerSize; // Size of this header (40 bytes)  
 int32\_t width; // Width in pixels  
 int32\_t height; // Height in pixels  
 uint16\_t planes; // Number of color planes (1)  
 uint16\_t bitsPerPixel; // Bits per pixel (e.g., 24 for RGB)  
 uint32\_t compression; // Compression method (0 for none)  
 uint32\_t imageSize; // Size of raw image data  
 int32\_t xPixelsPerMeter; // Horizontal resolution  
 int32\_t yPixelsPerMeter; // Vertical resolution  
 uint32\_t colorsUsed; // Number of colors in palette  
 uint32\_t colorsImportant; // Important colors  
};  
  
#pragma pack(pop)  
  
// RGB color structure  
struct Pixel {  
 uint8\_t blue;  
 uint8\_t green;  
 uint8\_t red;  
};  
  
// Create a simple BMP image with a gradient  
void createBMPImage(const string& filename, int width, int height) {  
 // Calculate row size (must be padded to 4-byte boundary)  
 int rowSize = ((width \* 3) + 3) & ~3;  
 int imageSize = rowSize \* height;  
   
 // Set up file header  
 BMPFileHeader fileHeader;  
 fileHeader.signature = 0x4D42; // 'BM' in little endian  
 fileHeader.fileSize = sizeof(BMPFileHeader) + sizeof(BMPInfoHeader) + imageSize;  
 fileHeader.reserved1 = 0;  
 fileHeader.reserved2 = 0;  
 fileHeader.dataOffset = sizeof(BMPFileHeader) + sizeof(BMPInfoHeader);  
   
 // Set up info header  
 BMPInfoHeader infoHeader;  
 infoHeader.headerSize = sizeof(BMPInfoHeader);  
 infoHeader.width = width;  
 infoHeader.height = height; // Positive for bottom-up image  
 infoHeader.planes = 1;  
 infoHeader.bitsPerPixel = 24; // RGB color (3 bytes per pixel)  
 infoHeader.compression = 0; // No compression  
 infoHeader.imageSize = imageSize;  
 infoHeader.xPixelsPerMeter = 2835; // 72 DPI  
 infoHeader.yPixelsPerMeter = 2835; // 72 DPI  
 infoHeader.colorsUsed = 0;  
 infoHeader.colorsImportant = 0;  
   
 // Create output file  
 ofstream outFile(filename, ios::binary);  
 if (!outFile) {  
 cerr << "Failed to create BMP file!" << endl;  
 return;  
 }  
   
 // Write headers  
 outFile.write(reinterpret\_cast<const char\*>(&fileHeader), sizeof(fileHeader));  
 outFile.write(reinterpret\_cast<const char\*>(&infoHeader), sizeof(infoHeader));  
   
 // Create and write pixel data  
 // Note: BMP stores images bottom-up (last row first)  
 vector<uint8\_t> rowData(rowSize, 0); // Initialize with zeros (for padding)  
   
 for (int y = height - 1; y >= 0; y--) { // Start from the bottom row  
 for (int x = 0; x < width; x++) {  
 // Create a gradient pattern  
 Pixel pixel;  
 pixel.red = static\_cast<uint8\_t>(255 \* x / width);  
 pixel.green = static\_cast<uint8\_t>(255 \* y / height);  
 pixel.blue = static\_cast<uint8\_t>(255 - ((pixel.red + pixel.green) / 2));  
   
 // Calculate position in the row buffer  
 int pos = x \* 3; // 3 bytes per pixel  
   
 // Store pixel in row buffer  
 rowData[pos] = pixel.blue;  
 rowData[pos + 1] = pixel.green;  
 rowData[pos + 2] = pixel.red;  
 }  
   
 // Write the entire row  
 outFile.write(reinterpret\_cast<const char\*>(rowData.data()), rowSize);  
 }  
   
 outFile.close();  
 cout << "Created " << width << "x" << height << " BMP image: " << filename << endl;  
}  
  
// Read and display information about a BMP file  
void readBMPInfo(const string& filename) {  
 ifstream inFile(filename, ios::binary);  
 if (!inFile) {  
 cerr << "Failed to open BMP file!" << endl;  
 return;  
 }  
   
 // Read headers  
 BMPFileHeader fileHeader;  
 BMPInfoHeader infoHeader;  
   
 inFile.read(reinterpret\_cast<char\*>(&fileHeader), sizeof(fileHeader));  
 inFile.read(reinterpret\_cast<char\*>(&infoHeader), sizeof(infoHeader));  
   
 // Verify it's a valid BMP file  
 if (fileHeader.signature != 0x4D42) {  
 cerr << "Not a valid BMP file!" << endl;  
 return;  
 }  
   
 // Display file information  
 cout << "BMP File Information:" << endl;  
 cout << "-----------------" << endl;  
 cout << "File size: " << fileHeader.fileSize << " bytes" << endl;  
 cout << "Image dimensions: " << infoHeader.width << "x" << infoHeader.height << " pixels" << endl;  
 cout << "Bits per pixel: " << infoHeader.bitsPerPixel << endl;  
 cout << "Compression: " << (infoHeader.compression == 0 ? "None" : "Yes") << endl;  
 cout << "Image size: " << infoHeader.imageSize << " bytes" << endl;  
   
 // Read the first few pixels (from bottom-left corner)  
 inFile.seekg(fileHeader.dataOffset);  
   
 // Read first pixel's color  
 Pixel pixel;  
 inFile.read(reinterpret\_cast<char\*>(&pixel), sizeof(pixel));  
 cout << "First pixel color (RGB): ("   
 << static\_cast<int>(pixel.red) << ","   
 << static\_cast<int>(pixel.green) << ","   
 << static\_cast<int>(pixel.blue) << ")" << endl;  
   
 inFile.close();  
}  
  
int main() {  
 // Create a small gradient BMP image  
 createBMPImage("gradient.bmp", 256, 256);  
   
 // Read back the image information  
 readBMPInfo("gradient.bmp");  
   
 return 0;  
}

This example demonstrates: 1. Working with complex binary file formats 2. Handling headers and pixel data 3. Managing padding and alignment 4. Proper organization of read and write operations

### 9.3.9 Summary and Best Practices

#### Key Points About Binary File I/O:

1. **Efficiency**: Binary I/O is generally faster and more space-efficient than text I/O
2. **Precision**: No loss of precision due to text conversions
3. **Structure Preservation**: Maintains the exact structure of data in memory
4. **Complexity**: Requires more careful handling of data types and portability

#### Best Practices:

1. **Always check operations**: Verify that file operations succeed before proceeding
2. **Handle endianness**: Be aware of byte-order issues when portability matters
3. **Use fixed-width types**: Use int32\_t, uint64\_t, etc. when file format stability is important
4. **Structure packing**: Use packing directives when needed to control structure layout
5. **Include metadata**: File headers with version information help with compatibility
6. **Error handling**: Implement robust error checking and recovery
7. **Buffer management**: Ensure sufficient buffer space when reading variable-length data
8. **File format documentation**: Document your binary file formats thoroughly

Binary file I/O is essential for many applications including: - Database systems - Game file formats - Multimedia applications - Scientific data processing - Performance-critical applications - Applications working with proprietary data formats

By mastering binary file I/O, you gain precise control over how data is stored and retrieved from files, enabling efficient and flexible data management in your C++ applications.

# Chapter 9: File Handling in C++ (Part 3)

## 9.4 File Pointers and Random Access

While sequential file access is suitable for many applications, C++ also provides powerful random access capabilities that allow you to move directly to specific positions within a file without having to read through all the preceding data. This is especially useful when working with large files or database-like applications.

### 9.4.1 Understanding File Position Pointers

Each file stream object in C++ maintains internal pointers that track the current position for read and write operations:

1. **Get Pointer (g)**: Determines where the next read operation will occur
2. **Put Pointer (p)**: Determines where the next write operation will occur

For different stream types: - ifstream objects only use the get pointer - ofstream objects only use the put pointer - fstream objects use both pointers

These pointers are measured in bytes from the beginning of the file, with position 0 representing the start of the file.

### 9.4.2 File Position Functions

C++ provides four main functions for manipulating file positions:

1. **tellg()**: Returns the current position of the get (input) pointer
2. **seekg(pos, dir)**: Moves the get pointer to a specified position
3. **tellp()**: Returns the current position of the put (output) pointer
4. **seekp(pos, dir)**: Moves the put pointer to a specified position

#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main() {  
 // Create a sample text file  
 ofstream outFile("position\_test.txt");  
 outFile << "ABCDEFGHIJKLMNOPQRSTUVWXYZ"; // 26 characters  
 outFile.close();  
   
 // Open the file for both reading and writing  
 fstream file("position\_test.txt", ios::in | ios::out);  
   
 if (!file) {  
 cerr << "Error opening file!" << endl;  
 return 1;  
 }  
   
 // Get initial position  
 streampos initialPos = file.tellg(); // Should be 0  
 cout << "Initial position: " << initialPos << endl;  
   
 // Read the first character  
 char c;  
 file.get(c);  
 cout << "First character: " << c << endl;  
   
 // Get position after reading one character  
 streampos newPos = file.tellg();  
 cout << "Position after reading one character: " << newPos << endl;  
   
 // Close the file  
 file.close();  
   
 return 0;  
}

### 9.4.3 Moving Within a File Using seekg() and seekp()

The seekg() and seekp() functions take two parameters: 1. An offset value (number of bytes) 2. A seek direction flag

The seek direction flags are: - ios::beg: Position relative to the beginning of the file - ios::cur: Position relative to the current position - ios::end: Position relative to the end of the file

#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main() {  
 // Create a sample file  
 ofstream outFile("seek\_test.txt");  
 outFile << "Line 1: This is the first line." << endl;  
 outFile << "Line 2: This is the second line." << endl;  
 outFile << "Line 3: This is the third line." << endl;  
 outFile.close();  
   
 // Open for reading  
 fstream file("seek\_test.txt", ios::in | ios::out);  
   
 if (!file) {  
 cerr << "Error opening file!" << endl;  
 return 1;  
 }  
   
 // Method 1: Seek from beginning of file  
 file.seekg(0, ios::beg); // Go to the beginning  
 char c1;  
 file.get(c1);  
 cout << "First character: " << c1 << endl;  
   
 // Method 2: Seek from current position  
 file.seekg(7, ios::cur); // Skip ahead 7 bytes  
 char c2;  
 file.get(c2);  
 cout << "Character 8 bytes in: " << c2 << endl;  
   
 // Method 3: Seek from end of file  
 file.seekg(-10, ios::end); // Go to 10 bytes before the end  
 char c3;  
 file.get(c3);  
 cout << "Character 10 bytes before end: " << c3 << endl;  
   
 // Absolute positioning  
 file.seekg(10); // Same as seekg(10, ios::beg)  
 char c4;  
 file.get(c4);  
 cout << "Character at position 10: " << c4 << endl;  
   
 file.close();  
   
 return 0;  
}

### 9.4.4 Working with Text Files and Line Positions

When working with text files, it’s often useful to find the starting positions of specific lines:

#include <iostream>  
#include <fstream>  
#include <string>  
#include <vector>  
using namespace std;  
  
int main() {  
 // Create a sample text file with multiple lines  
 ofstream outFile("lines.txt");  
 outFile << "Line 1: First line of text." << endl;  
 outFile << "Line 2: Second line of text." << endl;  
 outFile << "Line 3: Third line of text." << endl;  
 outFile << "Line 4: Fourth line of text." << endl;  
 outFile << "Line 5: Fifth line of text." << endl;  
 outFile.close();  
   
 // Open the file for reading  
 ifstream inFile("lines.txt");  
   
 if (!inFile) {  
 cerr << "Error opening file!" << endl;  
 return 1;  
 }  
   
 // Store the positions of the beginning of each line  
 vector<streampos> linePositions;  
 linePositions.push\_back(inFile.tellg()); // Start of file is first line  
   
 string line;  
 while (getline(inFile, line)) {  
 linePositions.push\_back(inFile.tellg()); // Position after each line  
 }  
   
 // We've reached the end of file  
 cout << "Found " << linePositions.size() - 1 << " lines in file." << endl;  
   
 // Now we can jump directly to any line  
 inFile.clear(); // Clear EOF flag  
   
 // Jump to the beginning of line 3 (index 2)  
 int targetLine = 3;  
 if (targetLine > 0 && targetLine < linePositions.size()) {  
 inFile.seekg(linePositions[targetLine - 1]);  
 getline(inFile, line);  
 cout << "Line " << targetLine << ": " << line << endl;  
 }  
   
 // Jump to the beginning of line 5 (index 4)  
 targetLine = 5;  
 if (targetLine > 0 && targetLine < linePositions.size()) {  
 inFile.seekg(linePositions[targetLine - 1]);  
 getline(inFile, line);  
 cout << "Line " << targetLine << ": " << line << endl;  
 }  
   
 inFile.close();  
   
 return 0;  
}

### 9.4.5 Random Access in Binary Files

Random access is particularly useful with binary files, especially when working with fixed-size records:

#include <iostream>  
#include <fstream>  
#include <string>  
#include <vector>  
using namespace std;  
  
// Define a fixed-size record structure  
struct Employee {  
 int id;  
 char name[30]; // Fixed-size for consistent record sizes  
 double salary;  
};  
  
// Initialize an employee record  
void initEmployee(Employee& emp, int id, const string& name, double salary) {  
 emp.id = id;  
   
 // Copy name (ensuring it fits in the fixed-size buffer)  
 strncpy(emp.name, name.c\_str(), sizeof(emp.name) - 1);  
 emp.name[sizeof(emp.name) - 1] = '\0'; // Ensure null termination  
   
 emp.salary = salary;  
}  
  
// Display employee record  
void displayEmployee(const Employee& emp) {  
 cout << "ID: " << emp.id << ", Name: " << emp.name   
 << ", Salary: $" << emp.salary << endl;  
}  
  
int main() {  
 // Create a binary file with employee records  
 ofstream outFile("employees.dat", ios::binary);  
   
 if (!outFile) {  
 cerr << "Error creating file!" << endl;  
 return 1;  
 }  
   
 // Create some employee records  
 const int NUM\_EMPLOYEES = 5;  
 vector<Employee> employees(NUM\_EMPLOYEES);  
   
 initEmployee(employees[0], 1001, "John Smith", 45000.0);  
 initEmployee(employees[1], 1002, "Mary Johnson", 52000.0);  
 initEmployee(employees[2], 1003, "James Williams", 48500.0);  
 initEmployee(employees[3], 1004, "Patricia Brown", 51000.0);  
 initEmployee(employees[4], 1005, "Robert Jones", 49500.0);  
   
 // Write all records to file  
 outFile.write(reinterpret\_cast<const char\*>(employees.data()),  
 NUM\_EMPLOYEES \* sizeof(Employee));  
   
 outFile.close();  
   
 // Open file for random access  
 fstream file("employees.dat", ios::in | ios::out | ios::binary);  
   
 if (!file) {  
 cerr << "Error opening file for random access!" << endl;  
 return 1;  
 }  
   
 // 1. Access a specific record directly (record 3, index 2)  
 int recordToAccess = 2; // 0-based index  
 file.seekg(recordToAccess \* sizeof(Employee));  
   
 Employee emp;  
 file.read(reinterpret\_cast<char\*>(&emp), sizeof(Employee));  
   
 cout << "Employee record at index " << recordToAccess << ":" << endl;  
 displayEmployee(emp);  
   
 // 2. Modify a record (give employee 1002 a raise)  
 int recordToModify = 1; // Mary Johnson (index 1)  
   
 // First read the record  
 file.seekg(recordToModify \* sizeof(Employee));  
 file.read(reinterpret\_cast<char\*>(&emp), sizeof(Employee));  
   
 // Modify the record in memory  
 emp.salary \*= 1.10; // 10% raise  
   
 // Write back to the same position  
 file.seekp(recordToModify \* sizeof(Employee));  
 file.write(reinterpret\_cast<const char\*>(&emp), sizeof(Employee));  
   
 // 3. Read all records to verify changes  
 file.seekg(0, ios::beg); // Go back to the beginning  
   
 cout << "\nAll employee records:" << endl;  
 for (int i = 0; i < NUM\_EMPLOYEES; i++) {  
 file.read(reinterpret\_cast<char\*>(&emp), sizeof(Employee));  
 cout << i << ": ";  
 displayEmployee(emp);  
 }  
   
 // 4. Find the file size  
 file.seekg(0, ios::end);  
 streampos fileSize = file.tellg();  
   
 cout << "\nFile size: " << fileSize << " bytes" << endl;  
 cout << "Number of records: " << (fileSize / sizeof(Employee)) << endl;  
   
 file.close();  
   
 return 0;  
}

### 9.4.6 Implementing a Simple Record-Based Database

Random access file handling is perfect for implementing a simple database system:

#include <iostream>  
#include <fstream>  
#include <string>  
#include <vector>  
#include <map>  
using namespace std;  
  
// Define a fixed-size record structure for a product inventory  
struct Product {  
 int id;  
 char name[50];  
 char category[30];  
 double price;  
 int quantity;  
 bool active; // Flag to mark if product is active or deleted  
};  
  
class InventoryManager {  
private:  
 string filename;  
 fstream file;  
 map<int, streampos> idToPosition; // Maps product ID to file position  
   
 // Helper to initialize a product  
 void initProduct(Product& product, int id, const string& name, const string& category,  
 double price, int quantity, bool active = true) {  
 product.id = id;  
   
 strncpy(product.name, name.c\_str(), sizeof(product.name) - 1);  
 product.name[sizeof(product.name) - 1] = '\0';  
   
 strncpy(product.category, category.c\_str(), sizeof(product.category) - 1);  
 product.category[sizeof(product.category) - 1] = '\0';  
   
 product.price = price;  
 product.quantity = quantity;  
 product.active = active;  
 }  
   
 // Build the ID-to-position map by scanning the file  
 void buildIndex() {  
 idToPosition.clear();  
   
 // Start from beginning of file  
 file.seekg(0, ios::beg);  
   
 Product product;  
 streampos position = 0;  
   
 // Read each record  
 while (file.read(reinterpret\_cast<char\*>(&product), sizeof(Product))) {  
 if (product.active) { // Only index active records  
 idToPosition[product.id] = position;  
 }  
 position += sizeof(Product);  
 }  
   
 // Clear any error flags (like EOF)  
 file.clear();  
 }  
   
public:  
 InventoryManager(const string& fname) : filename(fname) {  
 // Open the file for binary reading and writing  
 file.open(filename, ios::binary | ios::in | ios::out);  
   
 if (!file) {  
 // File doesn't exist, create it  
 file.clear();  
 file.open(filename, ios::binary | ios::out);  
 file.close();  
 file.open(filename, ios::binary | ios::in | ios::out);  
 }  
   
 buildIndex();  
 }  
   
 ~InventoryManager() {  
 if (file.is\_open()) {  
 file.close();  
 }  
 }  
   
 // Add a new product  
 bool addProduct(int id, const string& name, const string& category,  
 double price, int quantity) {  
 // Check if ID already exists  
 if (idToPosition.find(id) != idToPosition.end()) {  
 cerr << "Product with ID " << id << " already exists!" << endl;  
 return false;  
 }  
   
 // Create product record  
 Product product;  
 initProduct(product, id, name, category, price, quantity);  
   
 // Go to end of file  
 file.seekp(0, ios::end);  
 streampos position = file.tellp();  
   
 // Write the record  
 file.write(reinterpret\_cast<const char\*>(&product), sizeof(Product));  
   
 // Update index if write was successful  
 if (file.good()) {  
 idToPosition[id] = position;  
 return true;  
 }  
   
 return false;  
 }  
   
 // Get a product by ID  
 bool getProduct(int id, Product& product) {  
 // Check if product exists  
 auto it = idToPosition.find(id);  
 if (it == idToPosition.end()) {  
 return false;  
 }  
   
 // Go to the product's position  
 file.seekg(it->second);  
   
 // Read the product  
 file.read(reinterpret\_cast<char\*>(&product), sizeof(Product));  
   
 return file.good();  
 }  
   
 // Update an existing product  
 bool updateProduct(int id, const string& name, const string& category,  
 double price, int quantity) {  
 // Check if product exists  
 auto it = idToPosition.find(id);  
 if (it == idToPosition.end()) {  
 cerr << "Product with ID " << id << " not found!" << endl;  
 return false;  
 }  
   
 // Go to the product's position  
 file.seekp(it->second);  
   
 // Create updated product record  
 Product product;  
 initProduct(product, id, name, category, price, quantity);  
   
 // Write the updated record  
 file.write(reinterpret\_cast<const char\*>(&product), sizeof(Product));  
   
 return file.good();  
 }  
   
 // Delete a product (mark as inactive)  
 bool deleteProduct(int id) {  
 // Check if product exists  
 auto it = idToPosition.find(id);  
 if (it == idToPosition.end()) {  
 cerr << "Product with ID " << id << " not found!" << endl;  
 return false;  
 }  
   
 // Go to the product's position  
 file.seekg(it->second);  
   
 // Read the product  
 Product product;  
 file.read(reinterpret\_cast<char\*>(&product), sizeof(Product));  
   
 // Mark as inactive  
 product.active = false;  
   
 // Go back to the same position  
 file.seekp(it->second);  
   
 // Write the updated record  
 file.write(reinterpret\_cast<const char\*>(&product), sizeof(Product));  
   
 // Remove from index if write was successful  
 if (file.good()) {  
 idToPosition.erase(it);  
 return true;  
 }  
   
 return false;  
 }  
   
 // List all active products  
 vector<Product> getAllProducts() {  
 vector<Product> products;  
 Product product;  
   
 for (const auto& entry : idToPosition) {  
 if (getProduct(entry.first, product)) {  
 products.push\_back(product);  
 }  
 }  
   
 return products;  
 }  
   
 // Get the count of active products  
 int getProductCount() {  
 return idToPosition.size();  
 }  
   
 // Rebuild the index (useful if file was modified externally)  
 void rebuildIndex() {  
 buildIndex();  
 }  
   
 // List products in a specific category  
 vector<Product> getProductsByCategory(const string& category) {  
 vector<Product> products;  
 Product product;  
   
 for (const auto& entry : idToPosition) {  
 if (getProduct(entry.first, product)) {  
 if (strcmp(product.category, category.c\_str()) == 0) {  
 products.push\_back(product);  
 }  
 }  
 }  
   
 return products;  
 }  
   
 // Update product quantity (for inventory management)  
 bool updateProductQuantity(int id, int newQuantity) {  
 // Check if product exists  
 auto it = idToPosition.find(id);  
 if (it == idToPosition.end()) {  
 return false;  
 }  
   
 // Read the product  
 file.seekg(it->second);  
 Product product;  
 file.read(reinterpret\_cast<char\*>(&product), sizeof(Product));  
   
 // Update quantity  
 product.quantity = newQuantity;  
   
 // Write back  
 file.seekp(it->second);  
 file.write(reinterpret\_cast<const char\*>(&product), sizeof(Product));  
   
 return file.good();  
 }  
};  
  
int main() {  
 // Create an inventory manager  
 InventoryManager inventory("inventory.dat");  
   
 // Add some products  
 cout << "Adding products..." << endl;  
 inventory.addProduct(1001, "Laptop", "Electronics", 999.99, 10);  
 inventory.addProduct(1002, "Smartphone", "Electronics", 599.99, 20);  
 inventory.addProduct(1003, "Desk", "Furniture", 249.99, 5);  
 inventory.addProduct(1004, "Chair", "Furniture", 149.99, 12);  
 inventory.addProduct(1005, "Headphones", "Electronics", 89.99, 30);  
   
 // Get the count of products  
 cout << "Total products: " << inventory.getProductCount() << endl;  
   
 // Retrieve and display a product  
 Product product;  
 if (inventory.getProduct(1002, product)) {  
 cout << "\nProduct details:" << endl;  
 cout << "ID: " << product.id << endl;  
 cout << "Name: " << product.name << endl;  
 cout << "Category: " << product.category << endl;  
 cout << "Price: $" << product.price << endl;  
 cout << "Quantity: " << product.quantity << endl;  
 }  
   
 // Update a product  
 cout << "\nUpdating product 1003..." << endl;  
 inventory.updateProduct(1003, "Office Desk", "Furniture", 279.99, 8);  
   
 // Delete a product  
 cout << "Deleting product 1005..." << endl;  
 inventory.deleteProduct(1005);  
   
 // Get all remaining products  
 cout << "\nAll products:" << endl;  
 cout << "------------" << endl;  
   
 for (const auto& p : inventory.getAllProducts()) {  
 cout << p.id << ": " << p.name << " - $" << p.price   
 << " (" << p.quantity << " in stock)" << endl;  
 }  
   
 // Get products by category  
 cout << "\nElectronics products:" << endl;  
 cout << "-------------------" << endl;  
   
 for (const auto& p : inventory.getProductsByCategory("Electronics")) {  
 cout << p.id << ": " << p.name << " - $" << p.price   
 << " (" << p.quantity << " in stock)" << endl;  
 }  
   
 return 0;  
}

### 9.4.7 Handling Variable-Length Records with Indexes

When your records are of variable length (such as text entries with varying content), you can implement an index system:

#include <iostream>  
#include <fstream>  
#include <string>  
#include <vector>  
#include <map>  
using namespace std;  
  
// Structure to track record location in the data file  
struct RecordLocation {  
 streampos position;  
 size\_t length;  
};  
  
class NoteManager {  
private:  
 string dataFilename; // Stores the actual note data  
 string indexFilename; // Stores the note index information  
 fstream dataFile;  
 fstream indexFile;  
 map<int, RecordLocation> noteIndex; // Maps note ID to its location  
   
 // Next available ID to assign  
 int nextId = 1;  
   
 // Load the index from the index file  
 void loadIndex() {  
 noteIndex.clear();  
   
 indexFile.seekg(0, ios::beg);  
   
 // First read the next available ID  
 indexFile.read(reinterpret\_cast<char\*>(&nextId), sizeof(nextId));  
   
 // Then read how many entries are in the index  
 int count;  
 indexFile.read(reinterpret\_cast<char\*>(&count), sizeof(count));  
   
 // Read each index entry  
 for (int i = 0; i < count; i++) {  
 int id;  
 RecordLocation loc;  
   
 indexFile.read(reinterpret\_cast<char\*>(&id), sizeof(id));  
 indexFile.read(reinterpret\_cast<char\*>(&loc), sizeof(loc));  
   
 if (indexFile.good()) {  
 noteIndex[id] = loc;  
 }  
 }  
 }  
   
 // Save the index to the index file  
 void saveIndex() {  
 indexFile.seekp(0, ios::beg);  
   
 // First write the next available ID  
 indexFile.write(reinterpret\_cast<const char\*>(&nextId), sizeof(nextId));  
   
 // Then write how many entries are in the index  
 int count = noteIndex.size();  
 indexFile.write(reinterpret\_cast<const char\*>(&count), sizeof(count));  
   
 // Write each index entry  
 for (const auto& entry : noteIndex) {  
 int id = entry.first;  
 const RecordLocation& loc = entry.second;  
   
 indexFile.write(reinterpret\_cast<const char\*>(&id), sizeof(id));  
 indexFile.write(reinterpret\_cast<const char\*>(&loc), sizeof(loc));  
 }  
 }  
   
public:  
 NoteManager(const string& dataFilename, const string& indexFilename)  
 : dataFilename(dataFilename), indexFilename(indexFilename) {  
   
 // Open the data file  
 dataFile.open(dataFilename, ios::binary | ios::in | ios::out | ios::app);  
 if (!dataFile) {  
 // Create it if it doesn't exist  
 dataFile.clear();  
 dataFile.open(dataFilename, ios::binary | ios::out);  
 dataFile.close();  
 dataFile.open(dataFilename, ios::binary | ios::in | ios::out | ios::app);  
 }  
   
 // Open the index file  
 indexFile.open(indexFilename, ios::binary | ios::in | ios::out);  
 if (!indexFile) {  
 // Create it if it doesn't exist  
 indexFile.clear();  
 indexFile.open(indexFilename, ios::binary | ios::out);  
   
 // Initialize with default values  
 nextId = 1;  
 int count = 0;  
   
 indexFile.write(reinterpret\_cast<const char\*>(&nextId), sizeof(nextId));  
 indexFile.write(reinterpret\_cast<const char\*>(&count), sizeof(count));  
   
 indexFile.close();  
 indexFile.open(indexFilename, ios::binary | ios::in | ios::out);  
 }  
   
 loadIndex();  
 }  
   
 ~NoteManager() {  
 if (dataFile.is\_open()) {  
 dataFile.close();  
 }  
   
 if (indexFile.is\_open()) {  
 saveIndex();  
 indexFile.close();  
 }  
 }  
   
 // Add a new note, returns the assigned ID  
 int addNote(const string& content) {  
 // Assign a new ID  
 int id = nextId++;  
   
 // Go to the end of data file  
 dataFile.seekp(0, ios::end);  
 streampos position = dataFile.tellp();  
   
 // Write the content length followed by the content  
 size\_t length = content.length();  
 dataFile.write(reinterpret\_cast<const char\*>(&length), sizeof(length));  
 dataFile.write(content.c\_str(), length);  
   
 // Add to the index  
 RecordLocation loc = {position, sizeof(length) + length};  
 noteIndex[id] = loc;  
   
 // Save the updated index  
 saveIndex();  
   
 return id;  
 }  
   
 // Get a note by ID  
 bool getNote(int id, string& content) {  
 // Check if the note exists  
 auto it = noteIndex.find(id);  
 if (it == noteIndex.end()) {  
 return false;  
 }  
   
 // Go to the note's position  
 dataFile.seekg(it->second.position);  
   
 // Read the content length  
 size\_t length;  
 dataFile.read(reinterpret\_cast<char\*>(&length), sizeof(length));  
   
 // Read the content  
 content.resize(length);  
 dataFile.read(&content[0], length);  
   
 return dataFile.good();  
 }  
   
 // Update an existing note  
 bool updateNote(int id, const string& newContent) {  
 // First check if the note exists  
 if (noteIndex.find(id) == noteIndex.end()) {  
 return false;  
 }  
   
 // For simplicity, we'll just add the updated note at the end  
 // and update the index (real systems might have garbage collection)  
 dataFile.seekp(0, ios::end);  
 streampos position = dataFile.tellp();  
   
 // Write the content length followed by the content  
 size\_t length = newContent.length();  
 dataFile.write(reinterpret\_cast<const char\*>(&length), sizeof(length));  
 dataFile.write(newContent.c\_str(), length);  
   
 // Update the index  
 RecordLocation loc = {position, sizeof(length) + length};  
 noteIndex[id] = loc;  
   
 // Save the updated index  
 saveIndex();  
   
 return true;  
 }  
   
 // Delete a note  
 bool deleteNote(int id) {  
 // Check if the note exists  
 auto it = noteIndex.find(id);  
 if (it == noteIndex.end()) {  
 return false;  
 }  
   
 // Remove from the index (note data remains in file but becomes inaccessible)  
 noteIndex.erase(it);  
   
 // Save the updated index  
 saveIndex();  
   
 return true;  
 }  
   
 // List all note IDs  
 vector<int> getAllNoteIds() {  
 vector<int> ids;  
 for (const auto& entry : noteIndex) {  
 ids.push\_back(entry.first);  
 }  
 return ids;  
 }  
   
 // Get the count of notes  
 int getNoteCount() {  
 return noteIndex.size();  
 }  
};  
  
int main() {  
 // Create a note manager  
 NoteManager notes("notes.dat", "notes.idx");  
   
 // Add some notes  
 int id1 = notes.addNote("This is the first note. It contains some important information.");  
 int id2 = notes.addNote("A second, shorter note.");  
 int id3 = notes.addNote("The third note is quite a bit longer than the others. It contains multiple sentences and spans what would be multiple lines if displayed in a text editor. This demonstrates how variable-length records are handled.");  
   
 cout << "Added notes with IDs: " << id1 << ", " << id2 << ", " << id3 << endl;  
 cout << "Total notes: " << notes.getNoteCount() << endl;  
   
 // Retrieve and display notes  
 string content;  
   
 cout << "\nNote " << id1 << ":" << endl;  
 if (notes.getNote(id1, content)) {  
 cout << content << endl;  
 }  
   
 cout << "\nNote " << id2 << ":" << endl;  
 if (notes.getNote(id2, content)) {  
 cout << content << endl;  
 }  
   
 cout << "\nNote " << id3 << ":" << endl;  
 if (notes.getNote(id3, content)) {  
 cout << content << endl;  
 }  
   
 // Update a note  
 cout << "\nUpdating note " << id2 << "..." << endl;  
 notes.updateNote(id2, "This note has been updated with new content!");  
   
 // Retrieve the updated note  
 if (notes.getNote(id2, content)) {  
 cout << "Updated content: " << content << endl;  
 }  
   
 // Delete a note  
 cout << "\nDeleting note " << id1 << "..." << endl;  
 notes.deleteNote(id1);  
   
 // Try to retrieve the deleted note  
 if (!notes.getNote(id1, content)) {  
 cout << "Note " << id1 << " no longer exists!" << endl;  
 }  
   
 // List remaining note IDs  
 cout << "\nRemaining notes:" << endl;  
 for (int id : notes.getAllNoteIds()) {  
 notes.getNote(id, content);  
 cout << "Note " << id << ": " << content.substr(0, 20) << "..." << endl;  
 }  
   
 return 0;  
}

### 9.4.8 Best Practices for File Pointers and Random Access

1. **Always check for file open success** before attempting any file operations
2. **Clear error flags** (like EOF) before seeking or performing new operations
3. **Validate positions** before seeking to avoid going beyond file boundaries
4. **Remember that text files have platform-specific line endings**, which can affect position calculations
5. **Use fixed-size records for simpler random access** in binary files
6. **Implement proper indexing** for variable-length records or complex file structures
7. **Consider file locking** for multi-process or multi-threaded access
8. **Maintain data integrity** by carefully managing position pointers during updates
9. **Create backup mechanisms** for critical file operations

### 9.4.9 Limitations and Considerations

1. **Performance**: While random access is efficient for large files, excessive seeking can degrade performance
2. **File Size Limitations**: Very large files may require 64-bit position indicators
3. **Atomicity**: Multiple file operations aren’t atomic without additional synchronization
4. **Fragmentation**: Frequent updates with variable-length records can lead to file fragmentation
5. **Portability**: File offsets might be interpreted differently across platforms
6. **Corruption Risk**: Random access increases the risk of file corruption if errors aren’t handled properly

### 9.4.10 Summary

Random access file handling in C++ provides powerful capabilities for working directly with specific parts of files without reading through all preceding content. This is essential for:

* Database-like applications
* Large file processing
* Memory-efficient file operations
* File-based data structures
* Custom file formats

By understanding file position pointers and mastering seekg()/seekp() operations, you can implement sophisticated file handling logic that operates efficiently even on large files.

# Chapter 9: File Handling in C++ (Part 4)

## 9.5 Error Handling in File Operations

File operations are inherently error-prone because they interact with external resources outside your program’s control. Proper error handling is crucial for creating robust applications that can gracefully handle problems like missing files, permission issues, disk failures, and corrupted data.

### 9.5.1 Understanding Stream State Flags

C++ file streams maintain internal state flags to track error conditions:

1. **good()**: Returns true if the stream has no errors (all good)
2. **eof()**: Returns true if the end-of-file has been reached
3. **fail()**: Returns true if a logical error occurred (format error, etc.)
4. **bad()**: Returns true if a critical I/O error occurred (disk error, etc.)

Additionally: - The stream object itself can be evaluated in a boolean context, equivalent to !fail() - The ! operator on a stream returns fail()

These states form a hierarchy of conditions:

good() → No errors (all flags clear)  
 ↓  
eof() → End of file reached (only eof bit set)  
 ↓  
fail() → Operation failed (failbit set, possibly eofbit too)  
 ↓  
bad() → Fatal error (badbit set, likely with failbit too)

#include <iostream>  
#include <fstream>  
using namespace std;  
  
void displayStreamState(const fstream& file) {  
 cout << "Stream state flags:" << endl;  
 cout << "- good(): " << (file.good() ? "true" : "false") << endl;  
 cout << "- eof(): " << (file.eof() ? "true" : "false") << endl;  
 cout << "- fail(): " << (file.fail() ? "true" : "false") << endl;  
 cout << "- bad(): " << (file.bad() ? "true" : "false") << endl;  
 cout << "- Boolean conversion: " << (file ? "true" : "false") << endl;  
}  
  
int main() {  
 // Example 1: Successfully open a file  
 fstream file("example.txt", ios::out);  
 cout << "After opening file for writing:" << endl;  
 displayStreamState(file);  
 file.close();  
   
 // Example 2: Try to open a protected file for writing  
 fstream restrictedFile("/etc/passwd", ios::out);  
 cout << "\nAfter trying to open a restricted file:" << endl;  
 displayStreamState(restrictedFile);  
   
 // Example 3: Reading past end of file  
 file.open("example.txt", ios::in);  
 file.seekg(0, ios::end); // Go to end of file  
 char c;  
 file.get(c); // Try to read past EOF  
 cout << "\nAfter reading past EOF:" << endl;  
 displayStreamState(file);  
   
 // Example 4: Clear error flags  
 file.clear(); // Reset all error flags  
 cout << "\nAfter clearing error flags:" << endl;  
 displayStreamState(file);  
   
 file.close();  
 return 0;  
}

### 9.5.2 Basic Error Checking Techniques

#### Simple Boolean Check

The most common method of checking for file errors:

#include <iostream>  
#include <fstream>  
using namespace std;  
  
int main() {  
 // Method 1: Check after opening  
 ifstream file("nonexistent.txt");  
   
 if (!file) { // Equivalent to if (file.fail())  
 cerr << "Error opening file!" << endl;  
 return 1;  
 }  
   
 // Method 2: Check if file is open  
 if (!file.is\_open()) {  
 cerr << "File is not open!" << endl;  
 return 1;  
 }  
   
 // Method 3: Check after read/write operations  
 int value;  
 file >> value;  
   
 if (!file) {  
 cerr << "Error reading from file!" << endl;  
 return 1;  
 }  
   
 return 0;  
}

#### Checking Specific Error Conditions

Different errors require different responses, so check for specific conditions:

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
int main() {  
 // Open file  
 ifstream file("data.txt");  
   
 // Check if file opened  
 if (!file) {  
 cerr << "Failed to open file!" << endl;  
 return 1;  
 }  
   
 // Read data in a loop  
 int value;  
 while (file >> value) {  
 // Process each value  
 cout << "Read: " << value << endl;  
 }  
   
 // Check why the loop ended  
 if (file.eof()) {  
 cout << "Reached end of file normally." << endl;  
 }   
 else if (file.fail()) {  
 cerr << "Format error in file. Non-numeric data encountered." << endl;  
   
 // We can clear the error and try to read what caused it  
 file.clear(); // Reset error flags  
 string badToken;  
 file >> badToken; // Read the offending data  
 cerr << "Bad token: " << badToken << endl;  
 }  
 else if (file.bad()) {  
 cerr << "Critical I/O error occurred!" << endl;  
 // For bad() errors, usually best to abort the operation  
 return 1;  
 }  
   
 file.close();  
 return 0;  
}

#### Using errno for Detailed Error Information

C++ file operations set the global errno variable, which provides additional details:

#include <iostream>  
#include <fstream>  
#include <cerrno> // For errno  
#include <cstring> // For strerror  
using namespace std;  
  
int main() {  
 ifstream file("/root/protected.txt"); // Try to access a protected file  
   
 if (!file) {  
 cerr << "File error: " << strerror(errno) << " (errno=" << errno << ")" << endl;  
   
 // Common errno values for file operations:  
 // ENOENT (2): No such file or directory  
 // EACCES (13): Permission denied  
 // EMFILE (24): Too many open files  
   
 switch (errno) {  
 case ENOENT:  
 cerr << "The file does not exist." << endl;  
 break;  
 case EACCES:  
 cerr << "You don't have permission to access this file." << endl;  
 break;  
 default:  
 cerr << "An unknown error occurred." << endl;  
 }  
   
 return errno;  
 }  
   
 // Rest of the code...  
 file.close();  
 return 0;  
}

### 9.5.3 Exception-Based Error Handling

File streams can be configured to throw exceptions when errors occur:

#include <iostream>  
#include <fstream>  
#include <stdexcept>  
using namespace std;  
  
void readFileWithExceptions(const string& filename) {  
 ifstream file(filename);  
   
 // Configure the file stream to throw exceptions on error  
 file.exceptions(ifstream::failbit | ifstream::badbit);  
   
 try {  
 // Test if file is open  
 if (!file.is\_open()) {  
 throw runtime\_error("Failed to open file: " + filename);  
 }  
   
 string line;  
 while (getline(file, line)) {  
 cout << line << endl;  
 }  
   
 // The following code won't execute if EOF is reached  
 // because we didn't set the eofbit in exceptions()  
 }  
 catch (const ios\_base::failure& e) {  
 cerr << "I/O error: " << e.what() << endl;  
 throw; // Rethrow exception after logging  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 throw; // Rethrow  
 }  
}  
  
int main() {  
 try {  
 // Create a test file first  
 {  
 ofstream testFile("test.txt");  
 testFile << "Line 1: This is a test file.\n";  
 testFile << "Line 2: Testing exception handling.\n";  
 }  
   
 // Read the file with exception handling  
 readFileWithExceptions("test.txt");  
   
 // Try a non-existent file  
 readFileWithExceptions("nonexistent.txt");  
 }  
 catch (const exception& e) {  
 cerr << "Exception caught in main: " << e.what() << endl;  
 return 1;  
 }  
   
 return 0;  
}

### 9.5.4 RAII for File Handling

Resource Acquisition Is Initialization (RAII) is a powerful C++ technique that ensures resources (like files) are properly released:

#include <iostream>  
#include <fstream>  
#include <string>  
#include <stdexcept>  
using namespace std;  
  
// RAII wrapper for file handling  
class FileGuard {  
private:  
 fstream file;  
 string filename;  
 bool is\_open = false;  
   
public:  
 // Constructor opens the file  
 FileGuard(const string& filename, ios::openmode mode)   
 : filename(filename) {  
 file.open(filename, mode);  
 if (!file) {  
 throw runtime\_error("Could not open file: " + filename);  
 }  
 is\_open = true;  
 }  
   
 // No copying allowed  
 FileGuard(const FileGuard&) = delete;  
 FileGuard& operator=(const FileGuard&) = delete;  
   
 // Get access to the underlying stream  
 fstream& getStream() {  
 return file;  
 }  
   
 // Read a line from the file  
 bool readLine(string& line) {  
 return static\_cast<bool>(getline(file, line));  
 }  
   
 // Write to the file  
 template<typename T>  
 FileGuard& operator<<(const T& data) {  
 file << data;  
 if (!file) {  
 throw runtime\_error("Error writing to file: " + filename);  
 }  
 return \*this;  
 }  
   
 // Destructor automatically closes file  
 ~FileGuard() {  
 if (is\_open) {  
 try {  
 file.close();  
 } catch (...) {  
 // Destructors should never throw  
 cerr << "Error while closing file: " << filename << endl;  
 }  
 }  
 }  
};  
  
int main() {  
 try {  
 // Using RAII for file writing  
 {  
 FileGuard outputFile("raii\_example.txt", ios::out);  
 outputFile << "Line 1: RAII test\n";  
 outputFile << "Line 2: File will be closed automatically\n";  
 outputFile << "Line 3: Even if exceptions occur\n";  
   
 // File is closed automatically when outputFile goes out of scope  
 }  
   
 // Using RAII for file reading  
 {  
 FileGuard inputFile("raii\_example.txt", ios::in);  
 string line;  
   
 cout << "File contents:" << endl;  
 while (inputFile.readLine(line)) {  
 cout << line << endl;  
 }  
   
 // File is closed automatically  
 }  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 return 1;  
 }  
   
 return 0;  
}

### 9.5.5 Common File Operation Errors

Let’s explore common file errors and how to handle them:

#### 1. File Not Found

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
bool doesFileExist(const string& filename) {  
 ifstream file(filename);  
 return file.good();  
}  
  
int main() {  
 string filename = "nonexistent.txt";  
   
 if (!doesFileExist(filename)) {  
 cerr << "Error: File '" << filename << "' does not exist." << endl;  
   
 // Potential recovery options:  
 cout << "Options:" << endl;  
 cout << "1. Create a new file" << endl;  
 cout << "2. Specify a different filename" << endl;  
   
 int choice;  
 cout << "Enter choice (1-2): ";  
 cin >> choice;  
   
 if (choice == 1) {  
 ofstream newFile(filename);  
 if (newFile) {  
 cout << "Created new empty file: " << filename << endl;  
 } else {  
 cerr << "Failed to create file!" << endl;  
 }  
 } else if (choice == 2) {  
 cout << "Enter new filename: ";  
 cin >> filename;  
   
 if (doesFileExist(filename)) {  
 cout << "File found: " << filename << endl;  
 } else {  
 cerr << "New file also doesn't exist!" << endl;  
 }  
 }  
 }  
   
 return 0;  
}

#### 2. Permission Denied

#include <iostream>  
#include <fstream>  
#include <cerrno>  
#include <cstring>  
using namespace std;  
  
int main() {  
 // Attempt to write to a file in a restricted location  
 ofstream file("/etc/my\_config.txt");  
   
 if (!file) {  
 cerr << "Error opening file: " << strerror(errno) << endl;  
   
 if (errno == EACCES) {  
 cerr << "Permission denied. Try running with elevated privileges." << endl;  
   
 // Alternative: Try to write to a user-accessible location  
 string userPath = "user\_config.txt";  
 cout << "Trying alternative location: " << userPath << endl;  
   
 ofstream altFile(userPath);  
 if (altFile) {  
 altFile << "Config data that would have gone to system file" << endl;  
 cout << "Successfully wrote to alternative location." << endl;  
 }  
 }  
 }  
   
 return 0;  
}

#### 3. Disk Full Error

#include <iostream>  
#include <fstream>  
#include <vector>  
#include <string>  
using namespace std;  
  
bool writeWithDiskCheck(const string& filename, const string& data) {  
 ofstream file(filename);  
 if (!file) {  
 return false;  
 }  
   
 file << data;  
   
 if (!file) {  
 if (errno == ENOSPC) { // No space left on device  
 cerr << "Disk is full!" << endl;  
 } else {  
 cerr << "Write error: " << strerror(errno) << endl;  
 }  
 return false;  
 }  
   
 file.close();  
 return true;  
}  
  
int main() {  
 // Example of writing a large amount of data, checking for disk full  
 const size\_t megabytes = 100; // Try to write 100MB  
 string largeData(megabytes \* 1024 \* 1024, 'X'); // String of X's  
   
 if (!writeWithDiskCheck("large\_file.dat", largeData)) {  
 cerr << "Failed to write large file." << endl;  
   
 // Recovery options:  
 // 1. Try smaller file  
 string smallerData = largeData.substr(0, largeData.size() / 10); // 10% of original  
 cout << "Trying with smaller file size..." << endl;  
   
 if (writeWithDiskCheck("smaller\_file.dat", smallerData)) {  
 cout << "Successfully wrote smaller file." << endl;  
 }  
   
 // 2. Suggest cleanup  
 cout << "Consider freeing up disk space and trying again." << endl;  
 } else {  
 cout << "Successfully wrote large file." << endl;  
 }  
   
 return 0;  
}

#### 4. Format Errors

#include <iostream>  
#include <fstream>  
#include <string>  
#include <vector>  
using namespace std;  
  
struct Person {  
 string name;  
 int age;  
 double height;  
};  
  
vector<Person> readPeopleData(const string& filename) {  
 ifstream file(filename);  
 if (!file) {  
 throw runtime\_error("Failed to open file: " + filename);  
 }  
   
 vector<Person> people;  
 string line;  
 int lineNumber = 0;  
   
 while (getline(file, line)) {  
 lineNumber++;  
   
 if (line.empty() || line[0] == '#') {  
 continue; // Skip empty lines and comments  
 }  
   
 // Try to parse the line  
 Person person;  
 istringstream iss(line);  
   
 if (iss >> person.name >> person.age >> person.height) {  
 // Successful parse  
 people.push\_back(person);  
 } else {  
 // Format error  
 cerr << "Warning: Format error on line " << lineNumber   
 << ": " << line << endl;  
   
 // Try to recover with partial information  
 person.name = "Unknown";  
 person.age = 0;  
 person.height = 0.0;  
   
 istringstream retry(line);  
 string token;  
   
 if (retry >> token) {  
 person.name = token;  
   
 if (retry >> token) {  
 try {  
 person.age = stoi(token);  
   
 if (retry >> token) {  
 try {  
 person.height = stod(token);  
 } catch (...) {  
 cerr << " Invalid height format" << endl;  
 }  
 }  
 } catch (...) {  
 cerr << " Invalid age format" << endl;  
 }  
 }  
 }  
   
 cerr << " Recovered partial data: " << person.name   
 << ", age=" << person.age   
 << ", height=" << person.height << endl;  
   
 people.push\_back(person);  
 }  
 }  
   
 return people;  
}  
  
int main() {  
 // Create a test file with some bad data  
 {  
 ofstream testFile("people.txt");  
 testFile << "John 25 1.75\n";  
 testFile << "Mary 30 1.65\n";  
 testFile << "Bob thirty 1.80\n"; // Bad format for age  
 testFile << "Alice 28 tall\n"; // Bad format for height  
 testFile << "Mike\n"; // Missing data  
 testFile << "Susan 35 1.70\n";  
 }  
   
 try {  
 vector<Person> people = readPeopleData("people.txt");  
   
 cout << "\nSuccessfully read " << people.size() << " records:" << endl;  
 for (const auto& person : people) {  
 cout << person.name << ", age=" << person.age   
 << ", height=" << person.height << endl;  
 }  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 }  
   
 return 0;  
}

#### 5. Handling Partial Reads

#include <iostream>  
#include <fstream>  
#include <vector>  
using namespace std;  
  
bool readExactly(ifstream& file, char\* buffer, size\_t size) {  
 file.read(buffer, size);  
 size\_t bytesRead = file.gcount();  
   
 if (bytesRead < size) {  
 cerr << "Partial read: expected " << size << " bytes, got "   
 << bytesRead << " bytes" << endl;  
 return false;  
 }  
   
 return true;  
}  
  
int main() {  
 // Create a test file  
 {  
 ofstream outFile("binary.dat", ios::binary);  
 int values[] = {1, 2, 3, 4, 5};  
 outFile.write(reinterpret\_cast<char\*>(values), sizeof(values));  
 }  
   
 // Try to read more data than available  
 ifstream inFile("binary.dat", ios::binary);  
 if (!inFile) {  
 cerr << "Failed to open file!" << endl;  
 return 1;  
 }  
   
 // Get file size  
 inFile.seekg(0, ios::end);  
 streamsize fileSize = inFile.tellg();  
 inFile.seekg(0, ios::beg);  
   
 cout << "File size: " << fileSize << " bytes" << endl;  
   
 // Try to read too much  
 vector<char> buffer(fileSize + 10); // Try to read 10 extra bytes  
   
 inFile.read(buffer.data(), buffer.size());  
 streamsize bytesRead = inFile.gcount();  
   
 cout << "Requested " << buffer.size() << " bytes, actually read "   
 << bytesRead << " bytes" << endl;  
   
 if (inFile.eof()) {  
 cout << "End of file reached during read" << endl;  
 }  
   
 // Proper handling: read only what's available  
 inFile.clear(); // Clear EOF flag  
 inFile.seekg(0, ios::beg); // Back to beginning  
   
 vector<int> data(fileSize / sizeof(int)); // Correctly sized buffer  
   
 if (readExactly(inFile, reinterpret\_cast<char\*>(data.data()), fileSize)) {  
 cout << "Read complete data successfully:" << endl;  
 for (int value : data) {  
 cout << value << " ";  
 }  
 cout << endl;  
 }  
   
 inFile.close();  
 return 0;  
}

### 9.5.6 Recovery Strategies

#### 1. Retry Operations

For transient errors, a retry mechanism can help:

#include <iostream>  
#include <fstream>  
#include <string>  
#include <thread>  
#include <chrono>  
using namespace std;  
  
bool writeWithRetry(const string& filename, const string& data, int maxAttempts = 3) {  
 for (int attempt = 1; attempt <= maxAttempts; ++attempt) {  
 cout << "Attempt " << attempt << " to write file..." << endl;  
   
 ofstream file(filename);  
 if (!file) {  
 cerr << " Failed to open file." << endl;  
 } else {  
 file << data;  
 file.flush(); // Ensure data is written to OS buffer  
   
 if (file) {  
 file.close();  
 cout << " Write successful!" << endl;  
 return true;  
 }  
   
 cerr << " Failed to write data." << endl;  
 }  
   
 if (attempt < maxAttempts) {  
 int delaySeconds = attempt; // Progressive backoff  
 cerr << " Will retry in " << delaySeconds << " seconds..." << endl;  
 this\_thread::sleep\_for(chrono::seconds(delaySeconds));  
 }  
 }  
   
 cerr << "Failed after " << maxAttempts << " attempts." << endl;  
 return false;  
}  
  
int main() {  
 string data = "This is important data that must be saved correctly.";  
   
 bool success = writeWithRetry("output.txt", data);  
   
 if (success) {  
 cout << "Data saved successfully." << endl;  
 } else {  
 cerr << "Failed to save data after multiple attempts." << endl;  
 }  
   
 return success ? 0 : 1;  
}

#### 2. Using Backup Files

#include <iostream>  
#include <fstream>  
#include <string>  
using namespace std;  
  
bool safeUpdateFile(const string& filename, const string& newContent) {  
 // 1. Create backup of original file if it exists  
 string backupFilename = filename + ".bak";  
 bool originalExists = false;  
   
 {  
 ifstream checkFile(filename);  
 originalExists = checkFile.good();  
 }  
   
 if (originalExists) {  
 ifstream original(filename, ios::binary);  
 ofstream backup(backupFilename, ios::binary);  
   
 if (!backup) {  
 cerr << "Error: Failed to create backup file." << endl;  
 return false;  
 }  
   
 // Copy content  
 backup << original.rdbuf();  
   
 if (!backup) {  
 cerr << "Error: Failed to write to backup file." << endl;  
 return false;  
 }  
   
 backup.close();  
 original.close();  
   
 cout << "Created backup: " << backupFilename << endl;  
 }  
   
 // 2. Write new content  
 ofstream output(filename);  
 if (!output) {  
 cerr << "Error: Failed to open output file." << endl;  
 return false;  
 }  
   
 output << newContent;  
   
 if (!output) {  
 cerr << "Error: Failed to write new content." << endl;  
   
 // 3. Restore from backup if something went wrong  
 if (originalExists) {  
 cout << "Attempting to restore from backup..." << endl;  
   
 // Close the failed output file  
 output.close();  
   
 // Copy backup back to original  
 ifstream backupFile(backupFilename, ios::binary);  
 ofstream restoreFile(filename, ios::binary);  
   
 if (backupFile && restoreFile) {  
 restoreFile << backupFile.rdbuf();  
   
 if (restoreFile) {  
 cout << "Successfully restored from backup." << endl;  
 } else {  
 cerr << "Error: Failed to restore from backup." << endl;  
 }  
 } else {  
 cerr << "Error: Failed to open files for restore operation." << endl;  
 }  
 }  
   
 return false;  
 }  
   
 output.close();  
 return true;  
}  
  
int main() {  
 // Create an initial file  
 {  
 ofstream initial("config.txt");  
 initial << "# Configuration File\n";  
 initial << "setting1=value1\n";  
 initial << "setting2=value2\n";  
 }  
   
 // Update it safely  
 string newContent = "# Updated Configuration\n"  
 "setting1=newvalue1\n"  
 "setting2=newvalue2\n"  
 "setting3=value3\n";  
   
 if (safeUpdateFile("config.txt", newContent)) {  
 cout << "File updated successfully!" << endl;  
 } else {  
 cerr << "Failed to update file." << endl;  
 }  
   
 // Read and display the current content  
 cout << "\nCurrent file content:" << endl;  
 cout << "--------------------" << endl;  
   
 ifstream readFile("config.txt");  
 if (readFile) {  
 string line;  
 while (getline(readFile, line)) {  
 cout << line << endl;  
 }  
 }  
   
 return 0;  
}

#### 3. Transaction-Like Updates

#include <iostream>  
#include <fstream>  
#include <string>  
#include <ctime>  
using namespace std;  
  
class FileTransaction {  
private:  
 string filename;  
 string tempFilename;  
 ofstream tempFile;  
 bool committed = false;  
   
public:  
 FileTransaction(const string& fname)   
 : filename(fname), tempFilename(fname + ".tmp") {  
   
 tempFile.open(tempFilename);  
 if (!tempFile) {  
 throw runtime\_error("Could not create temporary file: " + tempFilename);  
 }  
 }  
   
 // Write to the transaction  
 template<typename T>  
 FileTransaction& operator<<(const T& data) {  
 tempFile << data;  
 if (!tempFile) {  
 throw runtime\_error("Error writing to transaction");  
 }  
 return \*this;  
 }  
   
 // Commit the transaction - replace original file with temp file  
 bool commit() {  
 if (committed) {  
 return false; // Already committed  
 }  
   
 tempFile.close();  
 if (tempFile.fail()) {  
 cerr << "Error closing temporary file" << endl;  
 return false;  
 }  
   
 // On some systems, rename fails if target exists  
 if (remove(filename.c\_str()) != 0 && errno != ENOENT) {  
 cerr << "Error removing original file: " << strerror(errno) << endl;  
 return false;  
 }  
   
 if (rename(tempFilename.c\_str(), filename.c\_str()) != 0) {  
 cerr << "Error committing transaction: " << strerror(errno) << endl;  
 return false;  
 }  
   
 committed = true;  
 return true;  
 }  
   
 // Roll back the transaction (discard changes)  
 void rollback() {  
 if (!committed) {  
 if (tempFile.is\_open()) {  
 tempFile.close();  
 }  
   
 // Remove the temporary file  
 remove(tempFilename.c\_str());  
 }  
 }  
   
 // Add a timestamp to the file (utility method)  
 void addTimestamp() {  
 time\_t now = time(nullptr);  
 char timestamp[64];  
 strftime(timestamp, sizeof(timestamp), "# Timestamp: %Y-%m-%d %H:%M:%S",   
 localtime(&now));  
   
 \*this << timestamp << endl;  
 }  
   
 // Destructor automatically rolls back if not committed  
 ~FileTransaction() {  
 if (!committed) {  
 rollback();  
 }  
 }  
};  
  
int main() {  
 try {  
 cout << "Starting file transaction..." << endl;  
   
 // Start transaction  
 FileTransaction transaction("transaction\_test.txt");  
   
 // Add content to the transaction  
 transaction.addTimestamp();  
 transaction << "Line 1: This is a transactional update." << endl;  
 transaction << "Line 2: Either all changes are applied, or none." << endl;  
 transaction << "Line 3: This helps maintain file consistency." << endl;  
   
 // More operations...  
 cout << "Transaction in progress..." << endl;  
   
 // Simulate user decision  
 char commit;  
 cout << "Commit transaction? (y/n): ";  
 cin >> commit;  
   
 if (commit == 'y' || commit == 'Y') {  
 // Commit changes  
 if (transaction.commit()) {  
 cout << "Transaction committed successfully!" << endl;  
 } else {  
 cerr << "Failed to commit transaction." << endl;  
 }  
 } else {  
 // Explicitly roll back (automatic if not committed)  
 transaction.rollback();  
 cout << "Transaction rolled back." << endl;  
 }  
 }  
 catch (const exception& e) {  
 cerr << "Error: " << e.what() << endl;  
 return 1;  
 }  
   
 return 0;  
}

### 9.5.7 Best Practices for Error Handling in File Operations

1. **Always check if file operations succeeded**
   * Don’t assume files will open or operations will succeed
   * Check return values and stream states
2. **Use appropriate error reporting for your audience**
   * For end users: Simple, actionable messages
   * For developers: Detailed technical information
   * Consider logging errors to files for later analysis
3. **Use RAII wrappers for file resources**
   * Ensures files are closed even if exceptions occur
   * Prevents resource leaks
   * Makes code more readable and maintainable
4. **Provide recovery mechanisms when possible**
   * Retry operations for transient errors
   * Use alternative locations if primary ones fail
   * Create backups before modifying important files
5. **Be specific about error conditions**
   * Check for specific errors (EOF, permission denied, etc.)
   * Different errors require different responses
6. **Handle format errors gracefully**
   * Provide helpful error messages with line numbers
   * Try to recover partial data when possible
   * Skip bad records rather than aborting entirely
7. **Manage partial reads and writes**
   * Check how many bytes were actually read/written
   * Don’t assume operations handled all requested data
8. **Use explicit close operations with error checking**
   * Check if close operations succeeded
   * Flush buffers before closing important files
9. **Consider transaction-like approaches for critical updates**
   * Write to temporary files, then rename
   * Create backups before modifications
   * Use atomic operations when available
10. **Clean up temporary files**
    * Remove temporary files when they’re no longer needed
    * Consider using unique names for temporary files
    * Handle cleanup in destructors or finally blocks

By implementing these error handling strategies, you’ll create more robust file handling code that can gracefully handle the unpredictable nature of external resources like files.

# Chapter 10: Templates in C++ (Part 1)

## Introduction to Templates

Templates are one of C++’s most powerful features, allowing you to write generic code that works with any data type while maintaining type safety. They are the foundation of C++’s Standard Template Library (STL) and modern C++ programming. Templates enable you to create flexible, reusable code without sacrificing performance or type safety.

In essence, templates provide a way to write a single function or class that can operate on different data types without rewriting the same logic for each type. The compiler generates the appropriate code for each specific type at compile time.

## 10.1 Function Templates

Function templates allow you to write a single function that can operate on different types. Instead of writing multiple overloaded functions with identical logic but different parameter types, you can write a single function template.

### 10.1.1 Basic Function Template Syntax

The basic syntax for a function template is:

template <typename T>  
return\_type function\_name(parameters) {  
 // Function body  
}

Here, T is a template parameter that will be replaced with an actual type when the template is instantiated. The keyword typename can be replaced with the equivalent keyword class.

Let’s look at a simple example:

#include <iostream>  
using namespace std;  
  
// Function template to find maximum of two values  
template <typename T>  
T maximum(T a, T b) {  
 return (a > b) ? a : b;  
}  
  
int main() {  
 // Using the template with integers  
 cout << "Max of 10 and 20: " << maximum<int>(10, 20) << endl;  
   
 // Using the template with doubles  
 cout << "Max of 10.5 and 20.7: " << maximum<double>(10.5, 20.7) << endl;  
   
 // Using the template with characters  
 cout << "Max of 'a' and 'z': " << maximum<char>('a', 'z') << endl;  
   
 // Type deduction (compiler determines the type)  
 cout << "Max of 100 and 200: " << maximum(100, 200) << endl;  
   
 return 0;  
}

When you call the template function with maximum<int>(10, 20), the compiler generates a specific version of the function for integers. The <int> part explicitly specifies the type, but in many cases, the compiler can deduce the type from the arguments, so you can just write maximum(10, 20).

### 10.1.2 Template Type Deduction

C++ compilers can often deduce the template parameter types based on the function arguments. This makes template code cleaner and more readable:

// Template instantiations with explicit type specification  
int max\_int = maximum<int>(10, 20);  
double max\_double = maximum<double>(10.5, 20.7);  
  
// Same effect with type deduction  
int max\_int2 = maximum(10, 20); // T is deduced as int  
double max\_double2 = maximum(10.5, 20.7); // T is deduced as double

However, there are cases where explicit specification is necessary:

// Explicit specification when deduction might be ambiguous  
double result = maximum<double>(10, 20.5); // Force T to be double

Without the explicit <double> here, the compiler would encounter ambiguity since the arguments are of different types.

### 10.1.3 Multiple Template Parameters

A function template can have multiple template parameters:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Template with two different types  
template <typename T, typename U>  
T convert\_and\_multiply(T a, U b) {  
 return a \* static\_cast<T>(b);  
}  
  
int main() {  
 // Using the template with different type combinations  
 int result1 = convert\_and\_multiply<int, double>(10, 2.5); // 10 \* 2 = 20  
 double result2 = convert\_and\_multiply<double, int>(10.5, 2); // 10.5 \* 2 = 21.0  
   
 cout << "Result 1: " << result1 << endl; // Output: 20  
 cout << "Result 2: " << result2 << endl; // Output: 21  
   
 return 0;  
}

### 10.1.4 Template Specialization

Sometimes, you want to provide a special implementation of a template for a specific type. This is called template specialization:

#include <iostream>  
#include <string>  
using namespace std;  
  
// General template  
template <typename T>  
T add(T a, T b) {  
 cout << "General template called" << endl;  
 return a + b;  
}  
  
// Specialization for char\*  
template <>  
const char\* add<const char\*>(const char\* a, const char\* b) {  
 cout << "Specialized template for const char\* called" << endl;  
 string result = string(a) + string(b);  
 // Warning: Returning a pointer to a temporary - for demonstration only  
 // In real code, you would need to handle memory properly  
 return strdup(result.c\_str());  
}  
  
int main() {  
 int int\_result = add(5, 3); // Uses general template  
 double double\_result = add(5.7, 3.2); // Uses general template  
   
 const char\* s1 = "Hello, ";  
 const char\* s2 = "world!";  
 const char\* str\_result = add(s1, s2); // Uses specialized template  
   
 cout << "Int result: " << int\_result << endl;  
 cout << "Double result: " << double\_result << endl;  
 cout << "String result: " << str\_result << endl;  
   
 // Clean up (since we used strdup)  
 free((void\*)str\_result);  
   
 return 0;  
}

In this example, the specialized version for const char\* provides a custom implementation to handle string concatenation.

### 10.1.5 Non-Type Template Parameters

Templates can also have non-type parameters, such as integers, pointers, or references:

#include <iostream>  
#include <array>  
using namespace std;  
  
// Template with a non-type parameter  
template <typename T, int Size>  
class Buffer {  
private:  
 T data[Size];  
   
public:  
 Buffer() : data{} {}  
   
 T& operator[](int index) {  
 if (index < 0 || index >= Size) {  
 throw out\_of\_range("Index out of bounds");  
 }  
 return data[index];  
 }  
   
 int size() const {  
 return Size;  
 }  
};  
  
int main() {  
 // Create buffers of different types and sizes  
 Buffer<int, 10> int\_buffer;  
 Buffer<double, 5> double\_buffer;  
   
 // Populate buffers  
 for (int i = 0; i < int\_buffer.size(); ++i) {  
 int\_buffer[i] = i \* 10;  
 }  
   
 for (int i = 0; i < double\_buffer.size(); ++i) {  
 double\_buffer[i] = i \* 1.5;  
 }  
   
 // Display buffer contents  
 cout << "Int buffer: ";  
 for (int i = 0; i < int\_buffer.size(); ++i) {  
 cout << int\_buffer[i] << " ";  
 }  
 cout << endl;  
   
 cout << "Double buffer: ";  
 for (int i = 0; i < double\_buffer.size(); ++i) {  
 cout << double\_buffer[i] << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

In this example, Size is a non-type template parameter that determines the size of the array at compile time.

### 10.1.6 Default Template Arguments

Template parameters can have default arguments:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Template with default parameter  
template <typename T = int>  
T identity(T value) {  
 return value;  
}  
  
int main() {  
 // Using the template with explicit type  
 double d = identity<double>(3.14);  
 string s = identity<string>("Hello");  
   
 // Using default argument (T = int)  
 int i = identity(42);  
   
 cout << "Double: " << d << endl;  
 cout << "String: " << s << endl;  
 cout << "Int: " << i << endl;  
   
 return 0;  
}

### 10.1.7 Using Concepts (C++20)

In C++20, you can use concepts to constrain template parameters:

#include <iostream>  
#include <concepts>  
using namespace std;  
  
// Define a concept for types that support addition  
template <typename T>  
concept Addable = requires(T a, T b) {  
 { a + b } -> std::same\_as<T>;  
};  
  
// Function that only works with Addable types  
template <Addable T>  
T add(T a, T b) {  
 return a + b;  
}  
  
// Another way to constrain a template  
template <typename T>  
requires Addable<T>  
T multiply\_and\_add(T a, T b, T c) {  
 return a \* b + c;  
}  
  
int main() {  
 cout << "Add integers: " << add(5, 3) << endl;  
 cout << "Add doubles: " << add(2.5, 3.7) << endl;  
 cout << "Multiply and add: " << multiply\_and\_add(2, 3, 4) << endl;  
   
 // This would cause a compilation error if uncommented:  
 // struct NonAddable {};  
 // add(NonAddable{}, NonAddable{});  
   
 return 0;  
}

Concepts provide a way to express requirements on template arguments, making template error messages more readable and allowing more precise control over which types can be used with a template.

### 10.1.8 Function Template Best Practices

1. **Prefer type deduction** when possible to make code cleaner.
2. **Use constraints** (C++20 concepts or SFINAE in earlier versions) to restrict which types can be used with your templates.
3. **Be mindful of implicit conversions** that might happen during type deduction.
4. **Specialize templates** only when you need different behavior for specific types.
5. **Consider using forwarding references** (T&&) with std::forward for perfect forwarding.
6. **Document template requirements** clearly, especially what operations the types need to support.
7. **Write clear error messages** for when requirements aren’t met.

## 10.2 Class Templates

Class templates extend the template concept to classes, allowing you to create a blueprint for a class where some types are left as parameters to be specified later.

### 10.2.1 Basic Class Template Syntax

The basic syntax for a class template is:

template <typename T>  
class ClassName {  
 // Class members  
};

Let’s look at a simple example of a generic container:

#include <iostream>  
using namespace std;  
  
// Class template  
template <typename T>  
class Container {  
private:  
 T element;  
   
public:  
 Container(T arg) : element(arg) {}  
   
 T getElement() const {  
 return element;  
 }  
   
 void setElement(T arg) {  
 element = arg;  
 }  
};  
  
int main() {  
 // Create containers of different types  
 Container<int> int\_container(42);  
 Container<double> double\_container(3.14);  
 Container<string> string\_container("Hello, templates!");  
   
 // Access and modify container contents  
 cout << "Int container: " << int\_container.getElement() << endl;  
 cout << "Double container: " << double\_container.getElement() << endl;  
 cout << "String container: " << string\_container.getElement() << endl;  
   
 int\_container.setElement(100);  
 cout << "Modified int container: " << int\_container.getElement() << endl;  
   
 return 0;  
}

Unlike function templates, the template arguments for class templates usually cannot be deduced from the constructor arguments (prior to C++17), so you must specify them explicitly.

### 10.2.2 Member Functions in Class Templates

Member functions of a class template can be defined either inside or outside the class definition:

// Inside the class definition  
template <typename T>  
class Container {  
 T element;  
public:  
 Container(T arg) : element(arg) {}  
   
 T getElement() const { return element; }  
 void setElement(T arg) { element = arg; }  
};  
  
// Outside the class definition  
template <typename T>  
class Box {  
 T content;  
public:  
 Box(T arg);  
 T getContent() const;  
 void setContent(T arg);  
};  
  
// Implementation of member functions  
template <typename T>  
Box<T>::Box(T arg) : content(arg) {}  
  
template <typename T>  
T Box<T>::getContent() const {  
 return content;  
}  
  
template <typename T>  
void Box<T>::setContent(T arg) {  
 content = arg;  
}

When defining member functions outside the class, you need to repeat the template declaration and use the full qualified name (Box<T>::methodName).

### 10.2.3 Multiple Template Parameters

Like function templates, class templates can have multiple template parameters:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Class with multiple template parameters  
template <typename KeyType, typename ValueType>  
class Pair {  
private:  
 KeyType key;  
 ValueType value;  
   
public:  
 Pair(KeyType k, ValueType v) : key(k), value(v) {}  
   
 KeyType getKey() const { return key; }  
 ValueType getValue() const { return value; }  
   
 void setKey(KeyType k) { key = k; }  
 void setValue(ValueType v) { value = v; }  
   
 void display() const {  
 cout << "Key: " << key << ", Value: " << value << endl;  
 }  
};  
  
int main() {  
 // Create pairs with different type combinations  
 Pair<int, string> student(101, "John Doe");  
 Pair<string, double> temperature("Celsius", 25.5);  
 Pair<char, int> ascii('A', 65);  
   
 // Display pairs  
 student.display();  
 temperature.display();  
 ascii.display();  
   
 return 0;  
}

### 10.2.4 Template Specialization for Classes

Like function templates, class templates can be specialized for specific types:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Primary template  
template <typename T>  
class Storage {  
private:  
 T data;  
   
public:  
 Storage(T d) : data(d) {}  
   
 void print() const {  
 cout << "Generic storage contains: " << data << endl;  
 }  
};  
  
// Full specialization for bool  
template <>  
class Storage<bool> {  
private:  
 bool data;  
   
public:  
 Storage(bool d) : data(d) {}  
   
 void print() const {  
 cout << "Boolean storage contains: " << (data ? "true" : "false") << endl;  
 }  
};  
  
// Partial specialization for pointer types  
template <typename T>  
class Storage<T\*> {  
private:  
 T\* data;  
   
public:  
 Storage(T\* d) : data(d) {}  
   
 void print() const {  
 cout << "Pointer storage contains address: " << data;  
 if (data) {  
 cout << " with value: " << \*data;  
 } else {  
 cout << " (nullptr)";  
 }  
 cout << endl;  
 }  
   
 ~Storage() {  
 // Specialized cleanup for pointer type  
 // Note: This assumes ownership of the pointer, which may not be appropriate  
 // in all cases.  
 delete data;  
 }  
};  
  
int main() {  
 // Using generic template  
 Storage<int> int\_storage(42);  
 Storage<string> string\_storage("Hello, world!");  
   
 // Using specialized template for bool  
 Storage<bool> bool\_storage(true);  
   
 // Using specialized template for pointer type  
 int\* value = new int(100);  
 Storage<int\*> ptr\_storage(value);  
   
 // Print all storages  
 int\_storage.print();  
 string\_storage.print();  
 bool\_storage.print();  
 ptr\_storage.print();  
   
 return 0;  
}

There are two types of class template specialization: - **Full Specialization**: The entire template is specialized for a specific type. - **Partial Specialization**: Only some template parameters are specialized, or a pattern of types (like pointers) is specialized.

### 10.2.5 Default Template Arguments for Classes

Like function templates, class templates can have default template arguments:

#include <iostream>  
#include <vector>  
using namespace std;  
  
// Class with default template argument  
template <typename T, typename Container = vector<T>>  
class Stack {  
private:  
 Container elements;  
   
public:  
 void push(const T& element) {  
 elements.push\_back(element);  
 }  
   
 T pop() {  
 if (elements.empty()) {  
 throw out\_of\_range("Stack is empty");  
 }  
 T top = elements.back();  
 elements.pop\_back();  
 return top;  
 }  
   
 bool empty() const {  
 return elements.empty();  
 }  
   
 size\_t size() const {  
 return elements.size();  
 }  
};  
  
int main() {  
 // Using default container (vector)  
 Stack<int> int\_stack;  
   
 // Push elements  
 int\_stack.push(10);  
 int\_stack.push(20);  
 int\_stack.push(30);  
   
 cout << "Stack size: " << int\_stack.size() << endl;  
   
 // Pop elements  
 while (!int\_stack.empty()) {  
 cout << "Popped: " << int\_stack.pop() << endl;  
 }  
   
 return 0;  
}

In this example, the Stack template has a default container type of vector<T>, but you could provide a different container type if needed.

### 10.2.6 Template Parameters as Template Template Parameters

You can have template parameters that are themselves templates:

#include <iostream>  
#include <vector>  
#include <deque>  
#include <list>  
using namespace std;  
  
// Template template parameter  
template <  
 typename T,  
 template <typename, typename> class Container = vector,  
 typename Allocator = allocator<T>  
>  
class Stack {  
private:  
 Container<T, Allocator> elements;  
   
public:  
 void push(const T& element) {  
 elements.push\_back(element);  
 }  
   
 T pop() {  
 if (elements.empty()) {  
 throw out\_of\_range("Stack is empty");  
 }  
 T top = elements.back();  
 elements.pop\_back();  
 return top;  
 }  
   
 bool empty() const {  
 return elements.empty();  
 }  
   
 size\_t size() const {  
 return elements.size();  
 }  
};  
  
int main() {  
 // Using different container types  
 Stack<int> vector\_stack; // Default: vector  
 Stack<int, deque> deque\_stack;  
 Stack<int, list> list\_stack;  
   
 // Push elements to all stacks  
 for (int i = 1; i <= 5; ++i) {  
 vector\_stack.push(i \* 10);  
 deque\_stack.push(i \* 100);  
 list\_stack.push(i \* 1000);  
 }  
   
 // Pop from vector stack  
 cout << "Vector stack contents: ";  
 while (!vector\_stack.empty()) {  
 cout << vector\_stack.pop() << " ";  
 }  
 cout << endl;  
   
 // Pop from deque stack  
 cout << "Deque stack contents: ";  
 while (!deque\_stack.empty()) {  
 cout << deque\_stack.pop() << " ";  
 }  
 cout << endl;  
   
 // Pop from list stack  
 cout << "List stack contents: ";  
 while (!list\_stack.empty()) {  
 cout << list\_stack.pop() << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

This advanced feature allows you to parameterize your class on different container implementations that share a common interface.

### 10.2.7 Nested Class Templates

You can nest class templates within class templates:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Outer template class  
template <typename T>  
class Outer {  
private:  
 T outerValue;  
   
public:  
 Outer(T value) : outerValue(value) {}  
   
 // Nested template class  
 template <typename U>  
 class Inner {  
 private:  
 U innerValue;  
 T\* outerPtr; // Can reference the outer class's type  
   
 public:  
 Inner(U value, Outer<T>\* outer) :   
 innerValue(value), outerPtr(outer) {}  
   
 void display() const {  
 cout << "Inner value: " << innerValue << endl;  
 cout << "Outer value: " << outerPtr->getOuterValue() << endl;  
 }  
 };  
   
 T getOuterValue() const {  
 return outerValue;  
 }  
   
 // Factory method to create Inner objects  
 template <typename U>  
 Inner<U> createInner(U value) {  
 return Inner<U>(value, this);  
 }  
};  
  
int main() {  
 // Create an outer object  
 Outer<string> outer("Hello from outer");  
   
 // Create inner objects  
 auto inner1 = outer.createInner(42);  
 auto inner2 = outer.createInner(3.14);  
   
 // Alternative direct creation  
 typename Outer<string>::Inner<char> inner3('A', &outer);  
   
 // Display inner objects  
 inner1.display();  
 inner2.display();  
 inner3.display();  
   
 return 0;  
}

Note the use of typename when referring to the nested class template from outside the class.

### 10.2.8 Friend Functions and Templates

Template classes can have friend functions and friend classes:

#include <iostream>  
using namespace std;  
  
// Forward declaration  
template <typename T>  
class Container;  
  
// Friend function declaration  
template <typename T>  
void display(const Container<T>& c);  
  
// Class template with friend  
template <typename T>  
class Container {  
private:  
 T value;  
   
public:  
 Container(T v) : value(v) {}  
   
 // Friend function declaration  
 friend void display<T>(const Container<T>& c);  
   
 // Non-template friend function  
 friend void reset(Container<T>& c) {  
 cout << "Resetting container value from " << c.value << " to default" << endl;  
 c.value = T();  
 }  
   
 // Friend class template  
 template <typename U>  
 friend class Inspector;  
};  
  
// Friend function definition  
template <typename T>  
void display(const Container<T>& c) {  
 cout << "Container value: " << c.value << endl;  
}  
  
// Friend class template  
template <typename T>  
class Inspector {  
public:  
 static void inspect(const Container<T>& c) {  
 cout << "Inspector accessing private value: " << c.value << endl;  
 }  
};  
  
int main() {  
 Container<int> ic(42);  
 Container<string> sc("Hello");  
   
 // Use friend function  
 display(ic);  
 display(sc);  
   
 // Use non-template friend function  
 reset(ic);  
 display(ic);  
   
 // Use friend class  
 Inspector<int>::inspect(ic);  
 Inspector<string>::inspect(sc);  
   
 return 0;  
}

Friend declarations in templates can be complex because they involve both the template mechanism and the friendship relationship.

### 10.2.9 Class Template Best Practices

1. **Make template parameters meaningful** with descriptive names like ValueType instead of just T.
2. **Use default template arguments** when sensible defaults exist.
3. **Consider providing type aliases** for common instantiations of your templates.
4. **Be careful with specializations** as they can lead to code duplication.
5. **Document template requirements** clearly, especially what operations the types need to support.
6. **Consider using concepts (C++20)** to constrain template parameters.
7. **Keep implementation details in the .cpp file when possible** by using explicit instantiation.
8. **Design for minimal dependencies** between template parameters.
9. **Use static assertions** to verify template requirements at compile time.
10. **Be mindful of code bloat** from excessive template instantiations.

By following these best practices, you can create effective, maintainable, and reusable template classes.

# Chapter 10: Templates in C++ (Part 2)

## 10.3 Template Specialization

Template specialization allows you to provide custom implementations for specific template arguments. This powerful mechanism enables you to optimize for particular types or handle edge cases that don’t fit with your generic implementation.

### 10.3.1 Full Template Specialization

Full specialization completely overrides the primary template for a specific type:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Primary template  
template <typename T>  
class DataHandler {  
public:  
 void process(T data) {  
 cout << "Generic processing of data: " << data << endl;  
 }  
};  
  
// Full specialization for int  
template <>  
class DataHandler<int> {  
public:  
 void process(int data) {  
 cout << "Specialized processing for int: " << data \* 2 << endl;  
 }  
};  
  
// Full specialization for string  
template <>  
class DataHandler<string> {  
public:  
 void process(string data) {  
 cout << "Specialized processing for string: " << data.length()   
 << " characters" << endl;  
 }  
};  
  
int main() {  
 // Using primary template  
 DataHandler<double> dh\_double;  
 dh\_double.process(3.14); // Generic processing  
   
 // Using specialized templates  
 DataHandler<int> dh\_int;  
 dh\_int.process(42); // Specialized processing for int  
   
 DataHandler<string> dh\_string;  
 dh\_string.process("Hello, template specialization!"); // Specialized for string  
   
 return 0;  
}

In this example, when DataHandler is instantiated with int or string, the specialized implementations are used instead of the primary template.

### 10.3.2 Function Template Specialization

Function templates can also be specialized:

#include <iostream>  
#include <string>  
#include <cstring>  
using namespace std;  
  
// Primary template function  
template <typename T>  
T max\_value(T a, T b) {  
 cout << "Using generic max\_value" << endl;  
 return (a > b) ? a : b;  
}  
  
// Specialization for C-style strings (const char\*)  
template <>  
const char\* max\_value<const char\*>(const char\* a, const char\* b) {  
 cout << "Using specialized max\_value for const char\*" << endl;  
 return (strcmp(a, b) > 0) ? a : b;  
}  
  
int main() {  
 int i1 = 42, i2 = 73;  
 cout << "Max int: " << max\_value(i1, i2) << endl;  
   
 const char\* s1 = "apple";  
 const char\* s2 = "banana";  
 cout << "Max string: " << max\_value(s1, s2) << endl;  
   
 // Using with auto type deduction  
 auto result = max\_value<double>(3.14, 2.71);  
 cout << "Max double: " << result << endl;  
   
 return 0;  
}

Function specializations are particularly useful when the generic algorithm doesn’t work for a specific type, like comparing C-strings which requires strcmp() instead of the > operator.

### 10.3.3 Partial Template Specialization

Partial specialization allows you to specialize a template for a subset of possible template arguments. This is only available for class templates, not function templates:

#include <iostream>  
#include <type\_traits>  
using namespace std;  
  
// Primary template  
template <typename T, typename U>  
class Pair {  
public:  
 Pair(T first, U second) : first\_(first), second\_(second) {  
 cout << "Primary template used" << endl;  
 }  
   
 void display() const {  
 cout << "Pair: " << first\_ << ", " << second\_ << endl;  
 }  
   
private:  
 T first\_;  
 U second\_;  
};  
  
// Partial specialization for when both types are the same  
template <typename T>  
class Pair<T, T> {  
public:  
 Pair(T first, T second) : first\_(first), second\_(second) {  
 cout << "Partial specialization for same types used" << endl;  
 }  
   
 void display() const {  
 cout << "Same-type pair: " << first\_ << ", " << second\_ << endl;  
 }  
   
 T max() const {  
 return first\_ > second\_ ? first\_ : second\_;  
 }  
   
private:  
 T first\_;  
 T second\_;  
};  
  
// Partial specialization for pointer types  
template <typename T, typename U>  
class Pair<T\*, U\*> {  
public:  
 Pair(T\* first, U\* second) : first\_(first), second\_(second) {  
 cout << "Partial specialization for pointers used" << endl;  
 }  
   
 void display() const {  
 cout << "Pointer pair: " << \*first\_ << ", " << \*second\_ << endl;  
 }  
   
private:  
 T\* first\_;  
 U\* second\_;  
};  
  
int main() {  
 // Uses primary template  
 Pair<int, string> p1(42, "hello");  
 p1.display();  
   
 // Uses partial specialization for same types  
 Pair<double, double> p2(3.14, 2.71);  
 p2.display();  
 cout << "Max value: " << p2.max() << endl; // Additional method available  
   
 // Uses partial specialization for pointers  
 int x = 10, y = 20;  
 Pair<int, int> p3(&x, &y);  
 p3.display();  
   
 return 0;  
}

Partial specialization patterns can include: - Same template parameters (Pair<T,T>) - Pointer types (Pair<T\*,U\*>) - Reference types (Pair<T&,U&>) - Const qualifiers (Pair<const T, U>) - Nested templates (Pair<vector<T>, U>)

### 10.3.4 Specialization for Array Types

Templates can be specialized for array types, which is particularly useful for creating safe array wrappers:

#include <iostream>  
#include <cstring>  
using namespace std;  
  
// Primary template  
template <typename T>  
class Array {  
public:  
 explicit Array(size\_t size) : size\_(size) {  
 data\_ = new T[size];  
 cout << "Generic Array constructor" << endl;  
 }  
   
 ~Array() {  
 delete[] data\_;  
 }  
   
 T& operator[](size\_t index) {  
 return data\_[index];  
 }  
   
 size\_t size() const {  
 return size\_;  
 }  
   
private:  
 T\* data\_;  
 size\_t size\_;  
};  
  
// Specialization for char arrays (strings)  
template <>  
class Array<char> {  
public:  
 explicit Array(size\_t size) : size\_(size) {  
 data\_ = new char[size + 1]; // Extra space for null terminator  
 data\_[0] = '\0'; // Initialize as empty string  
 cout << "Specialized Array<char> constructor" << endl;  
 }  
   
 ~Array() {  
 delete[] data\_;  
 }  
   
 char& operator[](size\_t index) {  
 return data\_[index];  
 }  
   
 size\_t size() const {  
 return size\_;  
 }  
   
 // String-specific methods  
 size\_t length() const {  
 return strlen(data\_);  
 }  
   
 void assign(const char\* str) {  
 size\_t len = strlen(str);  
 if (len > size\_) {  
 len = size\_; // Truncate if too long  
 }  
 strncpy(data\_, str, len);  
 data\_[len] = '\0'; // Ensure null termination  
 }  
   
private:  
 char\* data\_;  
 size\_t size\_; // Size excluding null terminator  
};  
  
int main() {  
 // Generic array of integers  
 Array<int> intArray(5);  
 for (size\_t i = 0; i < intArray.size(); ++i) {  
 intArray[i] = i \* 10;  
 }  
   
 cout << "Int array contents: ";  
 for (size\_t i = 0; i < intArray.size(); ++i) {  
 cout << intArray[i] << " ";  
 }  
 cout << endl;  
   
 // Specialized array for characters (string)  
 Array<char> charArray(20); // Space for 20 chars + null terminator  
 charArray.assign("Hello, specialization!");  
   
 cout << "Char array contents: " << &charArray[0] << endl;  
 cout << "Char array length: " << charArray.length() << endl;  
   
 return 0;  
}

This approach makes working with character arrays safer and more intuitive by providing string-specific functionality.

### 10.3.5 Specialization Based on Type Traits

Modern C++ provides type traits that can be used with SFINAE (Substitution Failure Is Not An Error) to create more sophisticated specializations:

#include <iostream>  
#include <type\_traits>  
#include <string>  
#include <vector>  
using namespace std;  
  
// Primary template handles non-container types  
template <typename T, typename = void>  
class ElementPrinter {  
public:  
 static void print(const T& value) {  
 cout << value << endl;  
 }  
};  
  
// Specialization for container types (using SFINAE with std::void\_t and decltype)  
template <typename T>  
class ElementPrinter<T,   
 std::void\_t<decltype(std::declval<T>().begin()), // Check for begin() method  
 decltype(std::declval<T>().end())>> // Check for end() method  
{  
public:  
 static void print(const T& container) {  
 cout << "Container elements: ";  
 for (const auto& elem : container) {  
 cout << elem << " ";  
 }  
 cout << endl;  
 }  
};  
  
// Another example with std::enable\_if  
template <typename T>  
typename std::enable\_if<std::is\_arithmetic<T>::value, T>::type  
square(T value) {  
 return value \* value;  
}  
  
template <typename T>  
typename std::enable\_if<!std::is\_arithmetic<T>::value, void>::type  
square(T value) {  
 cout << "Cannot square non-arithmetic type" << endl;  
}  
  
int main() {  
 // Test ElementPrinter  
 int num = 42;  
 string str = "Hello";  
 vector<int> vec = {1, 2, 3, 4, 5};  
   
 ElementPrinter<int>::print(num); // Uses primary template  
 ElementPrinter<string>::print(str); // Uses primary template  
 ElementPrinter<vector<int>>::print(vec); // Uses container specialization  
   
 // Test square function  
 cout << "Square of 5: " << square(5) << endl;  
 cout << "Square of 3.14: " << square(3.14) << endl;  
 square("Cannot square me"); // Uses non-arithmetic version  
   
 return 0;  
}

In C++17 and later, you can use if constexpr for compile-time decisions that are more readable than SFINAE:

#include <iostream>  
#include <type\_traits>  
using namespace std;  
  
// Single template function with compile-time branching  
template <typename T>  
auto process(T value) {  
 if constexpr (is\_integral\_v<T>) {  
 cout << "Processing integral type" << endl;  
 return value \* 2;  
 } else if constexpr (is\_floating\_point\_v<T>) {  
 cout << "Processing floating-point type" << endl;  
 return value \* 3.14;  
 } else {  
 cout << "Processing other type" << endl;  
 return value;  
 }  
}  
  
int main() {  
 int i = 10;  
 double d = 2.5;  
 string s = "test";  
   
 auto result1 = process(i); // int version  
 auto result2 = process(d); // double version  
 auto result3 = process(s); // other version  
   
 cout << "Results: " << result1 << ", " << result2 << ", " << result3 << endl;  
   
 return 0;  
}

### 10.3.6 When to Use Template Specialization

Template specialization is powerful but should be used judiciously:

1. **Type-specific optimizations**: When you can provide a more efficient implementation for certain types
2. **Edge cases**: When the generic algorithm doesn’t work for a specific type
3. **Different behaviors**: When you need fundamentally different behavior for certain types
4. **Adding functionality**: When you want to provide additional methods for specific types

Watch out for these pitfalls: 1. **Code bloat**: Excessive specializations can lead to code maintenance issues 2. **Surprising behavior**: Users might be surprised when their code behaves differently for different types 3. **Overspecialization**: Don’t specialize when simple overloading would work

## 10.4 Variadic Templates

Variadic templates allow you to write templates that accept an arbitrary number of arguments of arbitrary types. This feature, introduced in C++11, enables powerful metaprogramming and perfect forwarding.

### 10.4.1 Basic Variadic Templates

The syntax for variadic templates uses ... (ellipsis) in three different contexts: 1. **Parameter pack declaration**: template <typename... Args> 2. **Function parameter declaration**: void func(Args... args) 3. **Parameter pack expansion**: func(args...)

Here’s a simple example:

#include <iostream>  
using namespace std;  
  
// Variadic template function that prints all arguments  
template <typename... Args>  
void printAll(Args... args) {  
 // This doesn't work directly because args is a parameter pack  
 // We need a way to "unpack" it  
}  
  
// Helper function to print a single argument  
template <typename T>  
void print(T arg) {  
 cout << arg << " ";  
}  
  
// Base case for recursion (no arguments)  
void printAllHelper() {  
 cout << endl; // Just print a newline  
}  
  
// Recursive case with variadic template  
template <typename First, typename... Rest>  
void printAllHelper(First first, Rest... rest) {  
 print(first); // Process the first argument  
 printAllHelper(rest...); // Process the rest recursively  
}  
  
// User-friendly wrapper  
template <typename... Args>  
void printAll(Args... args) {  
 printAllHelper(args...);  
}  
  
int main() {  
 // Call with different number and types of arguments  
 printAll(1, 2, 3);  
 printAll("Hello", "variadic", "templates", 2023);  
 printAll(3.14, 'A', true, 42, "mixed types");  
   
 return 0;  
}

This example recursively processes the arguments in the parameter pack by peeling off one argument at a time.

### 10.4.2 Variadic Templates with Fold Expressions (C++17)

C++17 introduced fold expressions to simplify working with parameter packs:

#include <iostream>  
#include <string>  
using namespace std;  
  
// Print all arguments using a fold expression  
template <typename... Args>  
void printAll(Args... args) {  
 // Fold expression with comma operator  
 ((cout << args << " "), ...);  
 cout << endl;  
}  
  
// Sum all arguments  
template <typename... Args>  
auto sum(Args... args) {  
 // Fold expression with addition operator  
 return (... + args);  
}  
  
// Check if all conditions are true  
template <typename... Args>  
bool all(Args... args) {  
 // Fold expression with AND operator  
 return (... && args);  
}  
  
// Check if any condition is true  
template <typename... Args>  
bool any(Args... args) {  
 // Fold expression with OR operator  
 return (... || args);  
}  
  
int main() {  
 // Test printing  
 printAll(1, 2, 3, 4, 5);  
 printAll("C++", "is", "awesome");  
   
 // Test sum  
 cout << "Sum of {1, 2, 3, 4, 5} = " << sum(1, 2, 3, 4, 5) << endl;  
 cout << "Sum of {3.14, 2.71, 1.41} = " << sum(3.14, 2.71, 1.41) << endl;  
   
 // Test logical operations  
 cout << "all(true, true, true) = " << all(true, true, true) << endl;  
 cout << "all(true, false, true) = " << all(true, false, true) << endl;  
 cout << "any(false, false, true) = " << any(false, false, true) << endl;  
 cout << "any(false, false, false) = " << any(false, false, false) << endl;  
   
 return 0;  
}

Fold expressions come in four forms: 1. **Unary right fold**: (... op pack) 2. **Unary left fold**: (pack op ...) 3. **Binary right fold**: (init op ... op pack) 4. **Binary left fold**: (pack op ... op init)

### 10.4.3 Perfect Forwarding with Variadic Templates

Variadic templates are often used with perfect forwarding to pass arguments to another function:

#include <iostream>  
#include <utility> // For std::forward  
#include <vector>  
#include <string>  
using namespace std;  
  
class Widget {  
public:  
 template<typename... Args>  
 explicit Widget(Args&&... args) {  
 cout << "Widget constructor with " << sizeof...(args) << " arguments" << endl;  
 }  
};  
  
// Factory function that forwards arguments to constructor  
template<typename... Args>  
Widget makeWidget(Args&&... args) {  
 return Widget(std::forward<Args>(args)...);  
}  
  
// Example with container emplace  
template<typename T, typename... Args>  
void emplace\_back\_wrapper(vector<T>& v, Args&&... args) {  
 cout << "Adding element with " << sizeof...(args) << " constructor arguments" << endl;  
 v.emplace\_back(std::forward<Args>(args)...);  
}  
  
int main() {  
 // Create Widgets with different arguments  
 Widget w1 = makeWidget();  
 Widget w2 = makeWidget(10);  
 Widget w3 = makeWidget("hello", 42, true);  
   
 // Use with vector  
 vector<string> strings;  
   
 // Add strings with different constructors  
 emplace\_back\_wrapper(strings, "Direct string");  
 emplace\_back\_wrapper(strings, 10, 'a'); // string(10, 'a')  
 emplace\_back\_wrapper(strings, "Hello"s + " world"); // string from expression  
   
 cout << "Vector contains:" << endl;  
 for (const auto& s : strings) {  
 cout << "\"" << s << "\" (length: " << s.length() << ")" << endl;  
 }  
   
 return 0;  
}

This pattern allows creating wrapper functions that preserve the perfect forwarding of arguments to the wrapped function.

### 10.4.4 Variadic Template Class

Template classes can also be variadic:

#include <iostream>  
#include <tuple>  
using namespace std;  
  
// Variadic class template  
template<typename... Types>  
class Tuple {  
public:  
 static constexpr size\_t size = sizeof...(Types);  
   
 Tuple() {  
 cout << "Created Tuple with " << size << " types" << endl;  
 }  
};  
  
// Partial specialization for an empty tuple  
template<>  
class Tuple<> {  
public:  
 static constexpr size\_t size = 0;  
   
 Tuple() {  
 cout << "Created empty Tuple" << endl;  
 }  
};  
  
// Recursive inheritance approach (common pre-C++17 technique)  
template<typename Head, typename... Tail>  
class RecursiveTuple {  
private:  
 Head head\_;  
 RecursiveTuple<Tail...> tail\_;  
   
public:  
 RecursiveTuple(Head head, Tail... tail)  
 : head\_(head), tail\_(tail...) {  
 cout << "Added element of type " << typeid(Head).name() << endl;  
 }  
   
 Head getHead() const { return head\_; }  
   
 template<size\_t I>  
 auto get() const {  
 if constexpr (I == 0)  
 return head\_;  
 else  
 return tail\_.template get<I-1>();  
 }  
};  
  
// Base case for recursive inheritance  
template<typename T>  
class RecursiveTuple<T> {  
private:  
 T value\_;  
   
public:  
 RecursiveTuple(T value) : value\_(value) {  
 cout << "Added final element of type " << typeid(T).name() << endl;  
 }  
   
 T getHead() const { return value\_; }  
   
 template<size\_t I>  
 T get() const {  
 static\_assert(I == 0, "Index out of bounds");  
 return value\_;  
 }  
};  
  
int main() {  
 // Basic examples  
 Tuple<> t0;  
 Tuple<int> t1;  
 Tuple<int, double, string> t3;  
   
 // Recursive implementation  
 RecursiveTuple<int, double, string> rt(42, 3.14, "hello");  
   
 cout << "First element: " << rt.getHead() << endl;  
 cout << "Elements by index:" << endl;  
 cout << " [0]: " << rt.get<0>() << endl;  
 cout << " [1]: " << rt.get<1>() << endl;  
 cout << " [2]: " << rt.get<2>() << endl;  
   
 // Using standard library tuple  
 tuple<int, double, string> std\_tuple(42, 3.14, "hello");  
 cout << "std::tuple elements:" << endl;  
 cout << " [0]: " << get<0>(std\_tuple) << endl;  
 cout << " [1]: " << get<1>(std\_tuple) << endl;  
 cout << " [2]: " << get<2>(std\_tuple) << endl;  
   
 return 0;  
}

### 10.4.5 Expanding Parameter Packs in Different Contexts

Parameter packs can be expanded in various contexts:

#include <iostream>  
#include <array>  
#include <vector>  
using namespace std;  
  
// Function call expansion  
template<typename... Args>  
void callFunctions(Args... funcs) {  
 // Call each function with no arguments  
 (funcs(), ...); // Fold expression (C++17)  
}  
  
// Array initialization  
template<typename T, typename... Args>  
auto makeArray(Args... args) {  
 return array<T, sizeof...(args)>{args...};  
}  
  
// Capture in lambda expressions  
template<typename... Args>  
auto captureAll(Args... args) {  
 return [args...] {  
 // Use fold expression to print  
 ((cout << args << " "), ...);  
 cout << endl;  
 };  
}  
  
// Expansion in template arguments  
template<typename... Ts>  
struct TypeList {};  
  
template<typename... Args>  
auto makeTypeList() {  
 return TypeList<Args...>{};  
}  
  
// Parameter pack in initializer lists  
template<typename T, typename... Args>  
vector<T> makeVector(Args... args) {  
 return {static\_cast<T>(args)...};  
}  
  
int main() {  
 // Test function call expansion  
 auto f1 = []{ cout << "Function 1 called" << endl; };  
 auto f2 = []{ cout << "Function 2 called" << endl; };  
 auto f3 = []{ cout << "Function 3 called" << endl; };  
   
 callFunctions(f1, f2, f3);  
   
 // Test array creation  
 auto arr = makeArray<double>(1, 2, 3, 4, 5);  
 cout << "Array contents: ";  
 for (auto v : arr) {  
 cout << v << " ";  
 }  
 cout << endl;  
   
 // Test lambda capture  
 auto printer = captureAll(10, "hello", 3.14, 'A');  
 cout << "Lambda output: ";  
 printer();  
   
 // Test vector creation with type conversion  
 auto vec = makeVector<double>(1, 2, 3, 4.5f, 5.5);  
 cout << "Vector contents: ";  
 for (auto v : vec) {  
 cout << v << " ";  
 }  
 cout << endl;  
   
 return 0;  
}

### 10.4.6 Variadic Templates Best Practices

1. **Use fold expressions in C++17 and later** for simpler code
2. **Use sizeof...(args) to get the number of arguments** in the parameter pack
3. **Always provide a base case for recursive variadic templates** (pre-C++17)
4. **Use std::forward for perfect forwarding** of arguments
5. **Use decltype and declval for complex type deductions**
6. **Consider readability** - variadic templates can become complex
7. **Use standard library variadic class templates** like std::tuple when appropriate
8. **Test with different numbers of arguments** including zero arguments

## 10.5 Template Metaprogramming

Template metaprogramming (TMP) is a technique that uses C++ templates to perform computations at compile time. It’s a form of meta-programming, where the program manipulates itself during compilation.

### 10.5.1 Introduction to Template Metaprogramming

Template metaprogramming involves: 1. **Compile-time computation**: Calculations performed during compilation 2. **Type computation**: Creating and manipulating types programmatically 3. **Code generation**: Generating specialized code based on types or values

Here’s a classic example - computing factorials at compile time:

#include <iostream>  
using namespace std;  
  
// Factorial template metaprogram  
template <unsigned int N>  
struct Factorial {  
 static constexpr unsigned int value = N \* Factorial<N - 1>::value;  
};  
  
// Base case  
template <>  
struct Factorial<0> {  
 static constexpr unsigned int value = 1;  
};  
  
int main() {  
 cout << "Factorial of 5: " << Factorial<5>::value << endl;  
 cout << "Factorial of 10: " << Factorial<10>::value << endl;  
   
 // This value is computed at compile time!  
 constexpr unsigned int fact5 = Factorial<5>::value;  
   
 // Create an array with size determined at compile time  
 array<int, Factorial<5>::value> data;  
 cout << "Array size: " << data.size() << endl;  
   
 return 0;  
}

In C++14 and later, you can also use constexpr functions for compile-time computation:

#include <iostream>  
using namespace std;  
  
// Constexpr factorial function (C++14 and later)  
constexpr unsigned int factorial(unsigned int n) {  
 return (n <= 1) ? 1 : n \* factorial(n - 1);  
}  
  
int main() {  
 // Computed at compile time  
 constexpr unsigned int fact5 = factorial(5);  
 constexpr unsigned int fact10 = factorial(10);  
   
 cout << "Factorial of 5: " << fact5 << endl;  
 cout << "Factorial of 10: " << fact10 << endl;  
   
 // Create a compile-time array  
 array<int, factorial(6)> data;  
 cout << "Array size: " << data.size() << endl;  
   
 return 0;  
}

### 10.5.2 Type Traits and Type Manipulation

Template metaprogramming is often used to inspect and manipulate types:

#include <iostream>  
#include <type\_traits>  
using namespace std;  
  
// Custom type trait to check if a type can be incremented  
template <typename T, typename = void>  
struct is\_incrementable : false\_type {};  
  
template <typename T>  
struct is\_incrementable<T,   
 void\_t<decltype(++declval<T&>())>> : true\_type {};  
  
// Helper variable template (C++14)  
template <typename T>  
inline constexpr bool is\_incrementable\_v = is\_incrementable<T>::value;  
  
// Compute result type of adding two types  
template <typename T, typename U>  
struct addition\_result {  
 using type = decltype(declval<T>() + declval<U>());  
};  
  
template <typename T, typename U>  
using addition\_result\_t = typename addition\_result<T, U>::type;  
  
int main() {  
 // Check if types are incrementable  
 cout << "int is incrementable: " << is\_incrementable\_v<int> << endl;  
 cout << "float is incrementable: " << is\_incrementable\_v<float> << endl;  
 cout << "string is incrementable: " << is\_incrementable\_v<string> << endl;  
 cout << "vector<int> is incrementable: " << is\_incrementable\_v<vector<int>> << endl;  
   
 // Test addition result  
 using IntPlusFloat = addition\_result\_t<int, float>;  
 cout << "Type of int + float is: " << typeid(IntPlusFloat).name() << endl;  
   
 // Use standard type traits  
 cout << "is\_integral<int>: " << is\_integral\_v<int> << endl;  
 cout << "is\_floating\_point<double>: " << is\_floating\_point\_v<double> << endl;  
 cout << "is\_same<int, long>: " << is\_same\_v<int, long> << endl;  
 cout << "is\_convertible<float, int>: " << is\_convertible\_v<float, int> << endl;  
   
 return 0;  
}

C++ provides many built-in type traits in the <type\_traits> header, which form a crucial foundation for template metaprogramming.

### 10.5.3 SFINAE (Substitution Failure Is Not An Error)

SFINAE is a key concept in template metaprogramming, allowing you to select function overloads based on type properties:

#include <iostream>  
#include <type\_traits>  
#include <vector>  
using namespace std;  
  
// Print size for containers that have a size() method  
template <typename T>  
auto printSize(const T& container)   
 -> decltype(container.size(), void()) {  
 cout << "Container size: " << container.size() << endl;  
}  
  
// Fallback for types without a size() method  
template <typename T>  
void printSize(...) {  
 cout << "Object has no size() method" << endl;  
}  
  
// SFINAE with enable\_if for more complex conditions  
template <typename T>  
typename enable\_if<is\_integral<T>::value, T>::type  
twice(T value) {  
 return value \* 2;  
}  
  
template <typename T>  
typename enable\_if<!is\_integral<T>::value && is\_floating\_point<T>::value, T>::type  
twice(T value) {  
 return value \* 2.0;  
}  
  
// C++20: Using concepts instead of SFINAE  
#ifdef \_\_cpp\_concepts  
template <typename T>  
requires is\_integral\_v<T>  
T twice\_concept(T value) {  
 return value \* 2;  
}  
  
template <typename T>  
requires (!is\_integral\_v<T> && is\_floating\_point\_v<T>)  
T twice\_concept(T value) {  
 return value \* 2.0;  
}  
#endif  
  
int main() {  
 vector<int> vec = {1, 2, 3};  
 string str = "hello";  
 int num = 42;  
   
 printSize(vec); // Has size() method  
 printSize(str); // Has size() method  
 printSize(num); // No size() method  
   
 cout << "twice(10): " << twice(10) << endl;  
 cout << "twice(3.14): " << twice(3.14) << endl;  
   
#ifdef \_\_cpp\_concepts  
 cout << "twice\_concept(10): " << twice\_concept(10) << endl;  
 cout << "twice\_concept(3.14): " << twice\_concept(3.14) << endl;  
#endif  
   
 return 0;  
}

SFINAE relies on the C++ rule that if template argument substitution leads to an invalid type or expression, the compiler simply removes that overload from consideration rather than treating it as an error.

### 10.5.4 Conditional Compilation and Tag Dispatching

Tag dispatching is a technique that allows you to select different implementations based on type properties:

#include <iostream>  
#include <type\_traits>  
#include <vector>  
#include <list>  
using namespace std;  
  
// Tag types for dispatching  
struct random\_access\_tag {};  
struct bidirectional\_tag {};  
struct forward\_tag {};  
  
// Tag selection based on iterator category  
template <typename Iter>  
struct iterator\_category\_tag {  
 using type = conditional\_t<  
 is\_same\_v<typename iterator\_traits<Iter>::iterator\_category, random\_access\_iterator\_tag>,  
 random\_access\_tag,  
 conditional\_t<  
 is\_same\_v<typename iterator\_traits<Iter>::iterator\_category, bidirectional\_iterator\_tag>,  
 bidirectional\_tag,  
 forward\_tag  
 >  
 >;  
};  
  
// Implementation for random access iterators  
template <typename Iter>  
void advance\_impl(Iter& it, int n, random\_access\_tag) {  
 cout << "Using fast random access advance" << endl;  
 it += n; // Random access iterators can do this efficiently  
}  
  
// Implementation for bidirectional iterators  
template <typename Iter>  
void advance\_impl(Iter& it, int n, bidirectional\_tag) {  
 cout << "Using bidirectional advance" << endl;  
 if (n >= 0) {  
 while (n--) ++it;  
 } else {  
 while (n++) --it; // Can go backwards efficiently  
 }  
}  
  
// Implementation for forward iterators  
template <typename Iter>  
void advance\_impl(Iter& it, int n, forward\_tag) {  
 cout << "Using forward-only advance" << endl;  
 if (n < 0) {  
 throw runtime\_error("Cannot move backward with forward iterator");  
 }  
 while (n--) ++it;  
}  
  
// Unified interface that dispatches to the right implementation  
template <typename Iter>  
void custom\_advance(Iter& it, int n) {  
 advance\_impl(it, n, typename iterator\_category\_tag<Iter>::type{});  
}  
  
int main() {  
 vector<int> vec = {1, 2, 3, 4, 5};  
 list<int> lst = {1, 2, 3, 4, 5};  
   
 auto vecIt = vec.begin();  
 auto lstIt = lst.begin();  
   
 custom\_advance(vecIt, 2); // Uses random access  
 custom\_advance(lstIt, 2); // Uses bidirectional  
   
 cout << "Vector iterator now points to: " << \*vecIt << endl;  
 cout << "List iterator now points to: " << \*lstIt << endl;  
   
 return 0;  
}

Tag dispatching selects the most appropriate implementation at compile time based on type characteristics, without code duplication or runtime checks.

### 10.5.5 Recursive Template Pattern Matching

Template specializations can be used for pattern matching at compile time:

#include <iostream>  
#include <tuple>  
using namespace std;  
  
// Tuple printer using recursive templates  
// Primary template for empty case  
template <typename Tuple, size\_t N = 0>  
struct TuplePrinter {  
 static void print(const Tuple& t) {  
 // Do nothing - we've reached the end of recursion  
 }  
};  
  
// Specialization that handles each element  
template <typename Tuple, size\_t N>  
struct TuplePrinter<Tuple, N> {  
 static void print(const Tuple& t) {  
 // Base case: we're at the last element  
 if constexpr (N == tuple\_size\_v<Tuple> - 1) {  
 cout << get<N>(t);  
 } else {  
 // Recursive case: print this element and continue  
 cout << get<N>(t) << ", ";  
 TuplePrinter<Tuple, N+1>::print(t);  
 }  
 }  
};  
  
// User-friendly wrapper  
template <typename... Args>  
void printTuple(const tuple<Args...>& t) {  
 cout << "(";  
 TuplePrinter<tuple<Args...>, 0>::print(t);  
 cout << ")" << endl;  
}  
  
// Another example: compile-time list processing  
template <typename... Types>  
struct TypeList {};  
  
// Get head (first type) of a TypeList  
template <typename Head, typename... Tail>  
struct GetHead {  
 using type = Head;  
};  
  
template <typename... Types>  
using Head\_t = typename GetHead<Types...>::type;  
  
// Get tail (all but first) of a TypeList  
template <typename Head, typename... Tail>  
struct GetTail {  
 using type = TypeList<Tail...>;  
};  
  
template <typename... Types>  
using Tail\_t = typename GetTail<Types...>::type;  
  
// Is the TypeList empty?  
template <typename... Types>  
struct IsEmpty {  
 static constexpr bool value = sizeof...(Types) == 0;  
};  
  
template <typename... Types>  
inline constexpr bool IsEmpty\_v = IsEmpty<Types...>::value;  
  
int main() {  
 // Test tuple printer  
 tuple<int, string, double, char> t(42, "hello", 3.14, 'A');  
 printTuple(t);  
   
 // Test TypeList operations  
 using MyTypes = TypeList<int, double, string, vector<int>>;  
   
 // Extract some type information at compile time  
 using FirstType = Head\_t<int, double, string>;  
 using RemainingTypes = Tail\_t<int, double, string>;  
   
 cout << "IsEmpty<>: " << IsEmpty\_v<> << endl;  
 cout << "IsEmpty<int>: " << IsEmpty\_v<int> << endl;  
   
 cout << "First type is: " << typeid(FirstType).name() << endl;  
   
 return 0;  
}

This pattern is particularly useful for processing complex types at compile time.

### 10.5.6 Compile-Time if with constexpr (C++17)

C++17’s if constexpr greatly simplifies many template metaprogramming patterns:

#include <iostream>  
#include <type\_traits>  
#include <string>  
#include <vector>  
using namespace std;  
  
// Generic print function using if constexpr  
template <typename T>  
void smartPrint(const T& value) {  
 if constexpr (is\_same\_v<T, string>) {  
 cout << "String: \"" << value << "\"" << endl;  
 }  
 else if constexpr (is\_integral\_v<T>) {  
 cout << "Integer: " << value << " (0x" << hex << value << dec << ")" << endl;  
 }  
 else if constexpr (is\_floating\_point\_v<T>) {  
 cout << "Float: " << fixed << value << endl;  
 }  
 else if constexpr (requires { value.size(); }) {  
 cout << "Container with " << value.size() << " elements: ";  
 for (const auto& elem : value) {  
 cout << elem << " ";  
 }  
 cout << endl;  
 }  
 else {  
 cout << "Generic value: " << value << endl;  
 }  
}  
  
// Compile-time recursion with if constexpr  
template <typename Tuple, size\_t... I>  
void printTupleImpl(const Tuple& t, index\_sequence<I...>) {  
 // Using fold expression with if constexpr  
 cout << "(";  
 ((cout << (I == 0 ? "" : ", ") << get<I>(t)), ...);  
 cout << ")" << endl;  
}  
  
template <typename... Args>  
void printTuple2(const tuple<Args...>& t) {  
 printTupleImpl(t, make\_index\_sequence<sizeof...(Args)>{});  
}  
  
int main() {  
 smartPrint(42);  
 smartPrint(3.14159);  
 smartPrint("raw c-string");  
 smartPrint(string("std::string"));  
   
 vector<int> vec = {1, 2, 3, 4, 5};  
 smartPrint(vec);  
   
 // Test improved tuple printer  
 tuple<int, string, double, bool> t(42, "hello", 3.14, true);  
 printTuple2(t);  
   
 return 0;  
}

if constexpr evaluates the condition at compile time and only instantiates the selected branch, avoiding compilation errors that would occur in unselected branches.

### 10.5.7 Compile-Time Programming with C++20 Concepts

C++20 concepts formalize template constraints and can be used in metaprogramming:

#include <iostream>  
#include <type\_traits>  
#include <concepts>  
using namespace std;  
  
// Define a concept for arithmetic types  
template <typename T>  
concept Arithmetic = is\_arithmetic\_v<T>;  
  
// Define a concept for container types  
template <typename T>  
concept Container = requires(T container) {  
 { container.begin() } -> std::input\_or\_output\_iterator;  
 { container.end() } -> std::input\_or\_output\_iterator;  
 { container.size() } -> std::convertible\_to<size\_t>;  
};  
  
// Define a concept for printable types  
template <typename T>  
concept Printable = requires(T x, ostream& os) {  
 { os << x } -> same\_as<ostream&>;  
};  
  
// Function that works only with arithmetic types  
template <Arithmetic T>  
T square(T value) {  
 return value \* value;  
}  
  
// Function that works with containers  
template <Container T>  
auto sum(const T& container) {  
 using value\_type = decay\_t<decltype(\*begin(container))>;  
 value\_type result{};  
 for (const auto& item : container) {  
 result += item;  
 }  
 return result;  
}  
  
// Function that prints anything printable  
template <Printable T>  
void print(const T& value) {  
 cout << value << endl;  
}  
  
int main() {  
 // Test with arithmetic types  
 cout << "square(7) = " << square(7) << endl;  
 cout << "square(3.14) = " << square(3.14) << endl;  
   
 // Test with containers  
 vector<int> vec = {1, 2, 3, 4, 5};  
 cout << "sum(vec) = " << sum(vec) << endl;  
   
 array<double, 3> arr = {1.1, 2.2, 3.3};  
 cout << "sum(arr) = " << sum(arr) << endl;  
   
 // Test with printable types  
 print("Hello, concepts!");  
 print(42);  
 print(vector<int>{1, 2, 3});  
   
 return 0;  
}

### 10.5.8 Real-World Applications of Template Metaprogramming

TMP is used in many practical applications:

1. **Expression templates**:

#include <iostream>  
using namespace std;  
  
// Forward declarations  
template <typename T> class Expression;  
template <typename L, typename R> class Addition;  
  
// Base expression template  
template <typename Derived>  
class Expression {  
public:  
 // Cast this to the derived type  
 const Derived& derived() const {  
 return static\_cast<const Derived&>(\*this);  
 }  
   
 // Evaluate the expression  
 double evaluate() const {  
 return derived().evaluate();  
 }  
};  
  
// Scalar value expression  
template <typename T>  
class Scalar : public Expression<Scalar<T>> {  
private:  
 T value\_;  
   
public:  
 explicit Scalar(T value) : value\_(value) {}  
   
 double evaluate() const {  
 return value\_;  
 }  
};  
  
// Addition expression  
template <typename L, typename R>  
class Addition : public Expression<Addition<L, R>> {  
private:  
 const L& lhs\_;  
 const R& rhs\_;  
   
public:  
 Addition(const L& lhs, const R& rhs) : lhs\_(lhs), rhs\_(rhs) {}  
   
 double evaluate() const {  
 return lhs\_.evaluate() + rhs\_.evaluate();  
 }  
};  
  
// Operator overload to create expression templates  
template <typename L, typename R>  
Addition<L, R> operator+(const Expression<L>& lhs, const Expression<R>& rhs) {  
 return Addition<L, R>(lhs.derived(), rhs.derived());  
}  
  
int main() {  
 // Create expression: 2 + 3 + 4  
 Scalar<int> a(2), b(3), c(4);  
 auto expr = a + b + c;  
   
 // Evaluate the expression  
 cout << "Result of 2 + 3 + 4 = " << expr.evaluate() << endl;  
   
 // More complex expression: (2 + 3) + (4 + 5)  
 Scalar<int> d(5);  
 auto expr2 = (a + b) + (c + d);  
 cout << "Result of (2 + 3) + (4 + 5) = " << expr2.evaluate() << endl;  
   
 return 0;  
}

1. **Static reflection**:

#include <iostream>  
#include <string>  
#include <tuple>  
using namespace std;  
  
// Simple struct for demonstration  
struct Person {  
 string name;  
 int age;  
 double height;  
};  
  
// Define a reflection helper  
#define REFLECT\_STRUCT\_BEGIN(Type) \  
 template <> struct Reflector<Type> { \  
 using type = Type; \  
 static constexpr auto get\_members() { \  
 return make\_tuple(  
  
#define REFLECT\_MEMBER(name) \  
 Member{#name, &type::name},  
  
#define REFLECT\_STRUCT\_END() \  
 ); \  
 } \  
 };  
  
// Member information  
struct Member {  
 string\_view name;  
 void\* ptr;  
   
 template <typename T, typename U>  
 Member(string\_view name, U T::\* member\_ptr)   
 : name(name), ptr(reinterpret\_cast<void\*>(&member\_ptr)) {}  
};  
  
// Primary template  
template <typename T>  
struct Reflector {};  
  
// Apply reflection to Person struct  
REFLECT\_STRUCT\_BEGIN(Person)  
 REFLECT\_MEMBER(name)  
 REFLECT\_MEMBER(age)  
 REFLECT\_MEMBER(height)  
REFLECT\_STRUCT\_END()  
  
// Generic function to print any reflected struct  
template <typename T>  
void print\_struct(const T& obj) {  
 cout << "Struct of type " << typeid(T).name() << " {\n";  
   
 const auto members = Reflector<T>::get\_members();  
 apply([&obj](const auto&... members) {  
 ((cout << " " << members.name << ": " <<   
 get\_member\_value(obj, members) << "\n"), ...);  
 }, members);  
   
 cout << "}" << endl;  
}  
  
// Helper to get member value (simplified)  
template <typename T, typename Member>  
auto get\_member\_value(const T& obj, const Member& member) {  
 // This is a simplified implementation - real reflection would be more complex  
 if (member.name == "name")  
 return obj.name;  
 else if (member.name == "age")  
 return obj.age;  
 else if (member.name == "height")  
 return obj.height;  
 else  
 return string("unknown");  
}  
  
int main() {  
 Person p{"John Doe", 30, 1.75};  
 print\_struct(p);  
   
 return 0;  
}

Note: True C++ reflection is still evolving, with proposals being considered for standardization. The above is a simplified example.

1. **Policy-based design**:

#include <iostream>  
#include <vector>  
#include <list>  
#include <algorithm>  
using namespace std;  
  
// Sorting policies  
struct QuickSortPolicy {  
 template <typename Iterator>  
 static void sort(Iterator begin, Iterator end) {  
 cout << "Using QuickSort policy" << endl;  
 std::sort(begin, end); // std::sort uses quicksort-like algorithm  
 }  
};  
  
struct BubbleSortPolicy {  
 template <typename Iterator>  
 static void sort(Iterator begin, Iterator end) {  
 cout << "Using BubbleSort policy" << endl;  
 // Simple bubble sort implementation  
 auto n = distance(begin, end);  
 for (auto i = 0; i < n - 1; ++i) {  
 bool swapped = false;  
 for (auto j = 0; j < n - i - 1; ++j) {  
 auto current = begin;  
 advance(current, j);  
 auto next = current;  
 advance(next, 1);  
 if (\*current > \*next) {  
 iter\_swap(current, next);  
 swapped = true;  
 }  
 }  
 if (!swapped) break;  
 }  
 }  
};  
  
// Storage policies  
struct VectorStorage {  
 template <typename T>  
 using container\_type = vector<T>;  
   
 static constexpr const char\* name() { return "Vector Storage"; }  
};  
  
struct ListStorage {  
 template <typename T>  
 using container\_type = list<T>;  
   
 static constexpr const char\* name() { return "List Storage"; }  
};  
  
// Container class that uses policies  
template <  
 typename T,  
 typename StoragePolicy = VectorStorage,  
 typename SortPolicy = QuickSortPolicy  
>  
class SortedContainer {  
public:  
 using container\_t = typename StoragePolicy::template container\_type<T>;  
   
 void add(const T& value) {  
 data\_.push\_back(value);  
 }  
   
 void sort() {  
 cout << "Sorting " << StoragePolicy::name() << " using ";  
 SortPolicy::sort(data\_.begin(), data\_.end());  
 }  
   
 void display() const {  
 cout << "Container contents: ";  
 for (const auto& item : data\_) {  
 cout << item << " ";  
 }  
 cout << endl;  
 }  
   
private:  
 container\_t data\_;  
};  
  
int main() {  
 // Different container configurations using policies  
 SortedContainer<int> default\_container;  
 default\_container.add(3);  
 default\_container.add(1);  
 default\_container.add(4);  
 default\_container.add(2);  
   
 cout << "Default container: " << endl;  
 default\_container.sort();  
 default\_container.display();  
   
 // Container with list storage and bubble sort  
 SortedContainer<int, ListStorage, BubbleSortPolicy> custom\_container;  
 custom\_container.add(3);  
 custom\_container.add(1);  
 custom\_container.add(4);  
 custom\_container.add(2);  
   
 cout << "\nCustom container: " << endl;  
 custom\_container.sort();  
 custom\_container.display();  
   
 return 0;  
}

### 10.5.9 Template Metaprogramming Best Practices

1. **Use modern C++ features** like constexpr, if constexpr, and concepts when available
2. **Avoid excessive complexity** - TMP can become very difficult to understand
3. **Document your metaprogramming code thoroughly** - explain the intent and techniques
4. **Use standard library type traits** rather than reinventing them
5. **Use meaningful names** for template parameters and metafunctions
6. **Provide user-friendly wrappers** around complex metaprogramming
7. **Limit compile-time recursion depth** to avoid compiler limits
8. **Consider compile time** - heavy TMP can slow down compilation significantly
9. **Use static\_assert** to provide clear error messages for invalid usage
10. **Test both positive and negative cases** to ensure correct behavior

# Chapter 10: Templates in C++ (Part 3)

## 10.6 Template Applications and Advanced Topics

Templates in C++ provide powerful mechanisms for generic programming and code generation. This section explores practical applications and advanced techniques for using templates in real-world scenarios.

### 10.6.1 Smart Pointers Implementation

Smart pointers are one of the most important applications of templates in C++. They provide automatic memory management while maintaining the efficiency of raw pointers. Let’s explore how they’re implemented:

#include <iostream>  
#include <memory>  
#include <utility>  
  
// Simple implementation of a unique\_ptr-like smart pointer  
template <typename T>  
class UniquePtr {  
private:  
 T\* ptr;  
  
public:  
 // Constructor  
 explicit UniquePtr(T\* p = nullptr) : ptr(p) {  
 std::cout << "UniquePtr constructor called" << std::endl;  
 }  
  
 // Destructor  
 ~UniquePtr() {  
 std::cout << "UniquePtr destructor called" << std::endl;  
 if (ptr) {  
 delete ptr;  
 }  
 }  
  
 // Delete copy constructor and assignment operator  
 UniquePtr(const UniquePtr&) = delete;  
 UniquePtr& operator=(const UniquePtr&) = delete;  
  
 // Move constructor  
 UniquePtr(UniquePtr&& other) noexcept : ptr(other.ptr) {  
 std::cout << "UniquePtr move constructor called" << std::endl;  
 other.ptr = nullptr;  
 }  
  
 // Move assignment operator  
 UniquePtr& operator=(UniquePtr&& other) noexcept {  
 std::cout << "UniquePtr move assignment called" << std::endl;  
 if (this != &other) {  
 if (ptr) {  
 delete ptr;  
 }  
 ptr = other.ptr;  
 other.ptr = nullptr;  
 }  
 return \*this;  
 }  
  
 // Accessor methods  
 T\* get() const { return ptr; }  
   
 T& operator\*() const {  
 return \*ptr;  
 }  
   
 T\* operator->() const {  
 return ptr;  
 }  
   
 // Release ownership  
 T\* release() {  
 T\* temp = ptr;  
 ptr = nullptr;  
 return temp;  
 }  
   
 // Reset with a new pointer  
 void reset(T\* p = nullptr) {  
 if (ptr) {  
 delete ptr;  
 }  
 ptr = p;  
 }  
};  
  
// Example usage  
int main() {  
 // Create a smart pointer to an integer  
 UniquePtr<int> p1(new int(42));  
 std::cout << "p1 value: " << \*p1 << std::endl;  
   
 // Move ownership  
 UniquePtr<int> p2 = std::move(p1);  
 std::cout << "p2 value: " << \*p2 << std::endl;  
   
 // Create a smart pointer to a class  
 class Test {  
 public:  
 Test() { std::cout << "Test constructor" << std::endl; }  
 ~Test() { std::cout << "Test destructor" << std::endl; }  
 void hello() { std::cout << "Hello from Test" << std::endl; }  
 };  
   
 UniquePtr<Test> p3(new Test());  
 p3->hello(); // Using arrow operator  
   
 // Reset with a new object  
 p3.reset(new Test());  
   
 return 0; // All objects automatically cleaned up  
}

Key concepts in smart pointer implementation: 1. **Type parameterization**: The template parameter T allows the smart pointer to work with any type. 2. **RAII principle**: Resources are acquired in the constructor and released in the destructor. 3. **Move semantics**: Ownership can be transferred efficiently between smart pointers. 4. **Deleted copy operations**: Prevents multiple pointers from owning the same resource. 5. **Operator overloading**: Makes smart pointers behave like raw pointers.

### 10.6.2 Expression Templates

Expression templates are an advanced technique used to optimize operations on complex expressions by deferring evaluation and avoiding unnecessary temporaries:

#include <iostream>  
  
// Forward declarations  
template<typename T> class Scalar;  
template<typename L, typename R> class Addition;  
template<typename L, typename R> class Multiplication;  
  
// Base expression template class  
template<typename Derived>  
class Expression {  
public:  
 // Cast this expression to its derived type  
 const Derived& derived() const {  
 return static\_cast<const Derived&>(\*this);  
 }  
   
 // Evaluate the expression (to be implemented by derived classes)  
 double evaluate() const {  
 return derived().evaluate();  
 }  
};  
  
// Scalar expression (a terminal node in expression tree)  
template<typename T>  
class Scalar : public Expression<Scalar<T>> {  
private:  
 T value;  
   
public:  
 explicit Scalar(T val) : value(val) {}  
   
 // Evaluation of a scalar is just its value  
 double evaluate() const {  
 return static\_cast<double>(value);  
 }  
};  
  
// Addition expression: L + R  
template<typename L, typename R>  
class Addition : public Expression<Addition<L, R>> {  
private:  
 const L& left;  
 const R& right;  
   
public:  
 Addition(const L& l, const R& r) : left(l), right(r) {}  
   
 // Evaluate by adding the evaluated operands  
 double evaluate() const {  
 return left.evaluate() + right.evaluate();  
 }  
};  
  
// Multiplication expression: L \* R  
template<typename L, typename R>  
class Multiplication : public Expression<Multiplication<L, R>> {  
private:  
 const L& left;  
 const R& right;  
   
public:  
 Multiplication(const L& l, const R& r) : left(l), right(r) {}  
   
 // Evaluate by multiplying the evaluated operands  
 double evaluate() const {  
 return left.evaluate() \* right.evaluate();  
 }  
};  
  
// Operator overloads for creating expression templates  
  
// Addition of two expressions  
template<typename L, typename R>  
Addition<L, R> operator+(const Expression<L>& left, const Expression<R>& right) {  
 return Addition<L, R>(left.derived(), right.derived());  
}  
  
// Multiplication of two expressions  
template<typename L, typename R>  
Multiplication<L, R> operator\*(const Expression<L>& left, const Expression<R>& right) {  
 return Multiplication<L, R>(left.derived(), right.derived());  
}  
  
// Vector class that uses expression templates  
template<typename T, int Size>  
class Vector : public Expression<Vector<T, Size>> {  
private:  
 T data[Size];  
   
public:  
 // Default constructor  
 Vector() {  
 for (int i = 0; i < Size; ++i) {  
 data[i] = T(0);  
 }  
 }  
   
 // Copy constructor from another vector  
 Vector(const Vector<T, Size>& other) {  
 for (int i = 0; i < Size; ++i) {  
 data[i] = other.data[i];  
 }  
 }  
   
 // Constructor from expression  
 template<typename Expr>  
 Vector(const Expression<Expr>& expr) {  
 const Expr& e = expr.derived();  
 for (int i = 0; i < Size; ++i) {  
 data[i] = e[i]; // Assumes expression supports indexing  
 }  
 }  
   
 // Assignment from expression  
 template<typename Expr>  
 Vector<T, Size>& operator=(const Expression<Expr>& expr) {  
 const Expr& e = expr.derived();  
 for (int i = 0; i < Size; ++i) {  
 data[i] = e[i];  
 }  
 return \*this;  
 }  
   
 // Access elements  
 T& operator[](int index) {  
 return data[index];  
 }  
   
 const T& operator[](int index) const {  
 return data[index];  
 }  
   
 // Evaluate (for expression template interface)  
 double evaluate() const {  
 double sum = 0;  
 for (int i = 0; i < Size; ++i) {  
 sum += data[i];  
 }  
 return sum;  
 }  
   
 // Display vector contents  
 void print() const {  
 std::cout << "[";  
 for (int i = 0; i < Size; ++i) {  
 std::cout << data[i];  
 if (i < Size - 1) std::cout << ", ";  
 }  
 std::cout << "]" << std::endl;  
 }  
};  
  
// Vector addition expression  
template<typename L, typename R, int Size>  
class VectorAddition : public Expression<VectorAddition<L, R, Size>> {  
private:  
 const L& left;  
 const R& right;  
   
public:  
 VectorAddition(const L& l, const R& r) : left(l), right(r) {}  
   
 // Access element at index (for efficient evaluation)  
 double operator[](int index) const {  
 return left[index] + right[index];  
 }  
   
 // Evaluate the entire expression  
 double evaluate() const {  
 double sum = 0;  
 for (int i = 0; i < Size; ++i) {  
 sum += (\*this)[i];  
 }  
 return sum;  
 }  
};  
  
// Overload + for vectors  
template<typename T, int Size>  
VectorAddition<Vector<T, Size>, Vector<T, Size>, Size> operator+(  
 const Vector<T, Size>& left, const Vector<T, Size>& right) {  
 return VectorAddition<Vector<T, Size>, Vector<T, Size>, Size>(left, right);  
}  
  
int main() {  
 // Scalar expressions  
 Scalar<int> a(10), b(20), c(30);  
 auto expr = a + b \* c;  
 std::cout << "Result of a + b \* c: " << expr.evaluate() << std::endl;  
   
 // Vector expressions  
 Vector<double, 3> v1, v2, v3;  
   
 // Initialize vectors  
 v1[0] = 1.0; v1[1] = 2.0; v1[2] = 3.0;  
 v2[0] = 4.0; v2[1] = 5.0; v2[2] = 6.0;  
   
 // The following will create an expression object, not perform calculations yet  
 auto vecExpr = v1 + v2;  
   
 // Assign expression to v3, which evaluates the expression  
 v3 = vecExpr;  
   
 // Display results  
 std::cout << "v1: "; v1.print();  
 std::cout << "v2: "; v2.print();  
 std::cout << "v3 = v1 + v2: "; v3.print();  
   
 return 0;  
}

Key benefits of expression templates: 1. **Lazy evaluation**: Operations are performed only when needed. 2. **Elimination of temporaries**: Intermediate results aren’t stored in memory. 3. **Loop fusion**: Operations that would require multiple loops can be combined into one. 4. **Domain-specific optimizations**: The compile-time expression tree can be analyzed and optimized.

### 10.6.3 Template-Based Design Patterns

Templates enable powerful implementations of design patterns with compile-time flexibility. Here are some common patterns implemented with templates:

#### Factory Pattern

#include <iostream>  
#include <string>  
#include <map>  
#include <memory>  
#include <functional>  
  
// Base product class  
class Product {  
public:  
 virtual ~Product() = default;  
 virtual void use() const = 0;  
};  
  
// Concrete products  
class ConcreteProductA : public Product {  
public:  
 void use() const override {  
 std::cout << "Using ConcreteProductA" << std::endl;  
 }  
};  
  
class ConcreteProductB : public Product {  
public:  
 void use() const override {  
 std::cout << "Using ConcreteProductB" << std::endl;  
 }  
};  
  
// Template factory  
template<typename ProductType>  
class DefaultFactoryPolicy {  
public:  
 static std::unique\_ptr<ProductType> create() {  
 return std::make\_unique<ProductType>();  
 }  
};  
  
// Specialized factory policy  
template<>  
class DefaultFactoryPolicy<ConcreteProductB> {  
public:  
 static std::unique\_ptr<ConcreteProductB> create() {  
 // Special initialization for ProductB  
 auto product = std::make\_unique<ConcreteProductB>();  
 std::cout << "Special initialization for ConcreteProductB" << std::endl;  
 return product;  
 }  
};  
  
// Factory class using policy-based design  
template<typename ProductBaseType>  
class Factory {  
private:  
 using CreatorFunc = std::function<std::unique\_ptr<ProductBaseType>()>;  
 std::map<std::string, CreatorFunc> creators;  
  
public:  
 template<typename ConcreteProduct, typename FactoryPolicy = DefaultFactoryPolicy<ConcreteProduct>>  
 void registerProduct(const std::string& id) {  
 creators[id] = []() { return FactoryPolicy::create(); };  
 }  
  
 std::unique\_ptr<ProductBaseType> createProduct(const std::string& id) {  
 auto it = creators.find(id);  
 if (it != creators.end()) {  
 return it->second();  
 }  
 return nullptr;  
 }  
};  
  
int main() {  
 Factory<Product> factory;  
   
 // Register products with the factory  
 factory.registerProduct<ConcreteProductA>("A");  
 factory.registerProduct<ConcreteProductB>("B");  
   
 // Create products  
 auto productA = factory.createProduct("A");  
 auto productB = factory.createProduct("B");  
   
 if (productA) productA->use();  
 if (productB) productB->use();  
   
 // Try to create an unregistered product  
 auto unknownProduct = factory.createProduct("C");  
 if (unknownProduct) {  
 unknownProduct->use();  
 } else {  
 std::cout << "Unknown product ID" << std::endl;  
 }  
   
 return 0;  
}

#### Singleton Pattern

#include <iostream>  
#include <string>  
  
// Template-based singleton pattern  
template<typename T>  
class Singleton {  
private:  
 static T\* instance;  
   
 // Make constructors private to prevent external instantiation  
 Singleton(const Singleton&) = delete;  
 Singleton& operator=(const Singleton&) = delete;  
   
protected:  
 Singleton() = default;  
 virtual ~Singleton() = default;  
   
public:  
 static T& getInstance() {  
 if (!instance) {  
 instance = new T();  
 }  
 return \*instance;  
 }  
   
 static void destroyInstance() {  
 delete instance;  
 instance = nullptr;  
 }  
};  
  
// Initialize static member  
template<typename T>  
T\* Singleton<T>::instance = nullptr;  
  
// Example usage with a configuration manager  
class ConfigManager : public Singleton<ConfigManager> {  
 friend class Singleton<ConfigManager>; // Allow Singleton to access private constructor  
   
private:  
 std::string appName;  
 bool debugMode;  
   
 ConfigManager() : appName("Default App"), debugMode(false) {  
 std::cout << "ConfigManager created" << std::endl;  
 }  
   
 ~ConfigManager() {  
 std::cout << "ConfigManager destroyed" << std::endl;  
 }  
   
public:  
 void setAppName(const std::string& name) { appName = name; }  
 std::string getAppName() const { return appName; }  
   
 void setDebugMode(bool debug) { debugMode = debug; }  
 bool isDebugMode() const { return debugMode; }  
   
 void displayConfig() const {  
 std::cout << "Application: " << appName << std::endl;  
 std::cout << "Debug Mode: " << (debugMode ? "enabled" : "disabled") << std::endl;  
 }  
};  
  
// Another singleton example  
class Logger : public Singleton<Logger> {  
 friend class Singleton<Logger>; // Allow Singleton to access private constructor  
   
private:  
 bool enabled;  
   
 Logger() : enabled(true) {  
 std::cout << "Logger created" << std::endl;  
 }  
   
 ~Logger() {  
 std::cout << "Logger destroyed" << std::endl;  
 }  
   
public:  
 void log(const std::string& message) {  
 if (enabled) {  
 std::cout << "LOG: " << message << std::endl;  
 }  
 }  
   
 void setEnabled(bool enable) { enabled = enable; }  
 bool isEnabled() const { return enabled; }  
};  
  
int main() {  
 // Get the ConfigManager instance  
 ConfigManager& config = ConfigManager::getInstance();  
 config.setAppName("My Application");  
 config.setDebugMode(true);  
 config.displayConfig();  
   
 // Get the Logger instance  
 Logger& logger = Logger::getInstance();  
 logger.log("Application started");  
   
 // Both refer to the same instances  
 ConfigManager& config2 = ConfigManager::getInstance();  
 Logger& logger2 = Logger::getInstance();  
   
 std::cout << "Same config instance: " << (&config == &config2 ? "yes" : "no") << std::endl;  
 std::cout << "Same logger instance: " << (&logger == &logger2 ? "yes" : "no") << std::endl;  
   
 // Clean up (in real applications, singletons often live until program termination)  
 ConfigManager::destroyInstance();  
 Logger::destroyInstance();  
   
 return 0;  
}

#### Observer Pattern

#include <iostream>  
#include <vector>  
#include <memory>  
#include <algorithm>  
  
// Forward declaration  
template<typename T> class Subject;  
  
// Observer template base class  
template<typename T>  
class Observer {  
public:  
 virtual ~Observer() = default;  
 virtual void update(const T& data) = 0;  
   
 // Auto-register with subject  
 void observeSubject(Subject<T>\* subject) {  
 subject->addObserver(this);  
 subjects.push\_back(subject);  
 }  
   
 // Clean up registrations when destroyed  
 void cleanup() {  
 for (auto subject : subjects) {  
 subject->removeObserver(this);  
 }  
 subjects.clear();  
 }  
   
private:  
 std::vector<Subject<T>\*> subjects;  
};  
  
// Subject template base class  
template<typename T>  
class Subject {  
public:  
 void addObserver(Observer<T>\* observer) {  
 observers.push\_back(observer);  
 }  
   
 void removeObserver(Observer<T>\* observer) {  
 observers.erase(  
 std::remove(observers.begin(), observers.end(), observer),  
 observers.end()  
 );  
 }  
   
 void notifyObservers(const T& data) {  
 for (auto observer : observers) {  
 observer->update(data);  
 }  
 }  
   
private:  
 std::vector<Observer<T>\*> observers;  
};  
  
// Example usage with temperature monitoring  
struct WeatherData {  
 double temperature;  
 double humidity;  
 double pressure;  
   
 WeatherData(double t = 0.0, double h = 0.0, double p = 0.0)  
 : temperature(t), humidity(h), pressure(p) {}  
};  
  
// Concrete subject  
class WeatherStation : public Subject<WeatherData> {  
private:  
 WeatherData currentData;  
   
public:  
 void setMeasurements(double temperature, double humidity, double pressure) {  
 currentData = WeatherData(temperature, humidity, pressure);  
 notifyObservers(currentData);  
 }  
   
 WeatherData getCurrentData() const {  
 return currentData;  
 }  
};  
  
// Concrete observers  
class TemperatureDisplay : public Observer<WeatherData> {  
public:  
 TemperatureDisplay() = default;  
   
 ~TemperatureDisplay() {  
 cleanup(); // Clean up observer registrations  
 }  
   
 void update(const WeatherData& data) override {  
 std::cout << "Temperature Display: " << data.temperature << "°C" << std::endl;  
 }  
};  
  
class WeatherStatsTracker : public Observer<WeatherData> {  
private:  
 double minTemp = 1000.0;  
 double maxTemp = -1000.0;  
 double tempSum = 0.0;  
 int numReadings = 0;  
   
public:  
 WeatherStatsTracker() = default;  
   
 ~WeatherStatsTracker() {  
 cleanup(); // Clean up observer registrations  
 }  
   
 void update(const WeatherData& data) override {  
 double temp = data.temperature;  
   
 minTemp = std::min(minTemp, temp);  
 maxTemp = std::max(maxTemp, temp);  
 tempSum += temp;  
 ++numReadings;  
   
 std::cout << "Weather Stats:" << std::endl;  
 std::cout << " Min temperature: " << minTemp << "°C" << std::endl;  
 std::cout << " Max temperature: " << maxTemp << "°C" << std::endl;  
 std::cout << " Average temperature: " << (tempSum / numReadings) << "°C" << std::endl;  
 }  
};  
  
int main() {  
 WeatherStation weatherStation;  
   
 auto tempDisplay = std::make\_unique<TemperatureDisplay>();  
 auto statsTracker = std::make\_unique<WeatherStatsTracker>();  
   
 // Register observers with the subject  
 tempDisplay->observeSubject(&weatherStation);  
 statsTracker->observeSubject(&weatherStation);  
   
 // Simulate weather changes  
 std::cout << "Weather update 1:" << std::endl;  
 weatherStation.setMeasurements(25.2, 65.0, 1013.1);  
   
 std::cout << "\nWeather update 2:" << std::endl;  
 weatherStation.setMeasurements(26.8, 70.0, 1012.5);  
   
 std::cout << "\nWeather update 3:" << std::endl;  
 weatherStation.setMeasurements(24.5, 80.0, 1010.3);  
   
 return 0;  
}

### 10.6.4 CRTP (Curiously Recurring Template Pattern)

The Curiously Recurring Template Pattern (CRTP) is a C++ idiom where a class derives from a template class that takes the derived class as a template parameter:

#include <iostream>  
#include <string>  
#include <memory>  
  
// Base class template using CRTP  
template<typename Derived>  
class Base {  
public:  
 void interface() {  
 // Call the implementation in the derived class  
 static\_cast<Derived\*>(this)->implementation();  
 }  
   
 // Default implementation that can be overridden  
 void implementation() {  
 std::cout << "Base implementation" << std::endl;  
 }  
   
 // Static polymorphism example  
 void doSomething() {  
 std::cout << "Doing pre-work in base..." << std::endl;  
 static\_cast<Derived\*>(this)->implementation();  
 std::cout << "Doing post-work in base..." << std::endl;  
 }  
};  
  
// Derived class using CRTP  
class Derived1 : public Base<Derived1> {  
public:  
 // Override implementation  
 void implementation() {  
 std::cout << "Derived1 implementation" << std::endl;  
 }  
};  
  
// Another derived class  
class Derived2 : public Base<Derived2> {  
public:  
 // Override implementation  
 void implementation() {  
 std::cout << "Derived2 implementation" << std::endl;  
 }  
};  
  
// Example of CRTP for static polymorphism with countable mixin  
template<typename Derived>  
class ObjectCounter {  
private:  
 static inline size\_t count = 0; // C++17 inline static  
  
protected:  
 ObjectCounter() {  
 ++count;  
 }  
   
 ~ObjectCounter() {  
 --count;  
 }  
   
public:  
 static size\_t getCount() {  
 return count;  
 }  
};  
  
// Classes that want to be counted  
class Widget : public ObjectCounter<Widget> {  
public:  
 Widget() { std::cout << "Widget created" << std::endl; }  
 ~Widget() { std::cout << "Widget destroyed" << std::endl; }  
};  
  
class Gadget : public ObjectCounter<Gadget> {  
public:  
 Gadget() { std::cout << "Gadget created" << std::endl; }  
 ~Gadget() { std::cout << "Gadget destroyed" << std::endl; }  
};  
  
int main() {  
 // Test basic CRTP  
 Derived1 d1;  
 Derived2 d2;  
   
 // Call interface method which will use the derived implementation  
 d1.interface(); // Calls Derived1::implementation()  
 d2.interface(); // Calls Derived2::implementation()  
   
 // Call method with pre/post work  
 d1.doSomething();  
   
 // Test object counter  
 std::cout << "\nInitial count - Widgets: " << Widget::getCount()   
 << ", Gadgets: " << Gadget::getCount() << std::endl;  
   
 {  
 Widget w1, w2;  
 Gadget g1;  
   
 std::cout << "After creating objects - Widgets: " << Widget::getCount()   
 << ", Gadgets: " << Gadget::getCount() << std::endl;  
 } // Objects go out of scope and are destroyed  
   
 std::cout << "Final count - Widgets: " << Widget::getCount()   
 << ", Gadgets: " << Gadget::getCount() << std::endl;  
   
 return 0;  
}

Benefits of CRTP: 1. **Static polymorphism**: Achieved at compile time without the overhead of virtual functions 2. **Mixins**: Add functionality to classes without multiple inheritance issues 3. **Template method pattern**: Base class defines the algorithm structure, derived classes provide specific implementations 4. **Per-class storage**: Each derived class gets its own set of static variables

### 10.6.5 Traits and Policy Classes

Traits and policy classes are powerful template techniques for customizing behavior:

#### Traits Classes

Traits classes provide a way to access information about types:

#include <iostream>  
#include <string>  
#include <vector>  
#include <list>  
#include <type\_traits>  
  
// A traits class for containers  
template<typename Container>  
struct ContainerTraits {  
 // Default implementation for common containers  
 using value\_type = typename Container::value\_type;  
 using size\_type = typename Container::size\_type;  
 using iterator = typename Container::iterator;  
   
 static constexpr bool has\_random\_access = false; // Default assumption  
};  
  
// Specialization for std::vector  
template<typename T, typename Allocator>  
struct ContainerTraits<std::vector<T, Allocator>> {  
 using value\_type = typename std::vector<T, Allocator>::value\_type;  
 using size\_type = typename std::vector<T, Allocator>::size\_type;  
 using iterator = typename std::vector<T, Allocator>::iterator;  
   
 static constexpr bool has\_random\_access = true; // Vector has random access  
};  
  
// Specialization for std::list  
template<typename T, typename Allocator>  
struct ContainerTraits<std::list<T, Allocator>> {  
 using value\_type = typename std::list<T, Allocator>::value\_type;  
 using size\_type = typename std::list<T, Allocator>::size\_type;  
 using iterator = typename std::list<T, Allocator>::iterator;  
   
 static constexpr bool has\_random\_access = false; // List doesn't have random access  
};  
  
// Function that uses traits to optimize container operations  
template<typename Container>  
void process(Container& c) {  
 using Traits = ContainerTraits<Container>;  
 using value\_type = typename Traits::value\_type;  
   
 std::cout << "Container with value type: " << typeid(value\_type).name() << std::endl;  
   
 // Use traits to optimize operations  
 if constexpr (Traits::has\_random\_access) {  
 std::cout << "Using random access optimization..." << std::endl;  
 // Direct access to each element  
 for (typename Traits::size\_type i = 0; i < c.size(); ++i) {  
 c[i] = static\_cast<value\_type>(i);  
 }  
 } else {  
 std::cout << "Using sequential access..." << std::endl;  
 // Sequential traversal  
 typename Traits::size\_type i = 0;  
 for (auto& elem : c) {  
 elem = static\_cast<value\_type>(i++);  
 }  
 }  
}  
  
// Custom traits for numeric types to provide common operations  
template<typename T, typename Enable = void>  
struct NumericTraits {  
 static constexpr bool is\_numeric = false;  
 static constexpr bool is\_integral = false;  
 static constexpr bool is\_floating\_point = false;  
   
 // These won't be used for non-numeric types, but we need them for compilation  
 static T zero() { return T(); }  
 static T one() { return T(); }  
};  
  
// Specialization for integral types  
template<typename T>  
struct NumericTraits<T, typename std::enable\_if<std::is\_integral<T>::value>::type> {  
 static constexpr bool is\_numeric = true;  
 static constexpr bool is\_integral = true;  
 static constexpr bool is\_floating\_point = false;  
   
 static T zero() { return 0; }  
 static T one() { return 1; }  
 static T max\_value() { return std::numeric\_limits<T>::max(); }  
 static T min\_value() { return std::numeric\_limits<T>::min(); }  
};  
  
// Specialization for floating-point types  
template<typename T>  
struct NumericTraits<T, typename std::enable\_if<std::is\_floating\_point<T>::value>::type> {  
 static constexpr bool is\_numeric = true;  
 static constexpr bool is\_integral = false;  
 static constexpr bool is\_floating\_point = true;  
   
 static T zero() { return 0.0; }  
 static T one() { return 1.0; }  
 static T max\_value() { return std::numeric\_limits<T>::max(); }  
 static T min\_value() { return std::numeric\_limits<T>::lowest(); } // Note: lowest for float  
 static T epsilon() { return std::numeric\_limits<T>::epsilon(); }  
   
 // Check if two floating-point values are approximately equal  
 static bool approx\_equal(T a, T b, T tolerance = epsilon()) {  
 return std::abs(a - b) <= tolerance;  
 }  
};  
  
int main() {  
 // Test container traits  
 std::vector<int> vec(5);  
 std::list<double> lst(5);  
   
 process(vec);  
 process(lst);  
   
 std::cout << "\nVector contents: ";  
 for (const auto& v : vec) std::cout << v << " ";  
 std::cout << std::endl;  
   
 std::cout << "List contents: ";  
 for (const auto& v : lst) std::cout << v << " ";  
 std::cout << std::endl;  
   
 // Test numeric traits  
 using IntTraits = NumericTraits<int>;  
 using DoubleTraits = NumericTraits<double>;  
 using StringTraits = NumericTraits<std::string>;  
   
 std::cout << "\nNumeric traits for int:" << std::endl;  
 std::cout << "Is numeric: " << IntTraits::is\_numeric << std::endl;  
 std::cout << "Is integral: " << IntTraits::is\_integral << std::endl;  
 std::cout << "Is floating point: " << IntTraits::is\_floating\_point << std::endl;  
 std::cout << "Zero value: " << IntTraits::zero() << std::endl;  
 std::cout << "Max value: " << IntTraits::max\_value() << std::endl;  
   
 std::cout << "\nNumeric traits for double:" << std::endl;  
 std::cout << "Is numeric: " << DoubleTraits::is\_numeric << std::endl;  
 std::cout << "Is integral: " << DoubleTraits::is\_integral << std::endl;  
 std::cout << "Is floating point: " << DoubleTraits::is\_floating\_point << std::endl;  
 std::cout << "Epsilon: " << DoubleTraits::epsilon() << std::endl;  
   
 double a = 0.1 + 0.2;  
 double b = 0.3;  
 std::cout << "0.1 + 0.2 == 0.3? " << DoubleTraits::approx\_equal(a, b) << std::endl;  
   
 std::cout << "\nNumeric traits for string:" << std::endl;  
 std::cout << "Is numeric: " << StringTraits::is\_numeric << std::endl;  
   
 return 0;  
}

#### Policy Classes

Policy classes encapsulate behaviors that can be combined to customize class behavior:

#include <iostream>  
#include <vector>  
#include <list>  
#include <mutex>  
#include <string>  
#include <memory>  
  
// Threading policies  
class SingleThreadPolicy {  
public:  
 void lock() const {} // No-op  
 void unlock() const {} // No-op  
   
 const char\* name() const { return "Single-threaded"; }  
};  
  
class MultiThreadPolicy {  
private:  
 mutable std::mutex mtx;  
   
public:  
 void lock() const { mtx.lock(); }  
 void unlock() const { mtx.unlock(); }  
   
 const char\* name() const { return "Multi-threaded"; }  
};  
  
// Storage policies  
template<typename T>  
class VectorStorage {  
protected:  
 std::vector<T> data;  
   
public:  
 void add(const T& item) { data.push\_back(item); }  
   
 T& get(size\_t index) {   
 if (index >= data.size()) throw std::out\_of\_range("Index out of bounds");  
 return data[index];   
 }  
   
 size\_t size() const { return data.size(); }  
   
 const char\* name() const { return "Vector storage"; }  
};  
  
template<typename T>  
class ListStorage {  
protected:  
 std::list<T> data;  
   
public:  
 void add(const T& item) { data.push\_back(item); }  
   
 T& get(size\_t index) {   
 if (index >= data.size()) throw std::out\_of\_range("Index out of bounds");  
 auto it = data.begin();  
 std::advance(it, index);  
 return \*it;   
 }  
   
 size\_t size() const { return data.size(); }  
   
 const char\* name() const { return "List storage"; }  
};  
  
// Allocation policies  
template<typename T>  
class DefaultAllocPolicy {  
public:  
 T\* allocate() { return new T(); }  
 void deallocate(T\* ptr) { delete ptr; }  
   
 const char\* name() const { return "Default allocation"; }  
};  
  
template<typename T>  
class CountingAllocPolicy {  
private:  
 static inline size\_t allocCount = 0;  
 static inline size\_t deallocCount = 0;  
   
public:  
 T\* allocate() {   
 ++allocCount;   
 return new T();   
 }  
   
 void deallocate(T\* ptr) {   
 delete ptr;   
 ++deallocCount;   
 }  
   
 static size\_t getAllocCount() { return allocCount; }  
 static size\_t getDeallocCount() { return deallocCount; }  
   
 const char\* name() const { return "Counting allocation"; }  
};  
  
// Main container class that combines policies  
template<  
 typename T,  
 typename ThreadingPolicy = SingleThreadPolicy,  
 template<typename> class StoragePolicy = VectorStorage,  
 template<typename> class AllocPolicy = DefaultAllocPolicy  
>  
class Container : private ThreadingPolicy, private StoragePolicy<T>, private AllocPolicy<T> {  
public:  
 // Add item with thread safety  
 void add(const T& item) {  
 ThreadingPolicy::lock();  
 StoragePolicy<T>::add(item);  
 ThreadingPolicy::unlock();  
 }  
   
 // Get item with thread safety  
 T& get(size\_t index) {  
 ThreadingPolicy::lock();  
 try {  
 T& result = StoragePolicy<T>::get(index);  
 ThreadingPolicy::unlock();  
 return result;  
 } catch (...) {  
 ThreadingPolicy::unlock();  
 throw;  
 }  
 }  
   
 // Create a new item using the allocation policy  
 std::unique\_ptr<T> createItem() {  
 ThreadingPolicy::lock();  
 T\* item = AllocPolicy<T>::allocate();  
 ThreadingPolicy::unlock();  
   
 // Use unique\_ptr with custom deleter  
 return std::unique\_ptr<T>(item, [this](T\* ptr) {  
 ThreadingPolicy::lock();  
 AllocPolicy<T>::deallocate(ptr);  
 ThreadingPolicy::unlock();  
 });  
 }  
   
 // Get information about the container  
 void printInfo() const {  
 std::cout << "Container Info:" << std::endl;  
 std::cout << "- Thread policy: " << ThreadingPolicy::name() << std::endl;  
 std::cout << "- Storage policy: " << StoragePolicy<T>::name() << std::endl;  
 std::cout << "- Allocation policy: " << AllocPolicy<T>::name() << std::endl;  
 std::cout << "- Size: " << StoragePolicy<T>::size() << " items" << std::endl;  
 }  
};  
  
int main() {  
 // Single-threaded container with vector storage and default allocation  
 Container<int> basicContainer;  
 basicContainer.printInfo();  
   
 for (int i = 0; i < 5; ++i) {  
 basicContainer.add(i \* 10);  
 }  
   
 std::cout << "\nBasic container contents:" << std::endl;  
 for (size\_t i = 0; i < 5; ++i) {  
 std::cout << "Item " << i << ": " << basicContainer.get(i) << std::endl;  
 }  
   
 // Multi-threaded container with list storage and counting allocation  
 Container<std::string, MultiThreadPolicy, ListStorage, CountingAllocPolicy> advancedContainer;  
 advancedContainer.printInfo();  
   
 advancedContainer.add("Hello");  
 advancedContainer.add("Policy-based");  
 advancedContainer.add("Design");  
   
 std::cout << "\nAdvanced container contents:" << std::endl;  
 for (size\_t i = 0; i < 3; ++i) {  
 std::cout << "Item " << i << ": " << advancedContainer.get(i) << std::endl;  
 }  
   
 // Create items using allocation policy  
 auto item1 = advancedContainer.createItem();  
 auto item2 = advancedContainer.createItem();  
 \*item1 = "Allocated string 1";  
 \*item2 = "Allocated string 2";  
   
 std::cout << "\nAllocated items:" << std::endl;  
 std::cout << \*item1 << std::endl;  
 std::cout << \*item2 << std::endl;  
   
 // Check allocation statistics  
 std::cout << "\nAllocation statistics:" << std::endl;  
 std::cout << "- Allocations: " << CountingAllocPolicy<std::string>::getAllocCount() << std::endl;  
 std::cout << "- Deallocations: " << CountingAllocPolicy<std::string>::getDeallocCount() << std::endl;  
   
 return 0;  
}

### 10.6.6 Template Method Chaining

Template method chaining (fluent interfaces) can be implemented with templates to create expressive APIs:

#include <iostream>  
#include <string>  
#include <vector>  
#include <sstream>  
  
// Forward declaration of builder class  
template<typename BuilderType>  
class QueryBuilderBase;  
  
// Chain-enabled WHERE clause builder  
template<typename BuilderType>  
class WhereClause {  
private:  
 QueryBuilderBase<BuilderType>& builder;  
 bool hasWhere = false;  
   
public:  
 explicit WhereClause(QueryBuilderBase<BuilderType>& b) : builder(b) {}  
   
 BuilderType& where(const std::string& condition) {  
 builder.addClause(hasWhere ? "AND" : "WHERE", condition);  
 hasWhere = true;  
 return static\_cast<BuilderType&>(builder);  
 }  
   
 BuilderType& orWhere(const std::string& condition) {  
 builder.addClause("OR", condition);  
 hasWhere = true;  
 return static\_cast<BuilderType&>(builder);  
 }  
};  
  
// Chain-enabled ORDER BY clause builder  
template<typename BuilderType>  
class OrderByClause {  
private:  
 QueryBuilderBase<BuilderType>& builder;  
 bool hasOrderBy = false;  
   
public:  
 explicit OrderByClause(QueryBuilderBase<BuilderType>& b) : builder(b) {}  
   
 BuilderType& orderBy(const std::string& column, bool ascending = true) {  
 std::string direction = ascending ? "ASC" : "DESC";  
 builder.addClause(hasOrderBy ? "," : "ORDER BY", column + " " + direction);  
 hasOrderBy = true;  
 return static\_cast<BuilderType&>(builder);  
 }  
};  
  
// Base query builder with common functionality  
template<typename BuilderType>  
class QueryBuilderBase : public WhereClause<BuilderType>,   
 public OrderByClause<BuilderType> {  
private:  
 std::string tableName;  
 std::vector<std::string> clauses;  
   
protected:  
 QueryBuilderBase(const std::string& table)  
 : WhereClause<BuilderType>(\*this),  
 OrderByClause<BuilderType>(\*this),  
 tableName(table) {}  
   
 // Make base class a friend to access addClause  
 friend class WhereClause<BuilderType>;  
 friend class OrderByClause<BuilderType>;  
   
 // Add a clause to the query  
 void addClause(const std::string& keyword, const std::string& clause) {  
 clauses.push\_back(keyword + " " + clause);  
 }  
   
public:  
 // Get the generated SQL query  
 std::string getSQL() const {  
 std::ostringstream sql;  
 sql << getQueryPrefix() << " FROM " << tableName;  
   
 for (const auto& clause : clauses) {  
 sql << " " << clause;  
 }  
   
 return sql.str();  
 }  
   
 // Execute the query (in a real system, this would run the SQL)  
 void execute() {  
 std::cout << "Executing SQL: " << getSQL() << std::endl;  
 // In real code: run the SQL and return results  
 }  
   
 // Must be overridden by derived classes  
 virtual std::string getQueryPrefix() const = 0;  
};  
  
// Concrete Select query builder  
class SelectQueryBuilder : public QueryBuilderBase<SelectQueryBuilder> {  
private:  
 std::string columns;  
   
public:  
 SelectQueryBuilder() : QueryBuilderBase("") {}  
   
 // Start building a SELECT query  
 SelectQueryBuilder& from(const std::string& table) {  
 // Reset the table name in the base class  
 tableName = table;  
 return \*this;  
 }  
   
 // Select specific columns  
 SelectQueryBuilder& select(const std::string& columnList) {  
 columns = columnList;  
 return \*this;  
 }  
   
 // Override to provide SELECT-specific prefix  
 std::string getQueryPrefix() const override {  
 return "SELECT " + (columns.empty() ? "\*" : columns);  
 }  
};  
  
// Concrete Update query builder  
class UpdateQueryBuilder : public QueryBuilderBase<UpdateQueryBuilder> {  
private:  
 std::string setClause;  
   
public:  
 UpdateQueryBuilder(const std::string& table) : QueryBuilderBase(table) {}  
   
 // Set values to update  
 UpdateQueryBuilder& set(const std::string& updates) {  
 setClause = updates;  
 return \*this;  
 }  
   
 // Override to provide UPDATE-specific prefix  
 std::string getQueryPrefix() const override {  
 return "UPDATE " + tableName + " SET " + setClause;  
 }  
};  
  
// Concrete Delete query builder  
class DeleteQueryBuilder : public QueryBuilderBase<DeleteQueryBuilder> {  
public:  
 DeleteQueryBuilder(const std::string& table) : QueryBuilderBase(table) {}  
   
 // Override to provide DELETE-specific prefix  
 std::string getQueryPrefix() const override {  
 return "DELETE";  
 }  
};  
  
// Entry point for query building  
class QueryBuilder {  
public:  
 static SelectQueryBuilder select(const std::string& columns = "\*") {  
 return SelectQueryBuilder().select(columns);  
 }  
   
 static UpdateQueryBuilder update(const std::string& table) {  
 return UpdateQueryBuilder(table);  
 }  
   
 static DeleteQueryBuilder deleteFrom(const std::string& table) {  
 return DeleteQueryBuilder(table);  
 }  
};  
  
int main() {  
 // Build and execute a SELECT query  
 auto selectQuery = QueryBuilder::select("id, name, email")  
 .from("users")  
 .where("status = 'active'")  
 .andWhere("age > 18")  
 .orderBy("name")  
 .orderBy("id", false); // DESC  
   
 std::cout << "Select SQL: " << selectQuery.getSQL() << std::endl;  
 selectQuery.execute();  
   
 // Build and execute an UPDATE query  
 auto updateQuery = QueryBuilder::update("users")  
 .set("status = 'inactive', last\_login = CURRENT\_TIMESTAMP")  
 .where("last\_login < '2023-01-01'")  
 .orWhere("status = 'pending'");  
   
 std::cout << "\nUpdate SQL: " << updateQuery.getSQL() << std::endl;  
 updateQuery.execute();  
   
 // Build and execute a DELETE query  
 auto deleteQuery = QueryBuilder::deleteFrom("temp\_sessions")  
 .where("created\_at < '2023-01-01'")  
 .andWhere("status = 'expired'");  
   
 std::cout << "\nDelete SQL: " << deleteQuery.getSQL() << std::endl;  
 deleteQuery.execute();  
   
 return 0;  
}

### 10.6.7 Type Erasure with Templates

Type erasure is a technique that allows you to use objects of different types through a common interface without inheritance:

#include <iostream>  
#include <memory>  
#include <vector>  
#include <string>  
#include <functional>  
  
// Type-erased function wrapper  
class AnyCallable {  
private:  
 struct CallableConcept {  
 virtual ~CallableConcept() = default;  
 virtual void call() const = 0;  
 virtual std::unique\_ptr<CallableConcept> clone() const = 0;  
 };  
   
 template<typename F>  
 struct CallableModel : CallableConcept {  
 F func;  
   
 explicit CallableModel(F f) : func(std::move(f)) {}  
   
 void call() const override {  
 func();  
 }  
   
 std::unique\_ptr<CallableConcept> clone() const override {  
 return std::make\_unique<CallableModel<F>>(func);  
 }  
 };  
   
 std::unique\_ptr<CallableConcept> callable;  
   
public:  
 AnyCallable() = default;  
   
 template<typename F>  
 AnyCallable(F f) : callable(std::make\_unique<CallableModel<F>>(std::move(f))) {}  
   
 AnyCallable(const AnyCallable& other) : callable(other.callable ? other.callable->clone() : nullptr) {}  
   
 AnyCallable(AnyCallable&& other) noexcept = default;  
   
 AnyCallable& operator=(const AnyCallable& other) {  
 if (this != &other) {  
 callable = other.callable ? other.callable->clone() : nullptr;  
 }  
 return \*this;  
 }  
   
 AnyCallable& operator=(AnyCallable&& other) noexcept = default;  
   
 void operator()() const {  
 if (callable) {  
 callable->call();  
 }  
 }  
   
 explicit operator bool() const {  
 return callable != nullptr;  
 }  
};  
  
// Type-erased container using templates  
class AnyValue {  
private:  
 struct ValueConcept {  
 virtual ~ValueConcept() = default;  
 virtual std::unique\_ptr<ValueConcept> clone() const = 0;  
 virtual void print(std::ostream& os) const = 0;  
 virtual std::string getTypeName() const = 0;  
 };  
   
 template<typename T>  
 struct ValueModel : ValueConcept {  
 T value;  
   
 explicit ValueModel(T v) : value(std::move(v)) {}  
   
 std::unique\_ptr<ValueConcept> clone() const override {  
 return std::make\_unique<ValueModel<T>>(value);  
 }  
   
 void print(std::ostream& os) const override {  
 os << value;  
 }  
   
 std::string getTypeName() const override {  
 return typeid(T).name();  
 }  
 };  
   
 std::unique\_ptr<ValueConcept> content;  
   
public:  
 AnyValue() = default;  
   
 template<typename T>  
 explicit AnyValue(T value) : content(std::make\_unique<ValueModel<T>>(std::move(value))) {}  
   
 AnyValue(const AnyValue& other) : content(other.content ? other.content->clone() : nullptr) {}  
   
 AnyValue(AnyValue&& other) noexcept = default;  
   
 AnyValue& operator=(const AnyValue& other) {  
 if (this != &other) {  
 content = other.content ? other.content->clone() : nullptr;  
 }  
 return \*this;  
 }  
   
 AnyValue& operator=(AnyValue&& other) noexcept = default;  
   
 template<typename T>  
 T& get() {  
 auto\* model = dynamic\_cast<ValueModel<T>\*>(content.get());  
 if (!model) {  
 throw std::bad\_cast();  
 }  
 return model->value;  
 }  
   
 template<typename T>  
 bool is() const {  
 return dynamic\_cast<ValueModel<T>\*>(content.get()) != nullptr;  
 }  
   
 friend std::ostream& operator<<(std::ostream& os, const AnyValue& any) {  
 if (any.content) {  
 any.content->print(os);  
 } else {  
 os << "[empty]";  
 }  
 return os;  
 }  
   
 std::string getTypeName() const {  
 return content ? content->getTypeName() : "none";  
 }  
   
 explicit operator bool() const {  
 return content != nullptr;  
 }  
};  
  
int main() {  
 // Test AnyCallable  
 std::vector<AnyCallable> callbacks;  
   
 // Add different types of callables  
 callbacks.push\_back([]{ std::cout << "Hello from lambda!" << std::endl; });  
   
 struct Functor {  
 void operator()() const {   
 std::cout << "Hello from functor!" << std::endl;   
 }  
 };  
 callbacks.push\_back(Functor());  
   
 void regularFunction() {   
 std::cout << "Hello from regular function!" << std::endl;   
 }  
 callbacks.push\_back(regularFunction);  
   
 // Execute all callbacks  
 std::cout << "Executing callbacks:" << std::endl;  
 for (const auto& callback : callbacks) {  
 callback();  
 }  
   
 // Test AnyValue  
 std::vector<AnyValue> values;  
   
 // Store different types  
 values.push\_back(AnyValue(42));  
 values.push\_back(AnyValue(std::string("Hello, world!")));  
 values.push\_back(AnyValue(3.14159));  
 values.push\_back(AnyValue(true));  
   
 // Display all values  
 std::cout << "\nStored values:" << std::endl;  
 for (const auto& value : values) {  
 std::cout << "Type: " << value.getTypeName() << ", Value: " << value << std::endl;  
 }  
   
 // Access specific value  
 try {  
 std::string& str = values[1].get<std::string>();  
 str += " (modified)";  
   
 std::cout << "\nModified string: " << str << std::endl;  
 std::cout << "From container: " << values[1] << std::endl;  
 } catch (const std::bad\_cast& e) {  
 std::cerr << "Type conversion failed: " << e.what() << std::endl;  
 }  
   
 // Check types  
 std::cout << "\nType checking:" << std::endl;  
 std::cout << "values[0] is int: " << values[0].is<int>() << std::endl;  
 std::cout << "values[1] is string: " << values[1].is<std::string>() << std::endl;  
 std::cout << "values[2] is double: " << values[2].is<double>() << std::endl;  
 std::cout << "values[0] is double: " << values[0].is<double>() << std::endl;  
   
 return 0;  
}

### 10.6.8 Optimizing Template Code

Templates can lead to code bloat and increased compilation times if not used carefully. Here are some techniques to optimize template usage:

#include <iostream>  
#include <typeinfo>  
#include <memory>  
#include <vector>  
#include <map>  
  
// 1. Use explicit instantiation for common types  
template <typename T>  
class CommonContainer {  
public:  
 CommonContainer() { std::cout << "Instantiating for " << typeid(T).name() << std::endl; }  
 void process(const T& value) { std::cout << "Processing: " << value << std::endl; }  
};  
  
// Explicit instantiation in the .cpp file (reduces compile time in headers)  
template class CommonContainer<int>;  
template class CommonContainer<double>;  
template class CommonContainer<std::string>;  
  
// 2. Use the Pimpl idiom to reduce compilation dependencies  
template <typename T>  
class PimplTemplate {  
private:  
 // Forward declaration of implementation  
 class Impl;  
 std::unique\_ptr<Impl> pimpl;  
   
public:  
 PimplTemplate();  
 ~PimplTemplate();  
 void doSomething(const T& value);  
};  
  
// Implementation in .cpp file  
template <typename T>  
class PimplTemplate<T>::Impl {  
public:  
 void doSomething(const T& value) {  
 std::cout << "Implementation doing something with: " << value << std::endl;  
 }  
};  
  
template <typename T>  
PimplTemplate<T>::PimplTemplate() : pimpl(std::make\_unique<Impl>()) {}  
  
template <typename T>  
PimplTemplate<T>::~PimplTemplate() = default;  
  
template <typename T>  
void PimplTemplate<T>::doSomething(const T& value) {  
 pimpl->doSomething(value);  
}  
  
// Explicit instantiation for common types  
template class PimplTemplate<int>;  
template class PimplTemplate<std::string>;  
  
// 3. Use type erasure for template parameters that don't affect performance-critical code  
class TypeErasedInterface {  
public:  
 virtual ~TypeErasedInterface() = default;  
 virtual void process() = 0;  
};  
  
template <typename T>  
class TypeErasedImpl : public TypeErasedInterface {  
private:  
 T value;  
   
public:  
 explicit TypeErasedImpl(T v) : value(std::move(v)) {}  
   
 void process() override {  
 std::cout << "Processing with type erasure: " << value << std::endl;  
 }  
};  
  
// 4. Consider non-template base classes where possible  
class NonTemplateBase {  
public:  
 virtual ~NonTemplateBase() = default;  
 virtual void commonBehavior() {  
 std::cout << "Common behavior in non-template base" << std::endl;  
 }  
};  
  
template <typename T>  
class TemplatedDerived : public NonTemplateBase {  
private:  
 T specificData;  
   
public:  
 explicit TemplatedDerived(T data) : specificData(std::move(data)) {}  
   
 void specificBehavior() {  
 std::cout << "Specific behavior with: " << specificData << std::endl;  
 }  
};  
  
// 5. Use static polymorphism only where dynamic dispatch would be a performance bottleneck  
template <typename T>  
class StaticHandler {  
public:  
 void handle(const T& value) {  
 std::cout << "Static handling of: " << value << std::endl;  
 }  
};  
  
class DynamicHandler {  
public:  
 virtual ~DynamicHandler() = default;  
 virtual void handle(int value) {  
 std::cout << "Dynamic handling of int: " << value << std::endl;  
 }  
};  
  
class SpecializedHandler : public DynamicHandler {  
public:  
 void handle(int value) override {  
 std::cout << "Specialized handling of int: " << value \* 2 << std::endl;  
 }  
};  
  
int main() {  
 // 1. Explicit instantiation example  
 CommonContainer<int> intContainer;  
 intContainer.process(42);  
   
 CommonContainer<std::string> strContainer;  
 strContainer.process("Hello");  
   
 // 2. Pimpl idiom example  
 PimplTemplate<int> pimplInt;  
 pimplInt.doSomething(42);  
   
 // 3. Type erasure example  
 std::vector<std::unique\_ptr<TypeErasedInterface>> objects;  
 objects.push\_back(std::make\_unique<TypeErasedImpl<int>>(42));  
 objects.push\_back(std::make\_unique<TypeErasedImpl<std::string>>("Hello"));  
 objects.push\_back(std::make\_unique<TypeErasedImpl<double>>(3.14));  
   
 for (const auto& obj : objects) {  
 obj->process();  
 }  
   
 // 4. Non-template base example  
 TemplatedDerived<int> derivedInt(42);  
 derivedInt.commonBehavior();  
 derivedInt.specificBehavior();  
   
 TemplatedDerived<std::string> derivedStr("World");  
 derivedStr.commonBehavior();  
 derivedStr.specificBehavior();  
   
 // 5. Static vs dynamic polymorphism  
 StaticHandler<int> staticHandler;  
 staticHandler.handle(42);  
   
 DynamicHandler\* dynamicHandler = new SpecializedHandler();  
 dynamicHandler->handle(42);  
 delete dynamicHandler;  
   
 return 0;  
}

### 10.6.9 Best Practices for Advanced Template Usage

1. **Document template requirements clearly**
   * Specify requirements for template parameters
   * Provide examples of valid template arguments
   * Document any implicit assumptions
2. **Use concept constraints in C++20**
   * Make requirements explicit with concepts
   * Provide better error messages for users
   * Enable more IDE support and better diagnostics
3. **Balance flexibility and usability**
   * Don’t overtemplate code that doesn’t need it
   * Consider providing non-template alternatives where appropriate
   * Keep the interface simple for common use cases
4. **Reduce compilation dependencies**
   * Use forward declarations where possible
   * Apply the Pimpl idiom for complex implementations
   * Consider explicit instantiation for common types
5. **Optimize for both compile time and runtime**
   * Be aware of template instantiation costs
   * Use explicit instantiation to reduce duplicate code generation
   * Apply techniques like external polymorphism for large class templates
6. **Test template code thoroughly**
   * Test with various combinations of template arguments
   * Include edge cases and custom types
   * Test with SFINAE or concept failures to ensure error messages are helpful
7. **Use meaningful template parameter names**
   * Prefer descriptive names like ElementType over just T
   * Document the purpose of each template parameter
   * Use consistent naming conventions across your codebase
8. **Consider compile-time and runtime tradeoffs**
   * Templates can move work from runtime to compile time
   * Balance between static and dynamic polymorphism based on performance needs
   * Use runtime polymorphism for flexibility, static for performance-critical code
9. **Apply template design patterns appropriately**
   * Use policy-based design for customizable behaviors
   * Apply CRTP for static polymorphism
   * Use type erasure for interface uniformity with implementation flexibility
10. **Manage complexity**
    * Keep template metaprograms as simple as possible
    * Provide higher-level abstractions over complex template implementations
    * Consider alternative approaches before diving into complex metaprogramming

By following these best practices and understanding the advanced techniques demonstrated in this section, you’ll be able to leverage C++ templates to create efficient, flexible, and maintainable code for a wide range of applications.

# Chapter 11: C++ in the Real World (Part 1)

## Introduction to C++ in the Real World

C++ remains one of the most powerful and versatile programming languages in the industry today, despite being over four decades old. Its ability to provide low-level memory manipulation combined with high-level abstractions makes it uniquely suited for a wide range of applications, from embedded systems to large-scale enterprise software.

### Why C++ Continues to Thrive

1. **Performance**: C++ provides near-bare-metal performance while offering high-level abstractions. This combination is rare among programming languages.
2. **Portability**: C++ code can run on virtually any platform, from tiny embedded microcontrollers to supercomputers.
3. **Scalability**: The language scales well from small applications to massive codebases with millions of lines of code.
4. **Ecosystem**: A vast ecosystem of libraries, tools, and frameworks has evolved around C++.
5. **Evolving Standard**: The language continues to evolve with new standards (C++11, C++14, C++17, C++20, etc.) that add modern programming features while maintaining backward compatibility.

### Industries and Domains Where C++ Excels

1. **Gaming**: Most AAA game engines (Unreal Engine, Unity’s native parts, CryEngine) are built with C++
2. **Financial Systems**: High-frequency trading platforms and banking systems use C++ for its performance and deterministic behavior
3. **Embedded Systems**: From automotive to aerospace, C++ is used when hardware resources are limited
4. **Systems Software**: Operating systems, browsers, databases, and compilers are often written in C++
5. **Scientific Computing**: For applications that require complex calculations on large datasets
6. **Telecommunications**: Network infrastructure and telecommunications equipment frequently use C++
7. **Graphics and Visualization**: CAD software, 3D modeling tools, and multimedia applications

### The Professional C++ Developer’s Toolkit

A professional C++ developer typically works with:

1. **Build Systems**: CMake, Make, MSBuild
2. **Version Control**: Git, SVN
3. **Debuggers**: GDB, Visual Studio Debugger, LLDB
4. **Profilers**: Valgrind, Intel VTune, Visual Studio Profiler
5. **Static Analyzers**: Clang-Tidy, Coverity, CppCheck
6. **Package Managers**: Conan, vcpkg
7. **Unit Testing Frameworks**: Google Test, Catch2, Boost.Test
8. **Continuous Integration**: Jenkins, GitHub Actions, Travis CI

With this foundation in mind, let’s dive into our first real-world application of C++: Competitive Programming.

## 11.1 Competitive Programming with C++

Competitive programming is a mind sport where participants solve algorithmic problems within time constraints. C++ is the most popular language in this domain due to its execution speed, rich standard library, and powerful features.

### 11.1.1 Why C++ Dominates Competitive Programming

1. **Execution Speed**: C++ is one of the fastest languages, crucial when solutions need to process large inputs within strict time limits.
2. **Standard Template Library (STL)**: Provides optimized implementations of data structures and algorithms:
   * Containers: vector, map, set, queue, etc.
   * Algorithms: sort, binary\_search, next\_permutation, etc.
3. **Control over Memory**: Memory management control helps optimize solutions for tight memory constraints.
4. **Powerful Features**: Templates, lambdas, and other features enable concise yet efficient code.
5. **Mature Ecosystem**: Most online judges and competitions support C++.

### 11.1.2 Setting Up for Competitive Programming

#### Compiler Setup

For competitive programming, you’ll want an optimized compiler setup:

# GCC compilation with optimizations  
g++ -std=c++17 -O2 -Wall solution.cpp -o solution  
  
# Clang compilation  
clang++ -std=c++17 -O2 -Wall solution.cpp -o solution

The -O2 flag enables optimizations, which can significantly speed up your code in competitions.

#### Standard Template

Most competitive programmers start with a template to avoid rewriting common code:

#include <bits/stdc++.h>  
using namespace std;  
  
// Shorthand notations  
typedef long long ll;  
typedef vector<int> vi;  
typedef pair<int,int> pi;  
  
// Useful defines  
#define FOR(i,a,b) for (int i = (a); i < (b); i++)  
#define F0R(i,a) FOR(i,0,a)  
#define REP(i,a) FOR(i,0,a)  
#define all(x) x.begin(), x.end()  
#define pb push\_back  
#define mp make\_pair  
#define fi first  
#define se second  
#define lb lower\_bound  
#define ub upper\_bound  
  
int main() {  
 ios::sync\_with\_stdio(false); // Faster I/O  
 cin.tie(nullptr);  
   
 // Your solution goes here  
   
 return 0;  
}

**Note**: While #include <bits/stdc++.h> is convenient for competitions as it includes most standard library headers, it’s not portable and should only be used in competitive programming, not production code.

### 11.1.3 Essential C++ Features for Competitive Programming

#### Fast Input/Output

I/O operations can become a bottleneck in competitive programming:

// Fast I/O setup  
ios\_base::sync\_with\_stdio(false);  
cin.tie(nullptr);  
  
// Reading until EOF  
while (cin >> x) {  
 // Process x  
}  
  
// For extremely fast input (rare cases)  
int x;  
scanf("%d", &x);

#### STL Containers and Their Complexities

Understanding complexity is crucial for making the right container choice:

1. **Vector** - Dynamic array

* vector<int> v = {1, 2, 3, 4, 5};  
  v.push\_back(6); // O(1) amortized  
  cout << v[2]; // O(1) access  
  sort(v.begin(), v.end()); // O(n log n)

1. **Set/Map** - Balanced binary search trees

* set<int> s = {1, 2, 3};  
  s.insert(4); // O(log n)  
  s.count(2); // O(log n)  
    
  map<string, int> m;  
  m["Alice"] = 42; // O(log n)  
  cout << m["Bob"]; // O(log n)

1. **Unordered Set/Map** - Hash tables

* unordered\_set<int> us = {1, 2, 3};  
  us.insert(4); // O(1) average  
  us.count(2); // O(1) average  
    
  unordered\_map<string, int> um;  
  um["Alice"] = 42; // O(1) average  
  cout << um["Bob"]; // O(1) average

1. **Priority Queue** - Heap

* priority\_queue<int> pq;  
  pq.push(3); // O(log n)  
  pq.push(5);  
  pq.push(1);  
  cout << pq.top(); // O(1), prints 5  
  pq.pop(); // O(log n)

#### Useful STL Algorithms

STL algorithms can save significant coding time:

// Sorting  
sort(v.begin(), v.end()); // Default: ascending  
sort(v.begin(), v.end(), greater<int>()); // Descending  
  
// Binary search on sorted ranges  
binary\_search(v.begin(), v.end(), x); // True if x exists  
lower\_bound(v.begin(), v.end(), x); // Iterator to first element >= x  
upper\_bound(v.begin(), v.end(), x); // Iterator to first element > x  
  
// Min/Max  
int min\_elem = \*min\_element(v.begin(), v.end());  
int max\_elem = \*max\_element(v.begin(), v.end());  
  
// Permutations  
next\_permutation(v.begin(), v.end());  
prev\_permutation(v.begin(), v.end());  
  
// Heap operations  
make\_heap(v.begin(), v.end());  
push\_heap(v.begin(), v.end());  
pop\_heap(v.begin(), v.end());  
sort\_heap(v.begin(), v.end());  
  
// Numeric operations  
int sum = accumulate(v.begin(), v.end(), 0);  
int product = accumulate(v.begin(), v.end(), 1, multiplies<int>());  
  
// Set operations (on sorted ranges)  
set\_intersection(a.begin(), a.end(), b.begin(), b.end(), back\_inserter(result));  
set\_union(a.begin(), a.end(), b.begin(), b.end(), back\_inserter(result));  
set\_difference(a.begin(), a.end(), b.begin(), b.end(), back\_inserter(result));

### 11.1.4 Common Problem Types and Approaches

#### Graph Problems

Graphs are ubiquitous in competitive programming. Here’s a BFS implementation:

// BFS in an adjacency list representation  
vector<vector<int>> graph(n); // Adjacency list  
vector<bool> visited(n, false);  
queue<int> q;  
  
void bfs(int start) {  
 visited[start] = true;  
 q.push(start);  
   
 while (!q.empty()) {  
 int node = q.front();  
 q.pop();  
   
 // Process node  
 cout << "Visiting node: " << node << endl;  
   
 // Explore neighbors  
 for (int neighbor : graph[node]) {  
 if (!visited[neighbor]) {  
 visited[neighbor] = true;  
 q.push(neighbor);  
 }  
 }  
 }  
}

And DFS implementation:

// DFS in an adjacency list representation  
vector<vector<int>> graph(n); // Adjacency list  
vector<bool> visited(n, false);  
  
void dfs(int node) {  
 // Mark current node as visited  
 visited[node] = true;  
   
 // Process node  
 cout << "Visiting node: " << node << endl;  
   
 // Explore unvisited neighbors  
 for (int neighbor : graph[node]) {  
 if (!visited[neighbor]) {  
 dfs(neighbor);  
 }  
 }  
}

#### Dynamic Programming

Dynamic programming problems often appear in competitions. Here’s a classic example:

// Longest Increasing Subsequence  
int lengthOfLIS(vector<int>& nums) {  
 if (nums.empty()) return 0;  
   
 int n = nums.size();  
 vector<int> dp(n, 1); // dp[i] = length of LIS ending at i  
   
 for (int i = 1; i < n; i++) {  
 for (int j = 0; j < i; j++) {  
 if (nums[i] > nums[j]) {  
 dp[i] = max(dp[i], dp[j] + 1);  
 }  
 }  
 }  
   
 return \*max\_element(dp.begin(), dp.end());  
}

#### String Problems

String manipulation is common. Here’s an example of KMP pattern matching:

// KMP Algorithm for pattern matching  
vector<int> computeLPS(string pattern) {  
 int m = pattern.length();  
 vector<int> lps(m, 0);  
   
 int len = 0;  
 int i = 1;  
   
 while (i < m) {  
 if (pattern[i] == pattern[len]) {  
 len++;  
 lps[i] = len;  
 i++;  
 } else {  
 if (len != 0) {  
 len = lps[len - 1];  
 } else {  
 lps[i] = 0;  
 i++;  
 }  
 }  
 }  
   
 return lps;  
}  
  
vector<int> KMPSearch(string text, string pattern) {  
 vector<int> matches;  
 int n = text.length();  
 int m = pattern.length();  
   
 if (m == 0) return matches;  
   
 vector<int> lps = computeLPS(pattern);  
   
 int i = 0; // index for text  
 int j = 0; // index for pattern  
   
 while (i < n) {  
 if (pattern[j] == text[i]) {  
 j++;  
 i++;  
 }  
   
 if (j == m) {  
 matches.push\_back(i - j); // Found a match  
 j = lps[j - 1];  
 } else if (i < n && pattern[j] != text[i]) {  
 if (j != 0) {  
 j = lps[j - 1];  
 } else {  
 i++;  
 }  
 }  
 }  
   
 return matches;  
}

#### Number Theory

Number theory problems frequently appear in competitions:

// Sieve of Eratosthenes for finding all primes up to n  
vector<bool> sieve(int n) {  
 vector<bool> isPrime(n+1, true);  
 isPrime[0] = isPrime[1] = false;  
   
 for (int i = 2; i \* i <= n; i++) {  
 if (isPrime[i]) {  
 for (int j = i \* i; j <= n; j += i) {  
 isPrime[j] = false;  
 }  
 }  
 }  
   
 return isPrime;  
}  
  
// GCD (Greatest Common Divisor) using Euclidean algorithm  
int gcd(int a, int b) {  
 return b == 0 ? a : gcd(b, a % b);  
}  
  
// LCM (Least Common Multiple)  
int lcm(int a, int b) {  
 return a / gcd(a, b) \* b; // Avoid overflow: a\*b/gcd(a,b)  
}  
  
// Fast modular exponentiation  
int modPow(int base, int exp, int mod) {  
 int result = 1;  
 base %= mod;  
   
 while (exp > 0) {  
 if (exp & 1) result = (1LL \* result \* base) % mod;  
 exp >>= 1;  
 base = (1LL \* base \* base) % mod;  
 }  
   
 return result;  
}

### 11.1.5 Optimization Techniques

In competitive programming, you often need to optimize your solution:

#### Bit Manipulation

Bit manipulation can be a powerful tool:

// Check if a bit is set  
bool isBitSet(int num, int bit) {  
 return (num & (1 << bit)) != 0;  
}  
  
// Set a bit  
int setBit(int num, int bit) {  
 return num | (1 << bit);  
}  
  
// Unset a bit  
int unsetBit(int num, int bit) {  
 return num & ~(1 << bit);  
}  
  
// Toggle a bit  
int toggleBit(int num, int bit) {  
 return num ^ (1 << bit);  
}  
  
// Count set bits  
int countSetBits(int num) {  
 int count = 0;  
 while (num) {  
 count += num & 1;  
 num >>= 1;  
 }  
 return count;  
 // Alternatively: return \_\_builtin\_popcount(num); // GCC builtin  
}  
  
// Using bits for subset calculation  
void subsets(vector<int>& nums) {  
 int n = nums.size();  
 for (int mask = 0; mask < (1 << n); mask++) {  
 cout << "Subset: ";  
 for (int i = 0; i < n; i++) {  
 if (mask & (1 << i)) {  
 cout << nums[i] << " ";  
 }  
 }  
 cout << endl;  
 }  
}

#### Memoization and Tabulation

Remembering previously computed results can dramatically speed up recursive solutions:

// Memoization (top-down approach) for Fibonacci  
vector<int> memo(100, -1);  
  
int fibonacci(int n) {  
 if (n <= 1) return n;  
 if (memo[n] != -1) return memo[n];  
   
 memo[n] = fibonacci(n-1) + fibonacci(n-2);  
 return memo[n];  
}  
  
// Tabulation (bottom-up approach) for Fibonacci  
int fibonacci\_dp(int n) {  
 vector<int> dp(n+1);  
 dp[0] = 0;  
 dp[1] = 1;  
   
 for (int i = 2; i <= n; i++) {  
 dp[i] = dp[i-1] + dp[i-2];  
 }  
   
 return dp[n];  
}

#### Two Pointers Technique

The two pointers technique is useful for problems with sorted arrays:

// Finding pair with sum in a sorted array  
bool hasPairWithSum(vector<int>& nums, int target) {  
 int left = 0;  
 int right = nums.size() - 1;  
   
 while (left < right) {  
 int sum = nums[left] + nums[right];  
   
 if (sum == target) {  
 return true;  
 } else if (sum < target) {  
 left++;  
 } else {  
 right--;  
 }  
 }  
   
 return false;  
}

#### Sliding Window

For substring problems, the sliding window approach is efficient:

// Find maximum sum subarray of size k  
int maxSumSubarray(vector<int>& nums, int k) {  
 int n = nums.size();  
 if (n < k) return -1; // Invalid input  
   
 // Compute sum of first window  
 int windowSum = 0;  
 for (int i = 0; i < k; i++) {  
 windowSum += nums[i];  
 }  
   
 int maxSum = windowSum;  
   
 // Slide the window and update maximum  
 for (int i = k; i < n; i++) {  
 windowSum += nums[i] - nums[i-k];  
 maxSum = max(maxSum, windowSum);  
 }  
   
 return maxSum;  
}

### 11.1.6 Common Mistakes and How to Avoid Them

1. **Integer Overflow**: Use long long for large integers

* long long result = 1LL \* a \* b; // Explicit conversion to prevent overflow

1. **Off-by-one Errors**: Double-check array indices

* // Careful with bounds - ensure i < n, not i <= n  
  for (int i = 0; i < n; i++) { /\* ... \*/ }

1. **Uninitialized Variables**: Always initialize variables

* int sum = 0; // Initialize before using  
  for (int num : nums) sum += num;

1. **Stack Overflow**: Consider recursion depth

* // If recursion is too deep, rewrite using iteration or increase stack size  
  #pragma comment(linker, "/STACK:16777216") // MSVC

1. **Time Limit Exceeded (TLE)**: Analyze algorithm complexity

* // Instead of O(n^2), use a more efficient approach  
  unordered\_set<int> seen; // O(1) lookup instead of O(n) linear search

1. **Edge Cases**: Test with empty inputs, single elements, etc.

### 11.1.7 Resources for Competitive Programming

1. **Online Judges and Practice Platforms**:
   * [Codeforces](https://codeforces.com/)
   * [AtCoder](https://atcoder.jp/)
   * [LeetCode](https://leetcode.com/)
   * [HackerRank](https://www.hackerrank.com/)
   * [CodeChef](https://www.codechef.com/)
   * [SPOJ](https://www.spoj.com/)
2. **Books**:
   * “Competitive Programming” by Steven and Felix Halim
   * “Introduction to Algorithms” by Cormen, Leiserson, Rivest, and Stein
   * “Competitive Programmer’s Handbook” by Antti Laaksonen (free online)
3. **Online Resources**:
   * [CP-Algorithms](https://cp-algorithms.com/)
   * [USACO Guide](https://usaco.guide/)
   * [Codeforces EDU](https://codeforces.com/edu/courses)

### 11.1.8 A Complete Example: Solving a Competitive Programming Problem

Let’s walk through solving a classic problem: Finding the longest increasing subsequence.

**Problem Statement:** Given an array of integers, find the length of the longest strictly increasing subsequence.

**Example:** Input: [10, 9, 2, 5, 3, 7, 101, 18] Output: 4 Explanation: The longest increasing subsequence is [2, 5, 7, 101], with a length of 4.

**Approach 1: Dynamic Programming (O(n²))**

#include <bits/stdc++.h>  
using namespace std;  
  
int lengthOfLIS(vector<int>& nums) {  
 int n = nums.size();  
 if (n == 0) return 0;  
   
 vector<int> dp(n, 1); // dp[i] = length of LIS ending at index i  
   
 for (int i = 0; i < n; i++) {  
 for (int j = 0; j < i; j++) {  
 if (nums[i] > nums[j]) {  
 dp[i] = max(dp[i], dp[j] + 1);  
 }  
 }  
 }  
   
 return \*max\_element(dp.begin(), dp.end());  
}  
  
int main() {  
 ios::sync\_with\_stdio(false);  
 cin.tie(nullptr);  
   
 // Parse input  
 int n;  
 cin >> n;  
   
 vector<int> nums(n);  
 for (int i = 0; i < n; i++) {  
 cin >> nums[i];  
 }  
   
 // Solve and output  
 cout << lengthOfLIS(nums) << endl;  
   
 return 0;  
}

**Approach 2: Binary Search Optimization (O(n log n))**

#include <bits/stdc++.h>  
using namespace std;  
  
int lengthOfLIS(vector<int>& nums) {  
 int n = nums.size();  
 if (n == 0) return 0;  
   
 vector<int> tails; // tails[i] = smallest tail of all increasing subsequences of length i+1  
   
 for (int num : nums) {  
 // Binary search to find the position to place num  
 auto it = lower\_bound(tails.begin(), tails.end(), num);  
   
 if (it == tails.end()) {  
 // Found a new longest increasing subsequence  
 tails.push\_back(num);  
 } else {  
 // Replace the existing tail with a smaller value  
 \*it = num;  
 }  
 }  
   
 return tails.size();  
}  
  
int main() {  
 ios::sync\_with\_stdio(false);  
 cin.tie(nullptr);  
   
 // Parse input  
 int n;  
 cin >> n;  
   
 vector<int> nums(n);  
 for (int i = 0; i < n; i++) {  
 cin >> nums[i];  
 }  
   
 // Solve and output  
 cout << lengthOfLIS(nums) << endl;  
   
 return 0;  
}

### 11.1.9 Tips for Competition Day

1. **Read the Problem Carefully**: Understand all constraints and requirements before coding.
2. **Plan Before Coding**: Outline your approach and identify edge cases.
3. **Start with Easy Problems**: Build confidence and secure early points.
4. **Test Your Solution**: Create test cases for edge conditions.
5. **Submit Early, Submit Often**: Don’t wait until the last minute.
6. **Have a Debugging Strategy**: Print intermediate results or use a debugger.
7. **Keep Track of Time**: Allocate time wisely across problems.
8. **Don’t Get Stuck**: If a problem seems too difficult, move on and come back later.
9. **Learn from Others**: After the contest, study other solutions.
10. **Practice Regularly**: Skill improves with consistent practice.

### Conclusion

Competitive programming with C++ offers an excellent opportunity to strengthen your algorithmic thinking and coding skills. The techniques and patterns you learn while solving competitive programming problems transfer well to technical interviews and real-world software engineering challenges.

By mastering data structures, algorithms, and optimization techniques in the context of competitive programming, you’ll develop a deeper understanding of C++ and become a more effective problem solver in all programming contexts.

As you continue your journey in competitive programming, remember that consistent practice and review are key to improvement. Don’t be discouraged by difficult problems—each one is an opportunity to learn and grow as a programmer.

# Chapter 11: C++ in the Real World (Part 2)

## 11.2 C++ in Game Engines

Game engines are complex software frameworks that provide developers with the necessary tools to create video games efficiently. They handle core functionalities such as rendering, physics, audio, collision detection, artificial intelligence, and more. C++ has been the dominant programming language in the game industry for decades, powering most commercial game engines due to its unique combination of performance, control, and abstraction capabilities.

### 11.2.1 Why C++ Dominates Game Development

C++ offers several key advantages that make it ideal for game engine development:

#### 1. Performance

Games need to process enormous amounts of data in real-time while maintaining high frame rates (typically 30-60+ frames per second): - **Low-level optimizations**: Direct memory management allows for cache-friendly data layouts - **Minimal runtime overhead**: No garbage collection pauses or interpreter overhead - **Compiled code efficiency**: Modern C++ compilers produce highly optimized machine code - **SIMD instruction support**: Explicit vectorization for parallel data processing

#### 2. Hardware Control

C++ provides granular control over hardware resources: - **Direct memory management**: Essential for managing limited console memory - **Hardware acceleration**: Direct access to GPU APIs (DirectX, Vulkan, OpenGL) - **Platform-specific optimizations**: Ability to target specific hardware features - **Memory layout control**: Precise data alignment for performance-critical operations

#### 3. Object-Oriented Design with Performance

C++ offers a balanced approach to code organization: - **Classes and inheritance**: For organizing complex engine subsystems - **Templates**: For type-safe generic programming without runtime overhead - **Polymorphism when needed**: Virtual functions for extensible systems - **RAII pattern**: Resource management tied to object lifecycles

#### 4. Modern Features with Backward Compatibility

Modern C++ (C++11 onward) has added numerous features that benefit game development: - **Smart pointers**: Safer memory management without garbage collection - **Move semantics**: Efficient resource transfer without deep copying - **Lambda expressions**: Concise callback implementations - **Variadic templates**: Type-safe heterogeneous containers

### 11.2.2 Core Components of Game Engines in C++

#### 1. Rendering Engine

The rendering engine is responsible for drawing graphics on screen, typically implemented using a combination of C++ and shader languages (HLSL, GLSL):

class RenderingEngine {  
private:  
 std::vector<RenderableObject\*> renderQueue;  
 Camera\* activeCamera;  
 LightManager lightManager;  
   
 // Renderer state  
 ShaderProgram\* activeShader;  
 RenderTarget\* activeTarget;  
   
public:  
 void initialize() {  
 // Initialize graphics API (DirectX, OpenGL, Vulkan, etc.)  
 GraphicsAPI::initialize();  
   
 // Create default resources  
 ResourceManager::loadDefaultShaders();  
 }  
   
 void setActiveCamera(Camera\* camera) {  
 activeCamera = camera;  
 }  
   
 void submitForRendering(RenderableObject\* object) {  
 // Add object to render queue  
 renderQueue.push\_back(object);  
 }  
   
 void render() {  
 // Sort objects by material/shader for rendering efficiency  
 std::sort(renderQueue.begin(), renderQueue.end(),   
 [](const RenderableObject\* a, const RenderableObject\* b) {  
 return a->getMaterial()->getShaderID() < b->getMaterial()->getShaderID();  
 });  
   
 // Set up view and projection matrices from camera  
 Matrix4x4 viewMatrix = activeCamera->getViewMatrix();  
 Matrix4x4 projectionMatrix = activeCamera->getProjectionMatrix();  
   
 // Current shader for state tracking  
 unsigned int currentShaderId = 0;  
   
 // Render each object  
 for (RenderableObject\* object : renderQueue) {  
 // Skip objects outside the camera frustum  
 if (!activeCamera->isVisible(object->getBoundingBox())) {  
 continue;  
 }  
   
 // Set shader if different from current  
 unsigned int objectShaderId = object->getMaterial()->getShaderID();  
 if (objectShaderId != currentShaderId) {  
 ShaderProgram\* shader = ResourceManager::getShader(objectShaderId);  
 shader->bind();  
   
 // Set camera-related uniforms  
 shader->setUniformMatrix4("viewMatrix", viewMatrix);  
 shader->setUniformMatrix4("projectionMatrix", projectionMatrix);  
   
 // Set global lighting parameters  
 lightManager.setLightUniforms(shader);  
   
 currentShaderId = objectShaderId;  
 activeShader = shader;  
 }  
   
 // Set object-specific uniforms  
 activeShader->setUniformMatrix4("modelMatrix", object->getTransformMatrix());  
   
 // Bind material textures  
 object->getMaterial()->bind(activeShader);  
   
 // Draw the mesh  
 object->getMesh()->draw();  
 }  
   
 // Clear render queue for next frame  
 renderQueue.clear();  
 }  
};

#### 2. Physics Engine

The physics engine simulates physical interactions between game objects:

class PhysicsEngine {  
private:  
 std::vector<RigidBody\*> bodies;  
 std::vector<Collider\*> colliders;  
 Vector3 gravity;  
 float fixedTimestep;  
   
 // Spatial partitioning for efficient collision detection  
 OctTree spatialPartitioning;  
   
public:  
 PhysicsEngine() : gravity(0, -9.81f, 0), fixedTimestep(1.0f/60.0f) {}  
   
 void setGravity(const Vector3& g) {  
 gravity = g;  
 }  
   
 void addRigidBody(RigidBody\* body) {  
 bodies.push\_back(body);  
   
 // Add collider to spatial partitioning  
 if (body->getCollider()) {  
 spatialPartitioning.insert(body->getCollider());  
 colliders.push\_back(body->getCollider());  
 }  
 }  
   
 void removeRigidBody(RigidBody\* body) {  
 // Remove from rigid bodies list  
 auto it = std::find(bodies.begin(), bodies.end(), body);  
 if (it != bodies.end()) {  
 bodies.erase(it);  
 }  
   
 // Remove collider from spatial partitioning  
 if (body->getCollider()) {  
 spatialPartitioning.remove(body->getCollider());  
   
 // Remove from colliders list  
 auto colliderIt = std::find(colliders.begin(), colliders.end(), body->getCollider());  
 if (colliderIt != colliders.end()) {  
 colliders.erase(colliderIt);  
 }  
 }  
 }  
   
 void update(float deltaTime) {  
 // Use fixed timestep for stable physics  
 static float accumulator = 0.0f;  
 accumulator += deltaTime;  
   
 while (accumulator >= fixedTimestep) {  
 updatePhysics(fixedTimestep);  
 accumulator -= fixedTimestep;  
 }  
 }  
   
private:  
 void updatePhysics(float dt) {  
 // Apply forces to all bodies  
 for (RigidBody\* body : bodies) {  
 if (!body->isStatic()) {  
 // Apply gravity  
 body->applyForce(gravity \* body->getMass());  
   
 // Update velocity  
 Vector3 acceleration = body->getAccumulatedForce() \* body->getInverseMass();  
 body->setVelocity(body->getVelocity() + acceleration \* dt);  
   
 // Apply damping  
 body->setVelocity(body->getVelocity() \* (1.0f - body->getDamping() \* dt));  
   
 // Reset accumulated force  
 body->clearAccumulatedForce();  
 }  
 }  
   
 // Detect collisions  
 std::vector<CollisionPair> collisions = detectCollisions();  
   
 // Resolve collisions  
 resolveCollisions(collisions, dt);  
   
 // Update positions  
 for (RigidBody\* body : bodies) {  
 if (!body->isStatic()) {  
 // Update position based on velocity  
 body->setPosition(body->getPosition() + body->getVelocity() \* dt);  
 }  
 }  
   
 // Update spatial partitioning  
 updateSpatialPartitioning();  
 }  
   
 std::vector<CollisionPair> detectCollisions() {  
 std::vector<CollisionPair> collisions;  
   
 // Broad phase: use spatial partitioning to find potential collisions  
 for (Collider\* colliderA : colliders) {  
 // Query spatial partitioning for potential collisions  
 std::vector<Collider\*> potentialCollisions =   
 spatialPartitioning.query(colliderA->getBoundingBox());  
   
 // Narrow phase: detailed collision checks  
 for (Collider\* colliderB : potentialCollisions) {  
 // Skip self-collision  
 if (colliderA == colliderB) continue;  
   
 // Perform detailed collision detection  
 CollisionInfo info;  
 if (checkCollision(colliderA, colliderB, info)) {  
 collisions.push\_back({colliderA->getRigidBody(), colliderB->getRigidBody(), info});  
 }  
 }  
 }  
   
 return collisions;  
 }  
   
 void resolveCollisions(const std::vector<CollisionPair>& collisions, float dt) {  
 for (const CollisionPair& collision : collisions) {  
 RigidBody\* bodyA = collision.bodyA;  
 RigidBody\* bodyB = collision.bodyB;  
   
 // Skip if both are static  
 if (bodyA->isStatic() && bodyB->isStatic()) continue;  
   
 // Calculate relative velocity  
 Vector3 relativeVelocity = bodyB->getVelocity() - bodyA->getVelocity();  
   
 // Calculate impulse magnitude  
 float inverseMassSum = bodyA->getInverseMass() + bodyB->getInverseMass();  
 if (inverseMassSum == 0.0f) continue; // Both bodies have infinite mass  
   
 float velAlongNormal = relativeVelocity.dot(collision.info.normal);  
   
 // Don't resolve if objects are separating  
 if (velAlongNormal > 0) continue;  
   
 // Calculate coefficient of restitution (bounciness)  
 float e = std::min(bodyA->getRestitution(), bodyB->getRestitution());  
   
 // Calculate impulse scalar  
 float j = -(1.0f + e) \* velAlongNormal / inverseMassSum;  
   
 // Apply impulse  
 Vector3 impulse = collision.info.normal \* j;  
 bodyA->applyImpulse(-impulse);  
 bodyB->applyImpulse(impulse);  
   
 // Apply positional correction to prevent sinking  
 float percent = 0.2f; // Penetration percentage to correct  
 float slop = 0.01f; // Penetration allowance  
 Vector3 correction = collision.info.normal \*   
 (std::max(collision.info.penetrationDepth - slop, 0.0f) \*   
 percent / inverseMassSum);  
   
 bodyA->setPosition(bodyA->getPosition() - correction \* bodyA->getInverseMass());  
 bodyB->setPosition(bodyB->getPosition() + correction \* bodyB->getInverseMass());  
 }  
 }  
   
 void updateSpatialPartitioning() {  
 // Update the spatial partitioning structure with new object positions  
 spatialPartitioning.clear();  
 for (Collider\* collider : colliders) {  
 spatialPartitioning.insert(collider);  
 }  
 }  
};

#### 3. Entity Component System (ECS)

Modern game engines often use an ECS architecture for organizing game objects:

// Component base class  
class Component {  
public:  
 virtual ~Component() = default;  
 virtual void initialize() {}  
 virtual void update(float deltaTime) {}  
   
 Entity\* getEntity() const { return entity; }  
   
protected:  
 Entity\* entity = nullptr;  
 friend class Entity;  
};  
  
// Transform component  
class TransformComponent : public Component {  
private:  
 Vector3 position;  
 Quaternion rotation;  
 Vector3 scale;  
   
public:  
 TransformComponent() : position(0, 0, 0), rotation(), scale(1, 1, 1) {}  
   
 // Getters and setters  
 void setPosition(const Vector3& pos) { position = pos; }  
 Vector3 getPosition() const { return position; }  
   
 void setRotation(const Quaternion& rot) { rotation = rot; }  
 Quaternion getRotation() const { return rotation; }  
   
 void setScale(const Vector3& s) { scale = s; }  
 Vector3 getScale() const { return scale; }  
   
 // Generate transformation matrix  
 Matrix4x4 getTransformMatrix() const {  
 return Matrix4x4::createTransformation(position, rotation, scale);  
 }  
};  
  
// Render component  
class RenderComponent : public Component {  
private:  
 Mesh\* mesh;  
 Material\* material;  
 bool castsShadow;  
   
public:  
 RenderComponent() : mesh(nullptr), material(nullptr), castsShadow(true) {}  
   
 void initialize() override {  
 // Register with rendering system  
 RenderingSystem::getInstance().registerRenderable(this);  
 }  
   
 void setMesh(Mesh\* m) { mesh = m; }  
 Mesh\* getMesh() const { return mesh; }  
   
 void setMaterial(Material\* mat) { material = mat; }  
 Material\* getMaterial() const { return material; }  
   
 void setCastsShadow(bool cast) { castsShadow = cast; }  
 bool getCastsShadow() const { return castsShadow; }  
};  
  
// Entity class  
class Entity {  
private:  
 std::string name;  
 bool active;  
 std::vector<std::unique\_ptr<Component>> components;  
   
public:  
 Entity(const std::string& entityName = "Entity") : name(entityName), active(true) {}  
   
 template <typename T, typename... Args>  
 T\* addComponent(Args&&... args) {  
 static\_assert(std::is\_base\_of<Component, T>::value, "T must derive from Component");  
   
 // Check if component already exists  
 T\* existingComponent = getComponent<T>();  
 if (existingComponent) {  
 return existingComponent;  
 }  
   
 // Create component  
 auto component = std::make\_unique<T>(std::forward<Args>(args)...);  
 component->entity = this;  
   
 T\* componentPtr = component.get();  
 components.push\_back(std::move(component));  
   
 // Initialize the component  
 componentPtr->initialize();  
   
 return componentPtr;  
 }  
   
 template <typename T>  
 T\* getComponent() const {  
 static\_assert(std::is\_base\_of<Component, T>::value, "T must derive from Component");  
   
 for (const auto& component : components) {  
 if (auto casted = dynamic\_cast<T\*>(component.get())) {  
 return casted;  
 }  
 }  
   
 return nullptr;  
 }  
   
 void update(float deltaTime) {  
 if (!active) return;  
   
 for (auto& component : components) {  
 component->update(deltaTime);  
 }  
 }  
   
 const std::string& getName() const { return name; }  
 void setName(const std::string& newName) { name = newName; }  
   
 bool isActive() const { return active; }  
 void setActive(bool isActive) { active = isActive; }  
};  
  
// Scene class to manage entities  
class Scene {  
private:  
 std::vector<std::unique\_ptr<Entity>> entities;  
 std::string name;  
   
public:  
 Scene(const std::string& sceneName = "Scene") : name(sceneName) {}  
   
 Entity\* createEntity(const std::string& name = "Entity") {  
 auto entity = std::make\_unique<Entity>(name);  
 Entity\* entityPtr = entity.get();  
 entities.push\_back(std::move(entity));  
 return entityPtr;  
 }  
   
 void update(float deltaTime) {  
 for (auto& entity : entities) {  
 entity->update(deltaTime);  
 }  
 }  
   
 void removeEntity(Entity\* entity) {  
 auto it = std::find\_if(entities.begin(), entities.end(),  
 [entity](const std::unique\_ptr<Entity>& e) {  
 return e.get() == entity;  
 });  
   
 if (it != entities.end()) {  
 entities.erase(it);  
 }  
 }  
   
 std::vector<Entity\*> findEntitiesByName(const std::string& name) {  
 std::vector<Entity\*> result;  
   
 for (auto& entity : entities) {  
 if (entity->getName() == name) {  
 result.push\_back(entity.get());  
 }  
 }  
   
 return result;  
 }  
};

#### 4. Input System

The input system handles player interactions with the game:

// Input action types  
enum class InputActionType {  
 Pressed,  
 Released,  
 Held  
};  
  
// Input action mapping  
struct InputAction {  
 std::string name;  
 InputActionType type;  
 int keyCode;  
 std::function<void()> callback;  
};  
  
// Input system class  
class InputSystem {  
private:  
 // Key state tracking  
 std::unordered\_map<int, bool> currentKeyStates;  
 std::unordered\_map<int, bool> previousKeyStates;  
   
 // Input action mappings  
 std::vector<InputAction> actions;  
   
 // Mouse position  
 Vector2 mousePosition;  
 Vector2 previousMousePosition;  
 Vector2 mouseDelta;  
   
 // Singleton instance  
 static InputSystem\* instance;  
   
public:  
 // Get singleton instance  
 static InputSystem& getInstance() {  
 if (!instance) {  
 instance = new InputSystem();  
 }  
 return \*instance;  
 }  
   
 void initialize() {  
 // Platform-specific initialization  
 }  
   
 void update() {  
 // Update previous key states  
 previousKeyStates = currentKeyStates;  
   
 // Update previous mouse position  
 previousMousePosition = mousePosition;  
   
 // Platform-specific input polling  
 pollPlatformInput();  
   
 // Calculate mouse delta  
 mouseDelta = mousePosition - previousMousePosition;  
   
 // Process input actions  
 for (const InputAction& action : actions) {  
 int keyCode = action.keyCode;  
   
 switch (action.type) {  
 case InputActionType::Pressed:  
 if (isKeyPressed(keyCode) && action.callback) {  
 action.callback();  
 }  
 break;  
   
 case InputActionType::Released:  
 if (isKeyReleased(keyCode) && action.callback) {  
 action.callback();  
 }  
 break;  
   
 case InputActionType::Held:  
 if (isKeyDown(keyCode) && action.callback) {  
 action.callback();  
 }  
 break;  
 }  
 }  
 }  
   
 // Register an input action  
 void registerAction(const std::string& name, InputActionType type, int keyCode, std::function<void()> callback) {  
 actions.push\_back({name, type, keyCode, callback});  
 }  
   
 // Key state checking methods  
 bool isKeyDown(int keyCode) const {  
 auto it = currentKeyStates.find(keyCode);  
 return (it != currentKeyStates.end()) ? it->second : false;  
 }  
   
 bool isKeyPressed(int keyCode) const {  
 bool current = isKeyDown(keyCode);  
   
 auto it = previousKeyStates.find(keyCode);  
 bool previous = (it != previousKeyStates.end()) ? it->second : false;  
   
 return current && !previous;  
 }  
   
 bool isKeyReleased(int keyCode) const {  
 bool current = isKeyDown(keyCode);  
   
 auto it = previousKeyStates.find(keyCode);  
 bool previous = (it != previousKeyStates.end()) ? it->second : false;  
   
 return !current && previous;  
 }  
   
 // Mouse methods  
 Vector2 getMousePosition() const { return mousePosition; }  
 Vector2 getMouseDelta() const { return mouseDelta; }  
   
private:  
 InputSystem() {}  
   
 void pollPlatformInput() {  
 // Platform-specific input polling code  
 // For example, using GLFW, SDL, Win32, etc.  
   
 // This would update currentKeyStates and mousePosition  
 }  
};

### 11.2.3 Memory Management in Game Engines

Game engines require specialized memory management for performance and stability:

// Custom allocator for game objects  
class GameObjectAllocator {  
private:  
 struct MemoryChunk {  
 void\* memory;  
 size\_t size;  
 size\_t used;  
 MemoryChunk\* next;  
 };  
   
 MemoryChunk\* firstChunk;  
 size\_t chunkSize;  
   
 // Memory tracking  
 size\_t totalAllocated;  
 size\_t totalUsed;  
 int numAllocations;  
   
public:  
 GameObjectAllocator(size\_t defaultChunkSize = 1024 \* 1024) // 1MB default  
 : firstChunk(nullptr), chunkSize(defaultChunkSize),  
 totalAllocated(0), totalUsed(0), numAllocations(0) {}  
   
 ~GameObjectAllocator() {  
 // Free all allocated memory  
 MemoryChunk\* current = firstChunk;  
 while (current) {  
 MemoryChunk\* next = current->next;  
 free(current->memory);  
 free(current);  
 current = next;  
 }  
 }  
   
 // Allocate memory from the pool  
 void\* allocate(size\_t size, size\_t alignment = 16) {  
 size\_t alignmentMask = alignment - 1;  
 size\_t headerSize = sizeof(AllocationHeader);  
   
 // Find a chunk with enough space  
 MemoryChunk\* chunk = findChunkWithSpace(size + headerSize + alignment);  
 if (!chunk) {  
 // Create a new chunk if none has enough space  
 size\_t newChunkSize = std::max(chunkSize, size + headerSize + alignment);  
 chunk = createChunk(newChunkSize);  
 }  
   
 // Calculate aligned address  
 uintptr\_t baseAddress = reinterpret\_cast<uintptr\_t>(chunk->memory) + chunk->used;  
 uintptr\_t alignedAddress = (baseAddress + headerSize + alignmentMask) & ~alignmentMask;  
   
 // Calculate adjustment  
 size\_t adjustment = alignedAddress - (baseAddress + headerSize);  
   
 // Store allocation header before the aligned memory  
 AllocationHeader\* header = reinterpret\_cast<AllocationHeader\*>(alignedAddress - headerSize);  
 header->size = size;  
 header->adjustment = adjustment;  
   
 // Update chunk usage  
 chunk->used += size + headerSize + adjustment;  
   
 // Update statistics  
 totalUsed += size + headerSize + adjustment;  
 numAllocations++;  
   
 // Return aligned address  
 return reinterpret\_cast<void\*>(alignedAddress);  
 }  
   
 // Free memory  
 void deallocate(void\* ptr) {  
 if (!ptr) return;  
   
 // Get header  
 AllocationHeader\* header = reinterpret\_cast<AllocationHeader\*>(  
 reinterpret\_cast<uintptr\_t>(ptr) - sizeof(AllocationHeader));  
   
 // In a real pool allocator, we might not actually free anything here,  
 // but we can update statistics  
 totalUsed -= header->size + sizeof(AllocationHeader) + header->adjustment;  
 numAllocations--;  
 }  
   
 // Reset the allocator (for level transitions, etc.)  
 void reset() {  
 MemoryChunk\* chunk = firstChunk;  
 while (chunk) {  
 chunk->used = 0;  
 chunk = chunk->next;  
 }  
   
 totalUsed = 0;  
 numAllocations = 0;  
 }  
   
 // Statistics  
 size\_t getTotalAllocated() const { return totalAllocated; }  
 size\_t getTotalUsed() const { return totalUsed; }  
 int getNumAllocations() const { return numAllocations; }  
   
private:  
 struct AllocationHeader {  
 size\_t size;  
 size\_t adjustment;  
 };  
   
 MemoryChunk\* findChunkWithSpace(size\_t size) {  
 MemoryChunk\* chunk = firstChunk;  
 while (chunk) {  
 if (chunk->size - chunk->used >= size) {  
 return chunk;  
 }  
 chunk = chunk->next;  
 }  
 return nullptr;  
 }  
   
 MemoryChunk\* createChunk(size\_t size) {  
 // Allocate chunk header  
 MemoryChunk\* chunk = static\_cast<MemoryChunk\*>(malloc(sizeof(MemoryChunk)));  
   
 // Allocate chunk memory  
 chunk->memory = malloc(size);  
 chunk->size = size;  
 chunk->used = 0;  
   
 // Add to linked list  
 chunk->next = firstChunk;  
 firstChunk = chunk;  
   
 // Update statistics  
 totalAllocated += size;  
   
 return chunk;  
 }  
};  
  
// Custom allocator for game objects  
class GameObjectAllocator {  
private:  
 struct MemoryChunk {  
 void\* memory;  
 size\_t size;  
 size\_t used;  
 MemoryChunk\* next;  
 };  
   
 MemoryChunk\* firstChunk;  
 size\_t chunkSize;  
   
 // Memory tracking  
 size\_t totalAllocated;  
 size\_t totalUsed;  
 int numAllocations;  
   
public:  
 GameObjectAllocator(size\_t defaultChunkSize = 1024 \* 1024) // 1MB default  
 : firstChunk(nullptr), chunkSize(defaultChunkSize),  
 totalAllocated(0), totalUsed(0), numAllocations(0) {}  
   
 ~GameObjectAllocator() {  
 // Free all allocated memory  
 MemoryChunk\* current = firstChunk;  
 while (current) {  
 MemoryChunk\* next = current->next;  
 free(current->memory);  
 free(current);  
 current = next;  
 }  
 }  
   
 // Allocate memory from the pool  
 void\* allocate(size\_t size, size\_t alignment = 16) {  
 size\_t alignmentMask = alignment - 1;  
 size\_t headerSize = sizeof(AllocationHeader);  
   
 // Find a chunk with enough space  
 MemoryChunk\* chunk = findChunkWithSpace(size + headerSize + alignment);  
 if (!chunk) {  
 // Create a new chunk if none has enough space  
 size\_t newChunkSize = std::max(chunkSize, size + headerSize + alignment);  
 chunk = createChunk(newChunkSize);  
 }  
   
 // Calculate aligned address  
 uintptr\_t baseAddress = reinterpret\_cast<uintptr\_t>(chunk->memory) + chunk->used;  
 uintptr\_t alignedAddress = (baseAddress + headerSize + alignmentMask) & ~alignmentMask;  
   
 // Calculate adjustment  
 size\_t adjustment = alignedAddress - (baseAddress + headerSize);  
   
 // Store allocation header before the aligned memory  
 AllocationHeader\* header = reinterpret\_cast<AllocationHeader\*>(alignedAddress - headerSize);  
 header->size = size;  
 header->adjustment = adjustment;  
   
 // Update chunk usage  
 chunk->used += size + headerSize + adjustment;  
   
 // Update statistics  
 totalUsed += size + headerSize + adjustment;  
 numAllocations++;  
   
 // Return aligned address  
 return reinterpret\_cast<void\*>(alignedAddress);  
 }  
   
 // Free memory  
 void deallocate(void\* ptr) {  
 if (!ptr) return;  
   
 // Get header  
 AllocationHeader\* header = reinterpret\_cast<AllocationHeader\*>(  
 reinterpret\_cast<uintptr\_t>(ptr) - sizeof(AllocationHeader));  
   
 // In a real pool allocator, we might not actually free anything here,  
 // but we can update statistics  
 totalUsed -= header->size + sizeof(AllocationHeader) + header->adjustment;  
 numAllocations--;  
 }  
   
 // Reset the allocator (for level transitions, etc.)  
 void reset() {  
 MemoryChunk\* chunk = firstChunk;  
 while (chunk) {  
 chunk->used = 0;  
 chunk = chunk->next;  
 }  
   
 totalUsed = 0;  
 numAllocations = 0;  
 }  
   
 // Statistics  
 size\_t getTotalAllocated() const { return totalAllocated; }  
 size\_t getTotalUsed() const { return totalUsed; }  
 int getNumAllocations() const { return numAllocations; }  
   
private:  
 struct AllocationHeader {  
 size\_t size;  
 size\_t adjustment;  
 };  
   
 MemoryChunk\* findChunkWithSpace(size\_t size) {  
 MemoryChunk\* chunk = firstChunk;  
 while (chunk) {  
 if (chunk->size - chunk->used >= size) {  
 return chunk;  
 }  
 chunk = chunk->next;  
 }  
 return nullptr;  
 }  
   
 MemoryChunk\* createChunk(size\_t size) {  
 // Allocate chunk header  
 MemoryChunk\* chunk = static\_cast<MemoryChunk\*>(malloc(sizeof(MemoryChunk)));  
   
 // Allocate chunk memory  
 chunk->memory = malloc(size);  
 chunk->size = size;  
 chunk->used = 0;  
   
 // Add to linked list  
 chunk->next = firstChunk;  
 firstChunk = chunk;  
   
 // Update statistics  
 totalAllocated += size;  
   
 return chunk;  
 }  
};

### 11.2.4 Data-Oriented Design in Game Engines

Modern game engines often employ data-oriented design for better cache utilization:

// Traditional object-oriented approach  
class GameObject {  
private:  
 Vector3 position;  
 Quaternion rotation;  
 Vector3 scale;  
 Mesh\* mesh;  
 Material\* material;  
 PhysicsBody\* physicsBody;  
   
public:  
 void update(float deltaTime) {  
 // Update logic  
 if (physicsBody) {  
 position = physicsBody->getPosition();  
 rotation = physicsBody->getRotation();  
 }  
 }  
   
 void render() {  
 if (mesh && material) {  
 // Set transform  
 Matrix4x4 transform = Matrix4x4::createTransformMatrix(position, rotation, scale);  
   
 // Render the mesh  
 material->bind();  
 mesh->draw(transform);  
 }  
 }  
};  
  
// Data-oriented approach  
class GameObjectSystem {  
private:  
 // Components stored in structure-of-arrays for better cache coherency  
 struct {  
 std::vector<Vector3> positions;  
 std::vector<Quaternion> rotations;  
 std::vector<Vector3> scales;  
 } transforms;  
   
 struct {  
 std::vector<Mesh\*> meshes;  
 std::vector<Material\*> materials;  
 } renderers;  
   
 struct {  
 std::vector<PhysicsBody\*> bodies;  
 std::vector<bool> hasPhysics; // Flag for whether an entity has physics  
 } physics;  
   
 std::vector<bool> active; // Active status for each entity  
 int entityCount = 0;  
   
public:  
 int createEntity() {  
 int entityId = entityCount++;  
   
 // Add default component values  
 transforms.positions.push\_back(Vector3(0, 0, 0));  
 transforms.rotations.push\_back(Quaternion());  
 transforms.scales.push\_back(Vector3(1, 1, 1));  
   
 renderers.meshes.push\_back(nullptr);  
 renderers.materials.push\_back(nullptr);  
   
 physics.bodies.push\_back(nullptr);  
 physics.hasPhysics.push\_back(false);  
   
 active.push\_back(true);  
   
 return entityId;  
 }  
   
 void destroyEntity(int entityId) {  
 // In a real engine, we'd need more sophisticated removal  
 // This is a simplified version  
 if (entityId >= 0 && entityId < entityCount) {  
 active[entityId] = false;  
 }  
 }  
   
 void setTransform(int entityId, const Vector3& position, const Quaternion& rotation, const Vector3& scale) {  
 if (entityId >= 0 && entityId < entityCount) {  
 transforms.positions[entityId] = position;  
 transforms.rotations[entityId] = rotation;  
 transforms.scales[entityId] = scale;  
 }  
 }  
   
 void setRenderer(int entityId, Mesh\* mesh, Material\* material) {  
 if (entityId >= 0 && entityId < entityCount) {  
 renderers.meshes[entityId] = mesh;  
 renderers.materials[entityId] = material;  
 }  
 }  
   
 void setPhysics(int entityId, PhysicsBody\* body) {  
 if (entityId >= 0 && entityId < entityCount) {  
 physics.bodies[entityId] = body;  
 physics.hasPhysics[entityId] = (body != nullptr);  
 }  
 }  
   
 void updatePhysics(float deltaTime) {  
 // Update physics in batch  
 for (int i = 0; i < entityCount; ++i) {  
 if (!active[i] || !physics.hasPhysics[i]) continue;  
   
 PhysicsBody\* body = physics.bodies[i];  
 if (body) {  
 // Update transform from physics  
 transforms.positions[i] = body->getPosition();  
 transforms.rotations[i] = body->getRotation();  
 }  
 }  
 }  
   
 void render(RenderContext& context) {  
 // Sort by material for efficient rendering  
 std::vector<int> renderOrder;  
 for (int i = 0; i < entityCount; ++i) {  
 if (active[i] && renderers.meshes[i] && renderers.materials[i]) {  
 renderOrder.push\_back(i);  
 }  
 }  
   
 // Sort by material to minimize state changes  
 std::sort(renderOrder.begin(), renderOrder.end(),  
 [this](int a, int b) {  
 return renderers.materials[a] < renderers.materials[b];  
 });  
   
 // Render in optimized order  
 Material\* currentMaterial = nullptr;  
   
 for (int idx : renderOrder) {  
 // Apply material only when it changes  
 if (renderers.materials[idx] != currentMaterial) {  
 if (currentMaterial) {  
 currentMaterial->unbind();  
 }  
   
 currentMaterial = renderers.materials[idx];  
 currentMaterial->bind();  
 }  
   
 // Create transform matrix  
 Matrix4x4 transform = Matrix4x4::createTransformMatrix(  
 transforms.positions[idx],  
 transforms.rotations[idx],  
 transforms.scales[idx]  
 );  
   
 // Set transform uniform  
 context.setModelMatrix(transform);  
   
 // Draw mesh  
 renderers.meshes[idx]->draw();  
 }  
   
 // Unbind last material  
 if (currentMaterial) {  
 currentMaterial->unbind();  
 }  
 }  
};

### 11.2.5 Multithreading in Game Engines

Modern game engines utilize multiple threads for better performance:

class GameEngine {  
private:  
 bool running = false;  
   
 // Thread synchronization  
 std::atomic<bool> renderingCompleted = true;  
 std::atomic<bool> physicsCompleted = true;  
 std::mutex gameStateMutex;  
   
 // Game state  
 Scene\* activeScene;  
 float deltaTime;  
   
 // Threads  
 std::thread renderThread;  
 std::thread physicsThread;  
 std::thread audioThread;  
   
public:  
 void initialize() {  
 // Initialize systems  
 renderingSystem.initialize();  
 physicsSystem.initialize();  
 inputSystem.initialize();  
 audioSystem.initialize();  
   
 // Load initial scene  
 activeScene = loadScene("MainMenu");  
 }  
   
 void run() {  
 running = true;  
   
 // Start worker threads  
 renderThread = std::thread(&GameEngine::renderThreadFunc, this);  
 physicsThread = std::thread(&GameEngine::physicsThreadFunc, this);  
 audioThread = std::thread(&GameEngine::audioThreadFunc, this);  
   
 // Main thread handles game logic and input  
 mainThreadFunc();  
   
 // Wait for worker threads to finish  
 renderThread.join();  
 physicsThread.join();  
 audioThread.join();  
 }  
   
 void stop() {  
 running = false;  
 }  
   
private:  
 void mainThreadFunc() {  
 Timer timer;  
   
 while (running) {  
 // Calculate delta time  
 deltaTime = timer.getElapsedAndReset();  
   
 // Process input  
 inputSystem.update();  
   
 // Process game logic  
 {  
 std::lock\_guard<std::mutex> lock(gameStateMutex);  
   
 if (activeScene) {  
 activeScene->update(deltaTime);  
 }  
   
 // Wait for physics to complete before doing next update  
 while (!physicsCompleted.load() && running) {  
 std::this\_thread::yield();  
 }  
   
 // Reset physics flag for next frame  
 physicsCompleted.store(false);  
 }  
   
 // Cap frame rate if needed  
 limitFrameRate(60);  
 }  
 }  
   
 void renderThreadFunc() {  
 while (running) {  
 // Check if ready to render  
 if (activeScene) {  
 {  
 // Access scene data  
 std::lock\_guard<std::mutex> lock(gameStateMutex);  
   
 // Set up camera  
 Camera\* mainCamera = activeScene->getMainCamera();  
 renderingSystem.setActiveCamera(mainCamera);  
   
 // Submit objects for rendering  
 std::vector<Entity\*> entities = activeScene->getEntities();  
 for (Entity\* entity : entities) {  
 RenderComponent\* renderer = entity->getComponent<RenderComponent>();  
 TransformComponent\* transform = entity->getComponent<TransformComponent>();  
   
 if (renderer && transform) {  
 renderingSystem.submitForRendering(renderer, transform);  
 }  
 }  
 }  
   
 // Perform actual rendering  
 renderingSystem.render();  
   
 // Signal that rendering is complete  
 renderingCompleted.store(true);  
 }  
   
 // Limit render thread rate if rendering is too fast  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(1));  
 }  
 }  
   
 void physicsThreadFunc() {  
 // Fixed timestep for physics  
 const float fixedDt = 1.0f / 60.0f;  
 float accumulator = 0.0f;  
   
 while (running) {  
 {  
 std::lock\_guard<std::mutex> lock(gameStateMutex);  
   
 // Accumulate time  
 accumulator += deltaTime;  
   
 // Run physics in fixed timesteps  
 while (accumulator >= fixedDt) {  
 physicsSystem.update(fixedDt);  
 accumulator -= fixedDt;  
 }  
 }  
   
 // Signal that physics is complete  
 physicsCompleted.store(true);  
   
 // Wait a bit to avoid spinning  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(1));  
 }  
 }  
   
 void audioThreadFunc() {  
 while (running) {  
 // Process audio system updates  
 audioSystem.update();  
   
 // Audio processing is generally less frequent than rendering  
 std::this\_thread::sleep\_for(std::chrono::milliseconds(10));  
 }  
 }  
   
 void limitFrameRate(int fps) {  
 static auto lastFrameTime = std::chrono::high\_resolution\_clock::now();  
   
 // Calculate target frame duration  
 const std::chrono::duration<double> targetFrameDuration(1.0 / fps);  
   
 // Get current time  
 auto now = std::chrono::high\_resolution\_clock::now();  
   
 // Calculate elapsed time  
 std::chrono::duration<double> elapsed = now - lastFrameTime;  
   
 // Sleep if we're running too fast  
 if (elapsed < targetFrameDuration) {  
 std::this\_thread::sleep\_for(targetFrameDuration - elapsed);  
 }  
   
 // Update last frame time  
 lastFrameTime = std::chrono::high\_resolution\_clock::now();  
 }  
};

### 11.2.6 Popular Game Engines Using C++

#### Unreal Engine

Epic Games’ Unreal Engine is one of the most popular commercial game engines, written primarily in C++. It features:

1. **Blueprint Visual Scripting**: A visual scripting system that compiles down to C++
2. **Comprehensive Editor**: A full-featured editor with real-time preview
3. **Advanced Rendering**: PBR rendering, dynamic lighting, global illumination
4. **Physics**: Integrated PhysX physics engine
5. **Networking**: Robust multiplayer framework
6. **Cross-platform**: Supports multiple platforms including PC, consoles, mobile, VR

Example Unreal Engine C++ class:

// Character class in Unreal Engine  
#include "MyCharacter.h"  
#include "Camera/CameraComponent.h"  
#include "Components/CapsuleComponent.h"  
#include "Components/InputComponent.h"  
#include "GameFramework/CharacterMovementComponent.h"  
  
// Constructor  
AMyCharacter::AMyCharacter()  
{  
 // Set this character to call Tick() every frame  
 PrimaryActorTick.bCanEverTick = true;  
  
 // Create a camera component  
 FirstPersonCameraComponent = CreateDefaultSubobject<UCameraComponent>(TEXT("FirstPersonCamera"));  
 FirstPersonCameraComponent->SetupAttachment(GetCapsuleComponent());  
 FirstPersonCameraComponent->SetRelativeLocation(FVector(0.0f, 0.0f, 50.0f + BaseEyeHeight));  
 FirstPersonCameraComponent->bUsePawnControlRotation = true;  
  
 // Set character movement properties  
 GetCharacterMovement()->JumpZVelocity = 600.0f;  
 GetCharacterMovement()->AirControl = 0.2f;  
}  
  
// Called when game starts or when spawned  
void AMyCharacter::BeginPlay()  
{  
 Super::BeginPlay();  
   
 // Initialize health  
 CurrentHealth = MaxHealth;  
}  
  
// Called every frame  
void AMyCharacter::Tick(float DeltaTime)  
{  
 Super::Tick(DeltaTime);  
  
 // Update health regeneration  
 if (bRegenerateHealth && CurrentHealth < MaxHealth)  
 {  
 RegenerationTimer += DeltaTime;  
 if (RegenerationTimer >= RegenerationInterval)  
 {  
 RegenerationTimer = 0.0f;  
 CurrentHealth = FMath::Min(CurrentHealth + RegenerationAmount, MaxHealth);  
 }  
 }  
}  
  
// Called to bind functionality to input  
void AMyCharacter::SetupPlayerInputComponent(UInputComponent\* PlayerInputComponent)  
{  
 Super::SetupPlayerInputComponent(PlayerInputComponent);  
  
 // Set up movement bindings  
 PlayerInputComponent->BindAxis("MoveForward", this, &AMyCharacter::MoveForward);  
 PlayerInputComponent->BindAxis("MoveRight", this, &AMyCharacter::MoveRight);  
  
 // Set up look bindings  
 PlayerInputComponent->BindAxis("Turn", this, &APawn::AddControllerYawInput);  
 PlayerInputComponent->BindAxis("LookUp", this, &APawn::AddControllerPitchInput);  
  
 // Set up action bindings  
 PlayerInputComponent->BindAction("Jump", IE\_Pressed, this, &ACharacter::Jump);  
 PlayerInputComponent->BindAction("Fire", IE\_Pressed, this, &AMyCharacter::StartFire);  
 PlayerInputComponent->BindAction("Fire", IE\_Released, this, &AMyCharacter::StopFire);  
}  
  
// Handle moving forward/backward  
void AMyCharacter::MoveForward(float Value)  
{  
 if (Value != 0.0f)  
 {  
 // Find out which way is forward  
 const FRotator Rotation = Controller->GetControlRotation();  
 const FRotator YawRotation(0, Rotation.Yaw, 0);  
  
 // Get forward vector  
 const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);  
 AddMovementInput(Direction, Value);  
 }  
}  
  
// Handle moving right/left  
void AMyCharacter::MoveRight(float Value)  
{  
 if (Value != 0.0f)  
 {  
 // Find out which way is right  
 const FRotator Rotation = Controller->GetControlRotation();  
 const FRotator YawRotation(0, Rotation.Yaw, 0);  
   
 // Get right vector   
 const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);  
 AddMovementInput(Direction, Value);  
 }  
}  
  
// Start weapon firing  
void AMyCharacter::StartFire()  
{  
 if (WeaponComponent)  
 {  
 WeaponComponent->StartFire();  
 }  
}  
  
// Stop weapon firing  
void AMyCharacter::StopFire()  
{  
 if (WeaponComponent)  
 {  
 WeaponComponent->StopFire();  
 }  
}  
  
// Apply damage to this character  
float AMyCharacter::TakeDamage(float DamageAmount, FDamageEvent const& DamageEvent, AController\* EventInstigator, AActor\* DamageCauser)  
{  
 // Apply base damage calculation  
 float ActualDamage = Super::TakeDamage(DamageAmount, DamageEvent, EventInstigator, DamageCauser);  
  
 if (ActualDamage > 0.0f)  
 {  
 // Decrease health  
 CurrentHealth = FMath::Max(CurrentHealth - ActualDamage, 0.0f);  
  
 // Reset regeneration timer when damaged  
 RegenerationTimer = 0.0f;  
  
 // Check if dead  
 if (CurrentHealth <= 0.0f)  
 {  
 Die(DamageCauser);  
 }  
 else  
 {  
 // Play hit reaction  
 PlayHitReaction();  
 }  
 }  
  
 return ActualDamage;  
}  
  
// Handle character death  
void AMyCharacter::Die(AActor\* Killer)  
{  
 // Disable input  
 DisableInput(nullptr);  
  
 // Set appropriate collision  
 GetCapsuleComponent()->SetCollisionEnabled(ECollisionEnabled::NoCollision);  
 GetMesh()->SetCollisionEnabled(ECollisionEnabled::PhysicsOnly);  
 GetMesh()->SetSimulatePhysics(true);  
  
 // Broadcast death event  
 OnCharacterDied.Broadcast(this, Killer);  
  
 // Set timer to respawn or destroy  
 GetWorldTimerManager().SetTimer(  
 RespawnTimerHandle,  
 this,  
 &AMyCharacter::HandleRespawn,  
 RespawnDelay,  
 false  
 );  
}

#### Unity Engine

While Unity primarily uses C# for scripting, its core engine is written in C++. Unity developers can also create native plugins in C++ for performance-critical features.

Example of a C++ native plugin for Unity:

#include "UnityNativePlugin.h"  
#include "Unity/IUnityGraphics.h"  
#include "Unity/IUnityInterface.h"  
  
// Plugin instance data  
static IUnityInterfaces\* s\_UnityInterfaces = nullptr;  
static IUnityGraphics\* s\_UnityGraphics = nullptr;  
  
// Function to handle rendering events from Unity  
static void UNITY\_INTERFACE\_API OnRenderEvent(int eventID)  
{  
 // Plugin-specific rendering code  
 // This would typically interact with the graphics API (OpenGL, DirectX, etc.)  
 switch (eventID)  
 {  
 case 0: // Custom post-processing effect  
 ApplyCustomPostProcess();  
 break;  
   
 case 1: // Custom shader effect  
 ApplyCustomShaderEffect();  
 break;  
 }  
}  
  
// Apply a custom post-processing effect  
void ApplyCustomPostProcess()  
{  
 // Implementation depends on the graphics API in use  
}  
  
// Apply a custom shader effect  
void ApplyCustomShaderEffect()  
{  
 // Implementation depends on the graphics API in use  
}  
  
// Unity plugin load event  
extern "C" void UNITY\_INTERFACE\_EXPORT UNITY\_INTERFACE\_API  
UnityPluginLoad(IUnityInterfaces\* unityInterfaces)  
{  
 s\_UnityInterfaces = unityInterfaces;  
 s\_UnityGraphics = unityInterfaces->Get<IUnityGraphics>();  
   
 // Register for graphics device events  
 s\_UnityGraphics->RegisterDeviceEventCallback(OnGraphicsDeviceEvent);  
   
 // Run OnGraphicsDeviceEvent(initialize) manually once  
 OnGraphicsDeviceEvent(kUnityGfxDeviceEventInitialize);  
}  
  
// Unity plugin unload event  
extern "C" void UNITY\_INTERFACE\_EXPORT UNITY\_INTERFACE\_API  
UnityPluginUnload()  
{  
 if (s\_UnityGraphics)  
 {  
 s\_UnityGraphics->UnregisterDeviceEventCallback(OnGraphicsDeviceEvent);  
 }  
   
 s\_UnityInterfaces = nullptr;  
 s\_UnityGraphics = nullptr;  
}  
  
// Function exposed to Unity  
extern "C" UnityRenderingEvent UNITY\_INTERFACE\_EXPORT UNITY\_INTERFACE\_API  
GetRenderEventFunc()  
{  
 return OnRenderEvent;  
}  
  
// Custom raycast implementation exposed to Unity  
extern "C" bool UNITY\_INTERFACE\_EXPORT UNITY\_INTERFACE\_API  
CustomRaycast(float originX, float originY, float originZ,  
 float dirX, float dirY, float dirZ,  
 float\* hitX, float\* hitY, float\* hitZ)  
{  
 // Implement custom raycast algorithm  
 // This could be an optimized version or use specialized acceleration structures  
   
 // For this example, let's do a simple sphere intersection  
 const float sphereX = 0.0f, sphereY = 0.0f, sphereZ = 0.0f;  
 const float sphereRadius = 1.0f;  
   
 float ox = originX - sphereX;  
 float oy = originY - sphereY;  
 float oz = originZ - sphereZ;  
   
 float a = dirX \* dirX + dirY \* dirY + dirZ \* dirZ;  
 float b = 2.0f \* (ox \* dirX + oy \* dirY + oz \* dirZ);  
 float c = ox \* ox + oy \* oy + oz \* oz - sphereRadius \* sphereRadius;  
   
 float discriminant = b \* b - 4.0f \* a \* c;  
 if (discriminant < 0.0f)  
 {  
 return false; // No intersection  
 }  
   
 // Calculate intersection point  
 float t = (-b - sqrt(discriminant)) / (2.0f \* a);  
 if (t < 0.0f)   
 {  
 t = (-b + sqrt(discriminant)) / (2.0f \* a);  
 if (t < 0.0f)  
 {  
 return false; // Intersection behind ray origin  
 }  
 }  
   
 // Calculate hit point  
 \*hitX = originX + dirX \* t;  
 \*hitY = originY + dirY \* t;  
 \*hitZ = originZ + dirZ \* t;  
   
 return true;  
}

### 11.2.7 Future Trends in Game Engine Development

As game engines evolve, several trends are emerging:

1. **Data-Oriented Design**: Moving away from traditional OOP toward more cache-friendly DOD approaches.
2. **Ray Tracing**: Hardware-accelerated ray tracing for realistic lighting and reflections.
3. **Machine Learning Integration**: Using ML for better animations, NPC behavior, and content generation.
4. **Cross-Platform Development**: Engines are focusing on supporting multiple platforms from a single codebase.
5. **Cloud-Based Technologies**: Distributed rendering and computation for more complex simulations.
6. **Procedural Generation**: Advanced algorithms for creating environments, characters, and gameplay elements.
7. **Extended Reality Support**: Better integration of AR/VR/XR technologies.

C++ continues to play a crucial role in these advancements due to its performance characteristics and direct hardware access capabilities.

### 11.2.8 Challenges in Game Engine Development

Game engine developers face several challenges:

1. **Performance vs. Abstraction**: Balancing high-level abstractions with performance requirements.
2. **Cross-Platform Support**: Supporting multiple hardware architectures and operating systems.
3. **Graphics API Evolution**: Keeping up with rapidly changing graphics APIs like DirectX, Vulkan, Metal.
4. **Memory Management**: Preventing fragmentation and ensuring predictable performance.
5. **Multithreading**: Efficiently utilizing multiple cores without introducing race conditions or deadlocks.
6. **Build Times**: Managing large codebases with long compilation times.
7. **Backward Compatibility**: Supporting older content while adding new features.
8. **Asset Pipeline**: Efficiently processing and optimizing assets for different platforms.

C++ addresses many of these challenges through its performance characteristics and flexibility, though it introduces complexity that must be managed through good architecture and coding practices.

### 11.2.9 Resources for Game Engine Developers

For those interested in learning more about game engine development:

1. **Books**:
   * “Game Engine Architecture” by Jason Gregory
   * “Game Programming Patterns” by Robert Nystrom
   * “Physics for Game Developers” by David M. Bourg
2. **Open Source Engines**:
   * Godot Engine: A fully open-source game engine in C++
   * OGRE: Open source graphics rendering engine
   * Box2D: 2D physics engine
3. **Conferences**:
   * Game Developers Conference (GDC)
   * SIGGRAPH
   * CppCon often has game development talks
4. **Online Resources**:
   * Unreal Engine documentation and source code
   * Gamasutra articles on game engine development
   * Graphics programming sites like LearnOpenGL

# Chapter 11: C++ in the Real World (Part 3)

## 11.2 C++ in Game Engines

Game engines are complex software frameworks that provide developers with tools and systems to create video games without having to build everything from scratch. They typically handle rendering, physics, collision detection, audio, input systems, animation, and more. C++ has been the dominant language in game engine development for decades due to its unique combination of performance and flexibility.

### 11.2.1 Why C++ Dominates Game Development

Several key characteristics make C++ the preferred language for game engine development:

#### 1. Performance

Games are performance-critical applications that must process vast amounts of data in real-time while maintaining high frame rates (typically 30-60+ frames per second):

* **Low-level control**: C++ provides direct memory management without garbage collection pauses
* **Compiled language efficiency**: C++ compiles to highly optimized native code
* **Minimal overhead**: No interpreter or virtual machine adding performance costs
* **SIMD/Vectorization**: C++ allows for explicit use of CPU vector instructions (SSE, AVX)
* **Cache-friendly code**: C++ developers can control memory layout for optimal cache utilization

#### 2. Hardware Access

Game engines need to interact directly with graphics hardware and other system components:

* **Graphics APIs**: Direct access to DirectX, Vulkan, OpenGL, Metal
* **Memory-mapped I/O**: Ability to interact with hardware registers
* **Hardware features**: Access to specialized hardware features like GPU compute
* **Low-level optimization**: Fine control over how code maps to hardware instructions

#### 3. Memory Management Control

Games often have strict memory constraints, especially on consoles and mobile devices:

* **Custom allocators**: Game engines typically implement specialized memory allocators
* **Memory pooling**: Pre-allocation of memory pools to avoid fragmentation
* **Deterministic cleanup**: RAII pattern ensures resources are properly managed
* **Memory layout control**: Explicit control over object layout and alignment

### 11.2.2 Core Components of Game Engines

Let’s explore the major components of a game engine and how they’re implemented in C++:

#### 1. Rendering Engine

The rendering system is responsible for displaying graphics on screen:

class RenderingEngine {  
private:  
 std::vector<RenderableObject\*> renderQueue;  
 Camera\* activeCamera;  
 LightManager\* lightManager;  
 ShaderManager\* shaderManager;  
   
 // OpenGL/DirectX/Vulkan context and resources  
 GraphicsContext\* graphicsContext;  
   
public:  
 void initialize(GraphicsAPI api = GraphicsAPI::OpenGL) {  
 // Initialize graphics context based on API  
 graphicsContext = createGraphicsContext(api);  
   
 // Initialize managers  
 shaderManager = new ShaderManager(graphicsContext);  
 lightManager = new LightManager();  
   
 // Load standard shaders  
 shaderManager->loadShader("standard", "shaders/standard.vert", "shaders/standard.frag");  
 shaderManager->loadShader("shadow", "shaders/shadow.vert", "shaders/shadow.frag");  
 }  
   
 void setActiveCamera(Camera\* camera) {  
 activeCamera = camera;  
 }  
   
 void submitForRendering(RenderableObject\* object) {  
 // Add object to render queue for this frame  
 renderQueue.push\_back(object);  
 }  
   
 void render() {  
 // Sort objects by material/shader for efficient state changes  
 std::sort(renderQueue.begin(), renderQueue.end(),   
 [](const RenderableObject\* a, const RenderableObject\* b) {  
 return a->getMaterial()->getShaderID() < b->getMaterial()->getShaderID();  
 });  
   
 // Set up view and projection matrices  
 Matrix4x4 viewMatrix = activeCamera->getViewMatrix();  
 Matrix4x4 projectionMatrix = activeCamera->getProjectionMatrix();  
   
 // Render shadow maps first (if enabled)  
 renderShadowMaps();  
   
 // Clear framebuffer  
 graphicsContext->clear(Color(0.1f, 0.1f, 0.2f, 1.0f));  
   
 // State tracking for batching optimization  
 Material\* currentMaterial = nullptr;  
 Shader\* currentShader = nullptr;  
   
 // Render all objects  
 for (RenderableObject\* object : renderQueue) {  
 // Skip objects outside view frustum  
 if (!activeCamera->isVisible(object->getBoundingBox())) {  
 continue;  
 }  
   
 // Apply material and shader if different from current  
 Material\* material = object->getMaterial();  
 if (material != currentMaterial) {  
 if (material->getShader() != currentShader) {  
 currentShader = material->getShader();  
 currentShader->bind();  
   
 // Set global uniforms that are the same for all objects with this shader  
 currentShader->setUniform("viewMatrix", viewMatrix);  
 currentShader->setUniform("projectionMatrix", projectionMatrix);  
   
 // Set lighting uniforms  
 lightManager->setLightUniforms(currentShader);  
 }  
   
 // Set material properties  
 material->apply();  
 currentMaterial = material;  
 }  
   
 // Set per-object uniforms  
 currentShader->setUniform("modelMatrix", object->getTransformMatrix());  
   
 // Draw the object's mesh  
 object->getMesh()->draw(graphicsContext);  
 }  
   
 // Clear render queue for next frame  
 renderQueue.clear();  
   
 // Swap buffers  
 graphicsContext->swapBuffers();  
 }  
   
private:  
 void renderShadowMaps() {  
 // Render the scene from light's perspective for shadow mapping  
 // (Implementation details omitted for brevity)  
 }  
};

#### 2. Physics Engine

The physics system simulates physical interactions between game objects:

class PhysicsEngine {  
private:  
 std::vector<RigidBody\*> bodies;  
 std::vector<Collider\*> colliders;  
 Vector3 gravity;  
   
 // Spatial partitioning for broad-phase collision detection  
 OctTree spatialPartitioning;  
   
 // Fixed timestep for stable physics  
 float fixedTimeStep;  
 float accumulator;  
   
public:  
 PhysicsEngine() : gravity(0, -9.81f, 0), fixedTimeStep(1.0f/60.0f), accumulator(0) {}  
   
 void setGravity(const Vector3& g) {  
 gravity = g;  
 }  
   
 void addRigidBody(RigidBody\* body) {  
 bodies.push\_back(body);  
   
 // Add body's collider to spatial partitioning  
 if (body->getCollider()) {  
 colliders.push\_back(body->getCollider());  
 spatialPartitioning.insert(body->getCollider());  
 }  
 }  
   
 void removeRigidBody(RigidBody\* body) {  
 // Remove from bodies list  
 auto it = std::find(bodies.begin(), bodies.end(), body);  
 if (it != bodies.end()) {  
 bodies.erase(it);  
 }  
   
 // Remove collider from spatial partitioning  
 if (body->getCollider()) {  
 spatialPartitioning.remove(body->getCollider());  
   
 auto colliderIt = std::find(colliders.begin(), colliders.end(), body->getCollider());  
 if (colliderIt != colliders.end()) {  
 colliders.erase(colliderIt);  
 }  
 }  
 }  
   
 void update(float deltaTime) {  
 // Accumulate time for fixed timestep physics  
 accumulator += deltaTime;  
   
 // Run physics in fixed timesteps for stability  
 while (accumulator >= fixedTimeStep) {  
 updatePhysics(fixedTimeStep);  
 accumulator -= fixedTimeStep;  
 }  
 }  
   
private:  
 void updatePhysics(float dt) {  
 // Apply forces (like gravity)  
 for (RigidBody\* body : bodies) {  
 if (!body->isStatic()) {  
 body->applyForce(gravity \* body->getMass());  
 }  
 }  
   
 // Update velocities based on forces  
 for (RigidBody\* body : bodies) {  
 if (!body->isStatic()) {  
 // v = v₀ + a\*t  
 Vector3 acceleration = body->getAccumulatedForce() \* body->getInverseMass();  
 body->setVelocity(body->getVelocity() + acceleration \* dt);  
   
 // Apply damping (simulates air resistance)  
 body->setVelocity(body->getVelocity() \* (1.0f - body->getDamping() \* dt));  
   
 // Reset accumulated force  
 body->clearAccumulatedForce();  
 }  
 }  
   
 // Detect and resolve collisions  
 std::vector<CollisionData> collisions = detectCollisions();  
 resolveCollisions(collisions, dt);  
   
 // Update positions  
 for (RigidBody\* body : bodies) {  
 if (!body->isStatic()) {  
 // p = p₀ + v\*t  
 Vector3 newPosition = body->getPosition() + body->getVelocity() \* dt;  
 body->setPosition(newPosition);  
 }  
 }  
   
 // Update spatial partitioning  
 updateSpatialPartitioning();  
 }  
   
 std::vector<CollisionData> detectCollisions() {  
 std::vector<CollisionData> collisions;  
   
 // Broad-phase: Find potentially colliding pairs using spatial partitioning  
 for (Collider\* colliderA : colliders) {  
 // Query for potential collision candidates  
 std::vector<Collider\*> potentialColliders =   
 spatialPartitioning.queryPotentialCollisions(colliderA);  
   
 // Narrow-phase: Detailed collision test for each pair  
 for (Collider\* colliderB : potentialColliders) {  
 // Skip self-collision  
 if (colliderA == colliderB) continue;  
   
 // Skip if both objects are static  
 if (colliderA->getRigidBody()->isStatic() &&   
 colliderB->getRigidBody()->isStatic()) {  
 continue;  
 }  
   
 // Perform detailed collision check  
 CollisionData collision;  
 if (checkCollision(colliderA, colliderB, collision)) {  
 collisions.push\_back(collision);  
 }  
 }  
 }  
   
 return collisions;  
 }  
   
 void resolveCollisions(const std::vector<CollisionData>& collisions, float dt) {  
 for (const CollisionData& collision : collisions) {  
 RigidBody\* bodyA = collision.colliderA->getRigidBody();  
 RigidBody\* bodyB = collision.colliderB->getRigidBody();  
   
 // Skip if one of the bodies has been removed  
 if (!bodyA || !bodyB) continue;  
   
 // Calculate relative velocity  
 Vector3 relativeVelocity = bodyB->getVelocity() - bodyA->getVelocity();  
   
 // Calculate velocity along normal  
 float velAlongNormal = relativeVelocity.dot(collision.normal);  
   
 // Skip if objects are moving apart  
 if (velAlongNormal > 0) continue;  
   
 // Calculate bounciness (coefficient of restitution)  
 float restitution = std::min(bodyA->getRestitution(), bodyB->getRestitution());  
   
 // Calculate impulse magnitude  
 float inverseMassA = bodyA->isStatic() ? 0.0f : bodyA->getInverseMass();  
 float inverseMassB = bodyB->isStatic() ? 0.0f : bodyB->getInverseMass();  
 float inverseMassSum = inverseMassA + inverseMassB;  
   
 if (inverseMassSum == 0.0f) continue; // Both bodies are static  
   
 float j = -(1.0f + restitution) \* velAlongNormal;  
 j /= inverseMassSum;  
   
 // Apply impulse  
 Vector3 impulse = collision.normal \* j;  
   
 if (!bodyA->isStatic()) {  
 bodyA->setVelocity(bodyA->getVelocity() - impulse \* bodyA->getInverseMass());  
 }  
   
 if (!bodyB->isStatic()) {  
 bodyB->setVelocity(bodyB->getVelocity() + impulse \* bodyB->getInverseMass());  
 }  
   
 // Position correction to prevent sinking (using the "bias" method)  
 const float percent = 0.2f; // Penetration percentage to correct  
 const float slop = 0.01f; // Allow small penetration for stability  
   
 Vector3 correction = collision.normal \*   
 (std::max(collision.penetrationDepth - slop, 0.0f) /   
 inverseMassSum \* percent);  
   
 if (!bodyA->isStatic()) {  
 bodyA->setPosition(bodyA->getPosition() - correction \* bodyA->getInverseMass());  
 }  
   
 if (!bodyB->isStatic()) {  
 bodyB->setPosition(bodyB->getPosition() + correction \* bodyB->getInverseMass());  
 }  
 }  
 }  
   
 void updateSpatialPartitioning() {  
 spatialPartitioning.clear();  
 for (Collider\* collider : colliders) {  
 spatialPartitioning.insert(collider);  
 }  
 }  
};

#### 3. Entity-Component System (ECS)

Modern game engines often use a component-based architecture for organizing game objects:

// Base Component class  
class Component {  
public:  
 virtual ~Component() = default;  
   
 virtual void initialize() {}  
 virtual void update(float deltaTime) {}  
   
 Entity\* getEntity() const { return entity; }  
   
 friend class Entity;  
   
protected:  
 Entity\* entity = nullptr;  
};  
  
// Transform Component  
class TransformComponent : public Component {  
private:  
 Vector3 position;  
 Quaternion rotation;  
 Vector3 scale;  
   
 // Caching  
 Matrix4x4 worldMatrix;  
 bool worldMatrixDirty = true;  
   
 // Hierarchy  
 TransformComponent\* parent = nullptr;  
 std::vector<TransformComponent\*> children;  
   
public:  
 TransformComponent() : position(0,0,0), rotation(), scale(1,1,1) {}  
   
 void setPosition(const Vector3& pos) {  
 position = pos;  
 setDirty();  
 }  
   
 void setRotation(const Quaternion& rot) {  
 rotation = rot;  
 setDirty();  
 }  
   
 void setScale(const Vector3& s) {  
 scale = s;  
 setDirty();  
 }  
   
 Vector3 getPosition() const { return position; }  
 Quaternion getRotation() const { return rotation; }  
 Vector3 getScale() const { return scale; }  
   
 // Parent-child relationship methods  
 void setParent(TransformComponent\* newParent) {  
 if (parent == newParent) return;  
   
 // Remove from old parent  
 if (parent) {  
 auto it = std::find(parent->children.begin(), parent->children.end(), this);  
 if (it != parent->children.end()) {  
 parent->children.erase(it);  
 }  
 }  
   
 // Set new parent  
 parent = newParent;  
   
 // Add to new parent  
 if (parent) {  
 parent->children.push\_back(this);  
 }  
   
 setDirty();  
 }  
   
 // World space transformations  
 Matrix4x4 getWorldMatrix() {  
 if (worldMatrixDirty) {  
 updateWorldMatrix();  
 }  
 return worldMatrix;  
 }  
   
 Vector3 getWorldPosition() {  
 return getWorldMatrix().getTranslation();  
 }  
   
 Quaternion getWorldRotation() {  
 return getWorldMatrix().getRotation();  
 }  
   
private:  
 void updateWorldMatrix() {  
 // Create local transform matrix  
 worldMatrix = Matrix4x4::createTransformMatrix(position, rotation, scale);  
   
 // Multiply by parent's world matrix if we have a parent  
 if (parent) {  
 worldMatrix = parent->getWorldMatrix() \* worldMatrix;  
 }  
   
 worldMatrixDirty = false;  
 }  
   
 void setDirty() {  
 worldMatrixDirty = true;  
   
 // Also mark all children dirty  
 for (TransformComponent\* child : children) {  
 child->setDirty();  
 }  
 }  
};  
  
// Mesh Renderer Component  
class MeshRendererComponent : public Component {  
private:  
 Mesh\* mesh;  
 Material\* material;  
 bool castsShadows;  
   
public:  
 MeshRendererComponent() : mesh(nullptr), material(nullptr), castsShadows(true) {}  
   
 void initialize() override {  
 // Register with rendering system when component is added to entity  
 RenderingSystem::getInstance().registerRenderer(this);  
 }  
   
 void setMesh(Mesh\* m) { mesh = m; }  
 Mesh\* getMesh() const { return mesh; }  
   
 void setMaterial(Material\* mat) { material = mat; }  
 Material\* getMaterial() const { return material; }  
   
 void setCastsShadows(bool cast) { castsShadows = cast; }  
 bool getCastsShadows() const { return castsShadows; }  
   
 // Used by the rendering system  
 void render(RenderContext& context) {  
 if (!mesh || !material) return;  
   
 // Get transform  
 TransformComponent\* transform = entity->getComponent<TransformComponent>();  
 if (!transform) return;  
   
 // Set up model matrix  
 Matrix4x4 modelMatrix = transform->getWorldMatrix();  
   
 // Apply material  
 material->apply(context);  
   
 // Set model matrix uniform  
 material->getShader()->setUniform("modelMatrix", modelMatrix);  
   
 // Draw mesh  
 mesh->draw();  
 }  
};  
  
// Physics Body Component  
class RigidBodyComponent : public Component {  
private:  
 RigidBody\* rigidBody;  
   
public:  
 RigidBodyComponent() : rigidBody(nullptr) {}  
   
 ~RigidBodyComponent() {  
 if (rigidBody) {  
 PhysicsSystem::getInstance().removeRigidBody(rigidBody);  
 delete rigidBody;  
 }  
 }  
   
 void initialize() override {  
 // Create physics body  
 rigidBody = new RigidBody();  
   
 // Set initial transform  
 TransformComponent\* transform = entity->getComponent<TransformComponent>();  
 if (transform) {  
 rigidBody->setPosition(transform->getPosition());  
 rigidBody->setRotation(transform->getRotation());  
 }  
   
 // Register with physics system  
 PhysicsSystem::getInstance().addRigidBody(rigidBody);  
 }  
   
 void update(float deltaTime) override {  
 // Sync transform with physics body  
 TransformComponent\* transform = entity->getComponent<TransformComponent>();  
 if (!transform || !rigidBody) return;  
   
 // Update entity transform from physics simulation  
 transform->setPosition(rigidBody->getPosition());  
 transform->setRotation(rigidBody->getRotation());  
 }  
   
 RigidBody\* getRigidBody() const { return rigidBody; }  
   
 void setMass(float mass) {  
 if (rigidBody) rigidBody->setMass(mass);  
 }  
   
 void applyForce(const Vector3& force) {  
 if (rigidBody) rigidBody->applyForce(force);  
 }  
   
 void applyImpulse(const Vector3& impulse) {  
 if (rigidBody) rigidBody->applyImpulse(impulse);  
 }  
   
 void setStatic(bool isStatic) {  
 if (rigidBody) rigidBody->setStatic(isStatic);  
 }  
};  
  
// Entity class  
class Entity {  
private:  
 std::string name;  
 std::vector<std::unique\_ptr<Component>> components;  
 bool active;  
 Scene\* scene;  
   
public:  
 Entity(const std::string& name = "Entity")   
 : name(name), active(true), scene(nullptr) {  
 // Entities always have a transform component  
 addComponent<TransformComponent>();  
 }  
   
 // Add a component  
 template<typename T, typename... Args>  
 T\* addComponent(Args&&... args) {  
 static\_assert(std::is\_base\_of<Component, T>::value,   
 "T must derive from Component");  
   
 // Check if component of this type already exists  
 T\* existingComponent = getComponent<T>();  
 if (existingComponent) {  
 return existingComponent;  
 }  
   
 // Create component  
 auto component = std::make\_unique<T>(std::forward<Args>(args)...);  
 T\* componentPtr = component.get();  
   
 // Set component's entity  
 componentPtr->entity = this;  
   
 // Store component  
 components.push\_back(std::move(component));  
   
 // Initialize component  
 componentPtr->initialize();  
   
 return componentPtr;  
 }  
   
 // Get a component by type  
 template<typename T>  
 T\* getComponent() const {  
 static\_assert(std::is\_base\_of<Component, T>::value,   
 "T must derive from Component");  
   
 for (const auto& component : components) {  
 if (T\* casted = dynamic\_cast<T\*>(component.get())) {  
 return casted;  
 }  
 }  
   
 return nullptr;  
 }  
   
 // Update all components  
 void update(float deltaTime) {  
 if (!active) return;  
   
 for (auto& component : components) {  
 component->update(deltaTime);  
 }  
 }  
   
 // Get entity name  
 const std::string& getName() const { return name; }  
   
 // Set entity name  
 void setName(const std::string& newName) { name = newName; }  
   
 // Check if entity is active  
 bool isActive() const { return active; }  
   
 // Set entity active state  
 void setActive(bool isActive) { active = isActive; }  
   
 // Get scene  
 Scene\* getScene() const { return scene; }  
   
 // Friend class Scene can set the scene pointer  
 friend class Scene;  
};

#### 4. Input System

The input system handles user interactions like keyboard, mouse, and gamepad input:

class InputSystem {  
public:  
 enum class KeyState {  
 Released, // Not currently pressed  
 Pressed, // Just pressed this frame  
 Held // Held down from previous frame  
 };  
   
private:  
 // Key state tracking  
 std::unordered\_map<int, bool> currentKeyStates;  
 std::unordered\_map<int, bool> previousKeyStates;  
   
 // Mouse state tracking  
 Vector2 mousePosition;  
 Vector2 previousMousePosition;  
 Vector2 mouseDelta;  
   
 // Gamepad state tracking  
 struct GamepadState {  
 bool connected = false;  
 std::array<float, 6> axes = {}; // Left X/Y, Right X/Y, Triggers L/R  
 std::array<bool, 15> buttons = {};  
 };  
   
 std::array<GamepadState, 4> gamepads;  
   
 // Singleton instance  
 static InputSystem\* instance;  
   
public:  
 // Singleton accessor  
 static InputSystem& getInstance() {  
 if (!instance) {  
 instance = new InputSystem();  
 }  
 return \*instance;  
 }  
   
 // Call at start of frame to update input state  
 void update() {  
 // Save previous states  
 previousKeyStates = currentKeyStates;  
 previousMousePosition = mousePosition;  
   
 // Platform-specific input polling would go here  
 pollPlatformInput();  
   
 // Calculate mouse delta  
 mouseDelta = mousePosition - previousMousePosition;  
 }  
   
 // Keyboard state checks  
 KeyState getKeyState(int keyCode) const {  
 bool previouslyPressed = wasKeyPressed(keyCode);  
 bool currentlyPressed = isKeyPressed(keyCode);  
   
 if (currentlyPressed && !previouslyPressed) {  
 return KeyState::Pressed;  
 } else if (currentlyPressed && previouslyPressed) {  
 return KeyState::Held;  
 } else {  
 return KeyState::Released;  
 }  
 }  
   
 bool isKeyPressed(int keyCode) const {  
 auto it = currentKeyStates.find(keyCode);  
 return (it != currentKeyStates.end() && it->second);  
 }  
   
 bool wasKeyPressed(int keyCode) const {  
 auto it = previousKeyStates.find(keyCode);  
 return (it != previousKeyStates.end() && it->second);  
 }  
   
 // Mouse state  
 Vector2 getMousePosition() const { return mousePosition; }  
 Vector2 getMouseDelta() const { return mouseDelta; }  
   
 bool isMouseButtonPressed(int button) const {  
 return isKeyPressed(MOUSE\_BUTTON\_OFFSET + button);  
 }  
   
 // Gamepad state  
 bool isGamepadConnected(int gamepadIndex) const {  
 if (gamepadIndex < 0 || gamepadIndex >= static\_cast<int>(gamepads.size())) {  
 return false;  
 }  
 return gamepads[gamepadIndex].connected;  
 }  
   
 float getGamepadAxisValue(int gamepadIndex, int axisIndex) const {  
 if (gamepadIndex < 0 || gamepadIndex >= static\_cast<int>(gamepads.size()) ||  
 !gamepads[gamepadIndex].connected ||   
 axisIndex < 0 || axisIndex >= static\_cast<int>(gamepads[gamepadIndex].axes.size())) {  
 return 0.0f;  
 }  
 return gamepads[gamepadIndex].axes[axisIndex];  
 }  
   
 bool isGamepadButtonPressed(int gamepadIndex, int buttonIndex) const {  
 if (gamepadIndex < 0 || gamepadIndex >= static\_cast<int>(gamepads.size()) ||  
 !gamepads[gamepadIndex].connected ||   
 buttonIndex < 0 || buttonIndex >= static\_cast<int>(gamepads[gamepadIndex].buttons.size())) {  
 return false;  
 }  
 return gamepads[gamepadIndex].buttons[buttonIndex];  
 }  
   
private:  
 // Private constructor for singleton  
 InputSystem() {}  
   
 // Platform-specific input polling  
 void pollPlatformInput() {  
 // This would be implemented differently based on platform (Windows, macOS, etc.)  
 // and potentially using a library like GLFW, SDL, etc.  
   
 // For example, with GLFW:  
 // GLFWwindow\* window = GetMainWindow();  
 // currentKeyStates[GLFW\_KEY\_W] = glfwGetKey(window, GLFW\_KEY\_W) == GLFW\_PRESS;  
 // double mouseX, mouseY;  
 // glfwGetCursorPos(window, &mouseX, &mouseY);  
 // mousePosition = Vector2(static\_cast<float>(mouseX), static\_cast<float>(mouseY));  
 }  
   
 // Constants  
 static constexpr int MOUSE\_BUTTON\_OFFSET = 1000;  
};  
  
// Initialize the singleton instance pointer  
InputSystem\* InputSystem::instance = nullptr;

### 11.2.3 Memory Management in Game Engines

Memory management is critical in games for performance and stability:

// Custom memory allocator for game objects  
class GameObjectAllocator {  
private:  
 // Memory block for a pool of objects  
 struct MemoryBlock {  
 void\* memory;  
 size\_t blockSize;  
 size\_t used;  
 MemoryBlock\* next;  
 };  
   
 MemoryBlock\* firstBlock;  
 size\_t defaultBlockSize;  
   
 // Statistics  
 size\_t totalAllocated;  
 size\_t totalUsed;  
 int allocationCount;  
   
public:  
 GameObjectAllocator(size\_t blockSize = 1024 \* 1024) // 1MB default block size  
 : firstBlock(nullptr), defaultBlockSize(blockSize),  
 totalAllocated(0), totalUsed(0), allocationCount(0) {}  
   
 ~GameObjectAllocator() {  
 // Free all allocated memory  
 MemoryBlock\* current = firstBlock;  
 while (current) {  
 MemoryBlock\* next = current->next;  
 free(current->memory);  
 free(current);  
 current = next;  
 }  
 }  
   
 // Allocate memory from the pool  
 void\* allocate(size\_t size, size\_t alignment = 16) {  
 // Ensure minimum alignment  
 size = alignUp(size, alignment);  
   
 // Find a block with enough space  
 MemoryBlock\* block = findBlockWithSpace(size);  
 if (!block) {  
 // No suitable block found, create a new one  
 size\_t blockSize = std::max(size + sizeof(AllocationHeader), defaultBlockSize);  
 block = createMemoryBlock(blockSize);  
 }  
   
 // Calculate aligned address  
 uintptr\_t rawAddress = reinterpret\_cast<uintptr\_t>(block->memory) + block->used;  
 uintptr\_t misalignment = rawAddress & (alignment - 1);  
 size\_t padding = (misalignment > 0) ? (alignment - misalignment) : 0;  
   
 // Calculate final address  
 uintptr\_t alignedAddress = rawAddress + padding;  
   
 // Store allocation header just before the returned memory  
 AllocationHeader\* header = reinterpret\_cast<AllocationHeader\*>(alignedAddress - sizeof(AllocationHeader));  
 header->size = size;  
 header->padding = padding;  
   
 // Update block usage  
 block->used += size + padding;  
   
 // Update statistics  
 totalUsed += size + padding;  
 allocationCount++;  
   
 // Return aligned address  
 return reinterpret\_cast<void\*>(alignedAddress);  
 }  
   
 // Deallocate memory (not actually freeing memory in a pool allocator)  
 void deallocate(void\* ptr) {  
 if (!ptr) return;  
   
 // In a real game engine, you might implement a more sophisticated   
 // deallocation strategy or use different allocation schemes  
 // Here we're just tracking statistics  
 AllocationHeader\* header = reinterpret\_cast<AllocationHeader\*>(  
 reinterpret\_cast<uintptr\_t>(ptr) - sizeof(AllocationHeader)  
 );  
   
 totalUsed -= header->size + header->padding;  
 allocationCount--;  
 }  
   
 // Reset the allocator (free all objects at once)  
 void reset() {  
 MemoryBlock\* block = firstBlock;  
 while (block) {  
 block->used = 0;  
 block = block->next;  
 }  
   
 totalUsed = 0;  
 allocationCount = 0;  
 }  
   
 // Statistics  
 size\_t getTotalAllocated() const { return totalAllocated; }  
 size\_t getTotalUsed() const { return totalUsed; }  
 int getAllocationCount() const { return allocationCount; }  
   
private:  
 // Header stored before each allocation  
 struct AllocationHeader {  
 size\_t size;  
 size\_t padding;  
 };  
   
 // Helper method to align values  
 size\_t alignUp(size\_t value, size\_t alignment) {  
 return (value + alignment - 1) & ~(alignment - 1);  
 }  
   
 // Find a memory block with enough space  
 MemoryBlock\* findBlockWithSpace(size\_t size) {  
 MemoryBlock\* block = firstBlock;  
 while (block) {  
 size\_t availableSpace = block->blockSize - block->used;  
 if (availableSpace >= size + sizeof(AllocationHeader)) {  
 return block;  
 }  
 block = block->next;  
 }  
 return nullptr;  
 }  
   
 // Create a new memory block  
 MemoryBlock\* createMemoryBlock(size\_t blockSize) {  
 // Allocate block header  
 MemoryBlock\* block = static\_cast<MemoryBlock\*>(malloc(sizeof(MemoryBlock)));  
   
 // Allocate block memory  
 block->memory = malloc(blockSize);  
 block->blockSize = blockSize;  
 block->used = 0;  
   
 // Add to linked list of blocks  
 block->next = firstBlock;  
 firstBlock = block;  
   
 // Update statistics  
 totalAllocated += blockSize;  
   
 return block;  
 }  
};  
  
// Usage example:  
// GameObjectAllocator allocator;  
// Player\* player = new(allocator.allocate(sizeof(Player))) Player();  
// allocator.deallocate(player);

### 11.2.4 Data-Oriented Design

Modern game engines often use data-oriented design for better performance:

// Traditional object-oriented approach  
class GameObject {  
private:  
 Vector3 position;  
 Quaternion rotation;  
 Vector3 scale;  
 Mesh\* mesh;  
 Material\* material;  
 PhysicsBody\* physics;  
   
public:  
 void update(float deltaTime) {  
 // Update logic  
 if (physics) {  
 position = physics->getPosition();  
 rotation = physics->getRotation();  
 }  
 }  
   
 void render() {  
 // Render the object  
 if (mesh && material) {  
 Matrix4x4 transform = calculateTransform();  
 material->bind();  
 mesh->render(transform);  
 }  
 }  
};  
  
// Data-oriented approach using Structure of Arrays (SoA)  
class GameObjectSystem {  
private:  
 // Components stored in parallel arrays for better cache coherency  
 struct {  
 std::vector<Vector3> positions;  
 std::vector<Quaternion> rotations;  
 std::vector<Vector3> scales;  
 } transforms;  
   
 struct {  
 std::vector<Mesh\*> meshes;  
 std::vector<Material\*> materials;  
 } renderers;  
   
 struct {  
 std::vector<PhysicsBody\*> bodies;  
 std::vector<bool> hasPhysics; // Flag for whether an entity has physics  
 } physics;  
   
 std::vector<bool> active; // Active status for each entity  
 size\_t capacity = 0;  
 size\_t count = 0;  
   
public:  
 GameObjectSystem(size\_t initialCapacity = 1000) {  
 // Reserve space in all arrays  
 transforms.positions.reserve(initialCapacity);  
 transforms.rotations.reserve(initialCapacity);  
 transforms.scales.reserve(initialCapacity);  
   
 renderers.meshes.reserve(initialCapacity);  
 renderers.materials.reserve(initialCapacity);  
   
 physics.bodies.reserve(initialCapacity);  
 physics.hasPhysics.reserve(initialCapacity);  
   
 active.reserve(initialCapacity);  
   
 capacity = initialCapacity;  
 }  
   
 // Create a new entity  
 size\_t createEntity() {  
 size\_t entityId = count++;  
   
 // Ensure capacity  
 if (entityId >= capacity) {  
 enlargeCapacity();  
 }  
   
 // Initialize component data  
 transforms.positions.push\_back(Vector3(0, 0, 0));  
 transforms.rotations.push\_back(Quaternion::identity());  
 transforms.scales.push\_back(Vector3(1, 1, 1));  
   
 renderers.meshes.push\_back(nullptr);  
 renderers.materials.push\_back(nullptr);  
   
 physics.bodies.push\_back(nullptr);  
 physics.hasPhysics.push\_back(false);  
   
 active.push\_back(true);  
   
 return entityId;  
 }  
   
 // Update physics and transform data  
 void updatePhysics(float deltaTime) {  
 // Process all physics bodies in one batch for better cache utilization  
 for (size\_t i = 0; i < count; ++i) {  
 if (!active[i] || !physics.hasPhysics[i]) continue;  
   
 PhysicsBody\* body = physics.bodies[i];  
 if (body) {  
 // Update transform from physics simulation  
 transforms.positions[i] = body->getPosition();  
 transforms.rotations[i] = body->getRotation();  
 }  
 }  
 }  
   
 // Render all entities efficiently  
 void render(RenderContext& context) {  
 // First sort by material to minimize state changes  
 std::vector<size\_t> renderOrder;  
 renderOrder.reserve(count);  
   
 for (size\_t i = 0; i < count; ++i) {  
 if (active[i] && renderers.meshes[i] && renderers.materials[i]) {  
 renderOrder.push\_back(i);  
 }  
 }  
   
 std::sort(renderOrder.begin(), renderOrder.end(),   
 [this](size\_t a, size\_t b) {  
 return renderers.materials[a] < renderers.materials[b];  
 });  
   
 // Render sorted entities  
 Material\* currentMaterial = nullptr;  
   
 for (size\_t entityIdx : renderOrder) {  
 // Set material only when it changes  
 if (renderers.materials[entityIdx] != currentMaterial) {  
 if (currentMaterial) {  
 currentMaterial->unbind();  
 }  
   
 currentMaterial = renderers.materials[entityIdx];  
 currentMaterial->bind();  
 }  
   
 // Calculate transform matrix  
 Matrix4x4 transform = Matrix4x4::createTransformMatrix(  
 transforms.positions[entityIdx],  
 transforms.rotations[entityIdx],  
 transforms.scales[entityIdx]  
 );  
   
 // Set transform uniform  
 context.setModelMatrix(transform);  
   
 // Draw mesh  
 renderers.meshes[entityIdx]->draw();  
 }  
   
 // Unbind last material  
 if (currentMaterial) {  
 currentMaterial->unbind();  
 }  
 }  
   
 // Setters for entity components  
 void setTransform(size\_t entityId, const Vector3& position,   
 const Quaternion& rotation, const Vector3& scale) {  
 if (entityId >= count) return;  
   
 transforms.positions[entityId] = position;  
 transforms.rotations[entityId] = rotation;  
 transforms.scales[entityId] = scale;  
 }  
   
 void setRenderer(size\_t entityId, Mesh\* mesh, Material\* material) {  
 if (entityId >= count) return;  
   
 renderers.meshes[entityId] = mesh;  
 renderers.materials[entityId] = material;  
 }  
   
 void setPhysics(size\_t entityId, PhysicsBody\* body) {  
 if (entityId >= count) return;  
   
 physics.bodies[entityId] = body;  
 physics.hasPhysics[entityId] = (body != nullptr);  
 }  
   
private:  
 void enlargeCapacity() {  
 size\_t newCapacity = capacity \* 2;  
   
 transforms.positions.reserve(newCapacity);  
 transforms.rotations.reserve(newCapacity);  
 transforms.scales.reserve(newCapacity);  
   
 renderers.meshes.reserve(newCapacity);  
 renderers.materials.reserve(newCapacity);  
   
 physics.bodies.reserve(newCapacity);  
 physics.hasPhysics.reserve(newCapacity);  
   
 active.reserve(newCapacity);  
   
 capacity = newCapacity;  
 }  
};

### 11.2.5 Game Loop and Time Management

The game loop is the backbone of any game engine:

class GameEngine {  
private:  
 bool running = false;  
 double targetFrameRate = 60.0;  
 double targetFrameTime = 1.0 / 60.0; // In seconds  
   
 // Game state  
 Scene\* activeScene = nullptr;  
   
 // Core systems  
 RenderingEngine\* renderingEngine;  
 PhysicsEngine\* physicsEngine;  
 AudioEngine\* audioEngine;  
 InputSystem\* inputSystem;  
   
 // Timing  
 double lastFrameTime;  
 double deltaTime;  
   
 // Performance metrics  
 double fpsUpdateTime = 0.0;  
 int frameCount = 0;  
 double currentFps = 0.0;  
   
public:  
 GameEngine() {  
 // Create engine systems  
 renderingEngine = new RenderingEngine();  
 physicsEngine = new PhysicsEngine();  
 audioEngine = new AudioEngine();  
 inputSystem = &InputSystem::getInstance();  
 }  
   
 ~GameEngine() {  
 delete renderingEngine;  
 delete physicsEngine;  
 delete audioEngine;  
 }  
   
 void initialize() {  
 // Initialize all engine systems  
 renderingEngine->initialize();  
 physicsEngine->initialize();  
 audioEngine->initialize();  
   
 lastFrameTime = getCurrentTime();  
 running = true;  
 }  
   
 void run() {  
 initialize();  
   
 // Main game loop  
 while (running) {  
 // Process one frame  
 tick();  
 }  
   
 shutdown();  
 }  
   
 void setScene(Scene\* scene) {  
 activeScene = scene;  
 }  
   
 void setTargetFrameRate(double fps) {  
 targetFrameRate = fps;  
 targetFrameTime = 1.0 / fps;  
 }  
   
 double getFPS() const {  
 return currentFps;  
 }  
   
private:  
 void tick() {  
 // Calculate delta time  
 double currentTime = getCurrentTime();  
 deltaTime = currentTime - lastFrameTime;  
 lastFrameTime = currentTime;  
   
 // Cap delta time to prevent "spiral of death" on slow frames  
 if (deltaTime > 0.25) {  
 deltaTime = 0.25;  
 }  
   
 // Update FPS counter  
 frameCount++;  
 fpsUpdateTime += deltaTime;  
   
 if (fpsUpdateTime >= 1.0) {  
 currentFps = frameCount / fpsUpdateTime;  
 frameCount = 0;  
 fpsUpdateTime = 0.0;  
 }  
   
 // Process input  
 inputSystem->update();  
   
 // Check if we should quit  
 if (inputSystem->isKeyPressed(KeyCode::Escape)) {  
 running = false;  
 return;  
 }  
   
 // Update game logic and physics  
 update(deltaTime);  
   
 // Render the scene  
 render();  
   
 // Limit frame rate if needed  
 double frameEndTime = getCurrentTime();  
 double frameTime = frameEndTime - currentTime;  
 double sleepTime = targetFrameTime - frameTime;  
   
 if (sleepTime > 0) {  
 sleep(sleepTime);  
 }  
 }  
   
 void update(double dt) {  
 // Skip if no active scene  
 if (!activeScene) return;  
   
 // Update physics (may use fixed timestep internally)  
 physicsEngine->update(dt);  
   
 // Update scene entities  
 activeScene->update(dt);  
   
 // Update audio  
 audioEngine->update(dt);  
 }  
   
 void render() {  
 // Skip if no active scene  
 if (!activeScene) return;  
   
 // Set camera  
 Camera\* mainCamera = activeScene->getMainCamera();  
 if (mainCamera) {  
 renderingEngine->setActiveCamera(mainCamera);  
 }  
   
 // Clear previous frame  
 renderingEngine->clear();  
   
 // Render the scene  
 activeScene->render(\*renderingEngine);  
   
 // Finish rendering and display the frame  
 renderingEngine->present();  
 }  
   
 void shutdown() {  
 // Cleanup resources  
 if (activeScene) {  
 delete activeScene;  
 }  
 }  
   
 double getCurrentTime() {  
 // Platform-specific time function  
 // For example, using std::chrono:  
 auto now = std::chrono::high\_resolution\_clock::now();  
 auto duration = now.time\_since\_epoch();  
 auto seconds = std::chrono::duration\_cast<std::chrono::duration<double>>(duration);  
 return seconds.count();  
 }  
   
 void sleep(double seconds) {  
 // Platform-specific sleep function  
 // For example, using std::this\_thread::sleep\_for:  
 std::this\_thread::sleep\_for(std::chrono::duration<double>(seconds));  
 }  
};

### 11.2.6 Popular Game Engines Using C++

#### 1. Unreal Engine

Epic Games’ Unreal Engine is one of the most popular commercial game engines, using C++ as its primary language with a visual scripting system called Blueprints built on top.

**Key Features**: - Powerful rendering engine with real-time global illumination - Visual scripting via Blueprints - Advanced physics and animation systems - Robust networking capabilities - Built-in VR/AR support - Extensive marketplace for assets and plugins

**Example of Unreal Engine C++ class**:

// Example PlayerCharacter class in Unreal Engine  
#include "PlayerCharacter.h"  
#include "Camera/CameraComponent.h"  
#include "Components/InputComponent.h"  
#include "GameFramework/CharacterMovementComponent.h"  
  
APlayerCharacter::APlayerCharacter()  
{  
 // Create a camera boom  
 CameraBoom = CreateDefaultSubobject<USpringArmComponent>(TEXT("CameraBoom"));  
 CameraBoom->SetupAttachment(RootComponent);  
 CameraBoom->TargetArmLength = 300.0f;  
 CameraBoom->bUsePawnControlRotation = true;  
   
 // Create a follow camera  
 FollowCamera = CreateDefaultSubobject<UCameraComponent>(TEXT("FollowCamera"));  
 FollowCamera->SetupAttachment(CameraBoom, USpringArmComponent::SocketName);  
 FollowCamera->bUsePawnControlRotation = false;  
   
 // Set character movement properties  
 GetCharacterMovement()->bOrientRotationToMovement = true;  
 GetCharacterMovement()->RotationRate = FRotator(0.0f, 540.0f, 0.0f);  
 GetCharacterMovement()->JumpZVelocity = 600.f;  
 GetCharacterMovement()->AirControl = 0.2f;  
   
 // Don't rotate when the controller rotates  
 bUseControllerRotationPitch = false;  
 bUseControllerRotationYaw = false;  
 bUseControllerRotationRoll = false;  
}  
  
void APlayerCharacter::SetupPlayerInputComponent(UInputComponent\* PlayerInputComponent)  
{  
 // Set up gameplay key bindings  
 check(PlayerInputComponent);  
   
 PlayerInputComponent->BindAction("Jump", IE\_Pressed, this, &ACharacter::Jump);  
 PlayerInputComponent->BindAction("Jump", IE\_Released, this, &ACharacter::StopJumping);  
   
 PlayerInputComponent->BindAxis("MoveForward", this, &APlayerCharacter::MoveForward);  
 PlayerInputComponent->BindAxis("MoveRight", this, &APlayerCharacter::MoveRight);  
   
 PlayerInputComponent->BindAxis("Turn", this, &APawn::AddControllerYawInput);  
 PlayerInputComponent->BindAxis("LookUp", this, &APawn::AddControllerPitchInput);  
}  
  
void APlayerCharacter::MoveForward(float Value)  
{  
 if ((Controller != nullptr) && (Value != 0.0f))  
 {  
 // Find out which way is forward  
 const FRotator Rotation = Controller->GetControlRotation();  
 const FRotator YawRotation(0, Rotation.Yaw, 0);  
   
 // Get forward vector  
 const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::X);  
 AddMovementInput(Direction, Value);  
 }  
}  
  
void APlayerCharacter::MoveRight(float Value)  
{  
 if ((Controller != nullptr) && (Value != 0.0f))  
 {  
 // Find out which way is right  
 const FRotator Rotation = Controller->GetControlRotation();  
 const FRotator YawRotation(0, Rotation.Yaw, 0);  
   
 // Get right vector   
 const FVector Direction = FRotationMatrix(YawRotation).GetUnitAxis(EAxis::Y);  
   
 // Add movement in that direction  
 AddMovementInput(Direction, Value);  
 }  
}

#### 2. Unity

While Unity primarily uses C# for game scripting, its core engine is written in C++. Unity also allows developers to write native C++ plugins for performance-critical code.

// Example C++ plugin for Unity  
#include "Unity/IUnityInterface.h"  
#include "PhysicsSimulation.h"  
  
static PhysicsSimulation\* g\_PhysicsSim = nullptr;  
  
// Unity plugin load  
extern "C" void UNITY\_INTERFACE\_EXPORT UNITY\_INTERFACE\_API   
UnityPluginLoad(IUnityInterfaces\* unityInterfaces)  
{  
 // Create physics simulation  
 g\_PhysicsSim = new PhysicsSimulation();  
 g\_PhysicsSim->Initialize();  
}  
  
// Unity plugin unload  
extern "C" void UNITY\_INTERFACE\_EXPORT UNITY\_INTERFACE\_API   
UnityPluginUnload()  
{  
 if (g\_PhysicsSim)  
 {  
 g\_PhysicsSim->Shutdown();  
 delete g\_PhysicsSim;  
 g\_PhysicsSim = nullptr;  
 }  
}  
  
// Function called from Unity to update physics  
extern "C" void UNITY\_INTERFACE\_EXPORT UNITY\_INTERFACE\_API   
UpdatePhysicsSimulation(float deltaTime)  
{  
 if (g\_PhysicsSim)  
 {  
 g\_PhysicsSim->Update(deltaTime);  
 }  
}  
  
// Function to get particle positions  
extern "C" void UNITY\_INTERFACE\_EXPORT UNITY\_INTERFACE\_API   
GetParticlePositions(int\* count, float\* positions)  
{  
 if (g\_PhysicsSim)  
 {  
 g\_PhysicsSim->GetParticleData(count, positions);  
 }  
}

#### 3. CryEngine

CryEngine is another C++ based engine, known for its graphical fidelity:

// Example CryEngine entity class  
#include <CryEntitySystem/IEntityComponent.h>  
#include <CryPhysics/IPhysics.h>  
  
class CPlayerComponent : public IEntityComponent  
{  
public:  
 // IEntityComponent  
 virtual void Initialize() override  
 {  
 m\_pEntity->LoadGeometry(0, "Objects/Characters/Player/player.cgf");  
   
 // Add a physics component  
 SEntityPhysicalizeParams physParams;  
 physParams.type = PE\_LIVING;  
 physParams.mass = 90.0f;  
 m\_pEntity->Physicalize(physParams);  
   
 // Listen for input events  
 m\_pInput = gEnv->pInput;  
 m\_pInput->AddEventListener(this);  
 }  
   
 virtual void ProcessEvent(SEntityEvent& event) override  
 {  
 switch(event.event)  
 {  
 case ENTITY\_EVENT\_UPDATE:  
 {  
 const float deltaTime = event.fParam[0];  
 UpdateMovement(deltaTime);  
 break;  
 }  
 }  
 }  
   
 virtual void OnInputEvent(const SInputEvent& event) override  
 {  
 // Handle input events  
 if (event.keyId == eKI\_W && event.state == eIS\_Pressed)  
 {  
 m\_isMovingForward = true;  
 }  
 else if (event.keyId == eKI\_W && event.state == eIS\_Released)  
 {  
 m\_isMovingForward = false;  
 }  
   
 // Handle other input...  
 }  
   
private:  
 void UpdateMovement(float deltaTime)  
 {  
 if (!m\_isMovingForward)  
 return;  
   
 // Get current position & rotation  
 Matrix34 transform = m\_pEntity->GetWorldTM();  
 Vec3 position = transform.GetTranslation();  
   
 // Move forward  
 Vec3 forward = transform.GetColumn1().normalized();  
 position += forward \* m\_moveSpeed \* deltaTime;  
   
 // Update entity position  
 transform.SetTranslation(position);  
 m\_pEntity->SetWorldTM(transform);  
 }  
   
 IInput\* m\_pInput = nullptr;  
 bool m\_isMovingForward = false;  
 float m\_moveSpeed = 5.0f;  
};

### 11.2.7 Challenges in C++ Game Engine Development

Despite its strengths, C++ presents several challenges in game engine development:

1. **Compilation Times**: Large C++ projects can have lengthy build times
   * Solution: Precompiled headers, unity builds, modules (C++20)
2. **Memory Management**: Manual memory management can lead to leaks and crashes
   * Solution: Smart pointers, custom allocators, RAII pattern
3. **Platform Differences**: Supporting multiple platforms requires handling differences
   * Solution: Platform abstraction layers, conditional compilation
4. **Multithreading Complexity**: C++ multithreading requires careful synchronization
   * Solution: Task-based parallelism, thread pools, lock-free data structures
5. **Legacy Code**: Many game engines contain old code that’s hard to modernize
   * Solution: Incremental modernization, clear interfaces, comprehensive testing

### 11.2.8 Modern C++ Features in Game Engines

Modern C++ (C++11 and beyond) introduced many features beneficial for game development:

1. **Move Semantics**: Efficient transfer of resources without expensive copies

* // Move constructor example  
  class Mesh {  
  private:  
   std::vector<Vertex> vertices;  
   std::vector<unsigned int> indices;  
   unsigned int vao, vbo, ebo; // OpenGL handles  
    
  public:  
   // Move constructor  
   Mesh(Mesh&& other) noexcept  
   : vertices(std::move(other.vertices)),  
   indices(std::move(other.indices)),  
   vao(other.vao), vbo(other.vbo), ebo(other.ebo) {  
   // Invalidate other's OpenGL handles  
   other.vao = other.vbo = other.ebo = 0;  
   }  
    
   // Move assignment  
   Mesh& operator=(Mesh&& other) noexcept {  
   if (this != &other) {  
   // Clean up current resources  
   if (vao != 0) glDeleteVertexArrays(1, &vao);  
   if (vbo != 0) glDeleteBuffers(1, &vbo);  
   if (ebo != 0) glDeleteBuffers(1, &ebo);  
    
   // Move data from other  
   vertices = std::move(other.vertices);  
   indices = std::move(other.indices);  
   vao = other.vao;  
   vbo = other.vbo;  
   ebo = other.ebo;  
    
   // Invalidate other's OpenGL handles  
   other.vao = other.vbo = other.ebo = 0;  
   }  
   return \*this;  
   }  
  };

1. **Lambda Expressions**: Convenient for callbacks and short algorithms

* // Using lambdas for event callbacks  
  inputSystem.registerKeyCallback(KeyCode::Space, [this](KeyState state) {  
   if (state == KeyState::Pressed) {  
   player->jump();  
   }  
  });  
    
  // Using lambdas for custom sorting  
  std::sort(renderables.begin(), renderables.end(),   
   [](const Renderable& a, const Renderable& b) {  
   // Sort by material first, then by mesh  
   if (a.material->getID() != b.material->getID())  
   return a.material->getID() < b.material->getID();  
   return a.mesh->getID() < b.mesh->getID();  
   });

1. **Smart Pointers**: Safer memory management

* // Using unique\_ptr for exclusive ownership  
  std::unique\_ptr<Texture> loadTexture(const std::string& filename) {  
   auto texture = std::make\_unique<Texture>();  
   if (!texture->load(filename)) {  
   return nullptr;  
   }  
   return texture;  
  }  
    
  // Using shared\_ptr for shared resources  
  class Material {  
  private:  
   std::shared\_ptr<Shader> shader;  
   std::shared\_ptr<Texture> diffuseMap;  
   std::shared\_ptr<Texture> normalMap;  
    
  public:  
   void setShader(std::shared\_ptr<Shader> s) { shader = s; }  
   void setDiffuseMap(std::shared\_ptr<Texture> t) { diffuseMap = t; }  
   void setNormalMap(std::shared\_ptr<Texture> t) { normalMap = t; }  
  };

1. **constexpr**: Compile-time computation

* // Compile-time hash function for string literals  
  constexpr uint32\_t hashString(const char\* str, size\_t length) {  
   uint32\_t hash = 5381;  
   for (size\_t i = 0; i < length; ++i) {  
   hash = ((hash << 5) + hash) + static\_cast<unsigned char>(str[i]);  
   }  
   return hash;  
  }  
    
  // Usage  
  constexpr uint32\_t textureId = hashString("diffuse\_map", 11);

### 11.2.9 Future Trends in C++ Game Engines

Several trends are shaping the future of C++ game engines:

1. **Data-Oriented Design**: Focusing on data layout and processing for better cache utilization
2. **Parallelism and Concurrency**: Greater emphasis on multi-core utilization
3. **Modular Design**: More modular engines with plug-and-play components
4. **Real-Time Ray Tracing**: Adoption of hardware-accelerated ray tracing technology
5. **Machine Learning Integration**: Using ML for animation, procedural generation, and AI
6. **Cloud and Distributed Computing**: Offloading complex computations to the cloud
7. **WebAssembly Support**: Compiling C++ to run in browsers via WebAssembly

As hardware continues to evolve and new C++ standards emerge, game engines will continue to leverage the language’s unique combination of high performance and expressive abstractions to create ever more immersive and complex gaming experiences.