The Complete Guide to

C++ Pointers

Includes C++23 & C++26

Prepared by

Ayman Alheraki

DRAFT

The Complete Guide to C++ Pointers

Some drafting assistance and idea exploration were supported by modern AI tools, with full supervision, verification, correction, and authorship.

Prepared by Ayman Alheraki

Target Audience: All levels

simplifycpp.org

April 2026

Contents

Author's Introduction

The **C++** programming language continues to evolve, solidifying its position as one of the most powerful and flexible languages in existence. It masterfully bridges the gap between low-level memory manipulation and high-level abstraction. Among its core concepts, **pointers** stand out as a fundamental element that grants programmers unparalleled control over system resources and memory, a necessity for building high-performance applications. With the advent of **C++23** and the forthcoming **C++26**, the language refines how we manage memory, making pointer usage safer and more expressive than ever before.

This book, **"The Complete Guide to C++ Pointers"**, is designed to be your definitive companion for understanding and mastering pointers. It takes you on a journey from the foundational principles to the most advanced techniques, all while integrating the latest features of modern C++. Whether you are a student taking your first steps in programming or a seasoned professional seeking to deepen your expertise, this guide will serve as an invaluable resource.

Why This Book?

Throughout years of teaching and professional development, I've observed that pointers are often the most intimidating concept for programmers to grasp. Yet, they are the key that unlocks a profound understanding of program behavior, performance optimization, and system-level programming. This book demystifies pointers by presenting them in a logical, step-by-step manner. It begins with the classic, bare-metal concepts of memory addresses and evolves into the sophisticated, safety-oriented features introduced by modern C++ standards, including C++23 and the draft proposals for C++26.

What's New in This Edition?

This guide is meticulously updated to reflect the state-of-the-art in C++:

- **C++23 Features:** We explore new standard library additions like `std::out_ptr`, `std::inout_ptr`, and improvements to `std::unique_ptr` that make smart pointer

interoperability with legacy C APIs seamless and safer.

- **C++26 Draft Insights:** We provide a forward-looking perspective on proposed features that will further revolutionize pointer usage, such as `std::observer_ptr` for clear non-owning semantics, `std::ptr_len` for safer pointer-length pairs, and potential language enhancements for compile-time memory management.

- **Simplified Explanations:** Complex topics like pointer-to-member, function pointers, and allocator-aware smart pointers are broken down into easy-to-understand concepts with practical, real-world examples.

Structure of the Book

The book is organized into nine comprehensive chapters, each building upon the last to ensure a smooth learning curve:

1. **Chapter 1: Introduction to Pointers in C++**
   We begin by defining pointers, exploring their importance, distinguishing them from references, and understanding the fundamentals of memory management in modern C++.

2. **Chapter 2: Basics of Pointers**
   Learn how to declare and initialize pointers, master pointer arithmetic, and understand the intrinsic relationship between pointers and arrays.

3. **Chapter 3: Advanced Pointers**
   Dive into using pointers with functions, structures, and classes, and gain a deep understanding of dynamic memory management with `new` and `delete`, and their limitations in modern C++.

4. **Chapter 4: Modern C++ and Smart Pointers**
   Explore the full spectrum of smart pointers: `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`, and the new vocabulary types introduced in C++23 and proposed for C++26. Learn how to write exception-safe, memory-leak-free code.

5. **Chapter 5: Pointers and Object-Oriented Programming (OOP)**
   Discover how pointers enable polymorphism, manage object lifetimes in complex hierarchies, and facilitate design patterns like Factory and Observer.

6. **Chapter 6: Pointers and Data Structures**
   Apply pointer knowledge to build and manipulate fundamental data structures, including singly and doubly linked lists, binary trees, and graphs, with a focus on modern ownership semantics.

7. **Chapter 7: Pointers and Advanced C++ Features**
   Learn to integrate pointers with templates (including `std::shared_ptr<T>` in type-erased contexts), concurrency (atomic smart pointers, thread-safe weak pointers), and move semantics for optimal resource management.

8. **Chapter 8: Best Practices and Debugging**
   This chapter consolidates best practices to avoid common pitfalls like dangling pointers, double deletions, and memory fragmentation. It also covers using modern debugging tools like AddressSanitizer, Valgrind, and static analyzers to detect pointer-related issues.

9. **Appendices**
   Appendix A: Key Terminology
   Appendix B: A History of Pointers: From C to C++26
   Appendix C: Memory Management in Depth
   Appendix D: Common Pointer Errors and Debugging Techniques
   Appendix E: Practical Exercises with Solutions
   Appendix F: Additional Resources and Further Reading

How to Use This Book

This book is crafted to serve both as a structured educational tool and a comprehensive reference. If you are new to pointers, reading the chapters in order will provide you with a solid, foundational understanding. If you already have some experience, feel free to jump directly to the chapters on modern smart pointers or advanced techniques. Each chapter is enriched with practical, compilable code examples and concludes with exercises designed to reinforce the concepts covered.

A Final Word

My goal is to equip you with the knowledge and confidence to wield pointers—one of C++'s most powerful features—effectively and safely. Remember that mastery comes with practice. Experiment with the examples, modify them, and tackle the exercises. Embrace the power of modern C++ to write code that is not only performant but also robust and maintainable.

Stay Connected

For ongoing discussions, insights, and updates on **The Complete Guide to C++ Pointers** and the evolving standards, I invite you to connect with me on **LinkedIn**:
https://linkedin.com/in/aymanalheraki
You can also explore more resources and articles on my personal website:
https://simplifycpp.org

Ayman Alheraki

Preface

The Journey Ahead

Welcome to *The Complete Guide to C++ Pointers*. If you are holding this book—whether in digital or physical form—you are about to embark on a journey into one of the most powerful, yet often misunderstood, facets of the C++ programming language.

Pointers are the cornerstone of C++. They are the mechanism through which the language exposes the raw power of the underlying hardware, giving you, the programmer, direct control over memory, resources, and performance. However, with great power comes great responsibility. The history of C++ is, in many ways, a history of the evolution of memory management: from raw pointers, through the pitfalls of manual memory handling, to the sophisticated, safety-oriented abstractions of modern C++.

This book is born from a simple observation: most resources on pointers are either too basic, treating them as a mere syntax hurdle, or too archaic, focusing on C-style patterns that lead to unsafe and unmaintainable code. The advent of C++11, and the subsequent improvements in C++14, C++17, C++20, and now **C++23** and **C++26**, has fundamentally changed the landscape. The question is no longer *"how do I use pointers?"* but rather *"how do I use the right kind of pointer for the right job, safely and efficiently?"*

Philosophy of This Book

This guide is built on three core principles:

1. **Modern First, But Not Modern-Only:**

You cannot understand `std::unique_ptr` without first understanding what it replaces. You cannot appreciate the safety of `std::shared_ptr` without feeling the pain of manual reference counting. Therefore, we begin with the fundamentals—the memory model, raw pointers, and dynamic allocation—but we frame them as the foundation upon which modern constructs are built. Every raw pointer concept is followed by its modern counterpart, showing you not just *how* it works, but *why* we evolved beyond it.

2. **Standards-Aware Learning:**
   C++ is a living language. The ISO committee (WG21) continuously refines and expands the language. This book integrates features from **C++23** (such as `std::out_ptr`, `std::inout_ptr`, and stacktrace library improvements) and discusses proposed features for **C++26** (like `std::observer_ptr` and `std::ptr_len`) to ensure that your knowledge remains current for years to come. We will not only teach you what exists today but also prepare you for what is coming tomorrow.

3. **Practice Over Theory:**
   Every concept in this book is accompanied by complete, compilable code examples. You are encouraged to type them, run them, modify them, and break them. The appendices contain a wealth of practical exercises designed to reinforce each chapter's material. Programming is a craft, and a craft is honed through deliberate practice.

Who Is This Book For?

This book is designed for a wide spectrum of readers:

- **Beginners** who have some familiarity with C++ syntax (variables, loops, functions) but find pointers intimidating. Chapters 1–4 will provide a solid foundation.
- **Intermediate Developers** who use pointers occasionally but want to deepen their understanding of modern smart pointers, move semantics, and safe resource management. Chapters 4–6 will be particularly valuable.
- **Advanced Professionals** who need to master pointer-related techniques for high-performance computing, embedded systems, game development, or systems programming. Chapters 7–8 and the appendices delve into concurrency, template metaprogramming with pointers, and advanced debugging methodologies.

- **Educators and Trainers** seeking a comprehensive, up-to-date resource for teaching pointer concepts in C++ courses.

A Note on C++23 and C++26

The C++ standardization process is dynamic. At the time of writing, **C++23** is formally published (ISO/IEC 14882:2024), and the technical specifications for **C++26** are actively being drafted. This book incorporates:

- **For C++23:** We cover `std::out_ptr` and `std::inout_ptr` for seamless smart pointer interoperability with C APIs, `std::stacktrace` for debugging pointer-related crashes, and improvements to `std::unique_ptr` for array types.
- **For C++26 (Draft):** We provide forward-looking discussions on `std::observer_ptr` (a non-owning pointer vocabulary type), `std::ptr_len` for safe pointer-length pairs, and potential language features like `static_operator[]` that may influence pointer usage.

All code examples are annotated with the minimum standard version required (e.g., *// Requires C++23*), ensuring that you can experiment appropriately with your compiler.

How to Read This Book

If you are new to pointers, I recommend reading the chapters sequentially. Each chapter builds on the concepts established in previous ones.
If you have prior experience, feel free to use the book as a reference. Chapter 4 (Modern C++ and Smart Pointers) and Chapter 8 (Best Practices and Debugging) are essential reading for anyone working with pointers in production code.
Throughout the book, you will find:

- **Code Blocks:** Ready-to-compile examples that illustrate concepts.
- **Warning Boxes:** Highlighting common pitfalls and undefined behavior.
- **Modern C++ Notes:** Sidebars that contrast classic pointer usage with modern alternatives.
- **Exercises:** At the end of each chapter (and extensively in Appendix E) to test your understanding.

A Personal Note

I began my journey with C++ over three decades ago. I remember the frustration of chasing memory leaks, the dread of dangling pointers, and the confusion of pointer-to-pointer syntax. But I also remember the exhilaration of finally understanding how memory works, the elegance of a well-designed smart pointer hierarchy, and the satisfaction of writing code that is both fast and safe. This book is the culmination of those experiences—the mistakes I made, the lessons I learned, and the insights I've gained from teaching thousands of programmers. My goal is to shorten your learning curve, to guide you away from common errors, and to empower you with the knowledge to use C++ pointers with confidence and precision.

Acknowledgments

I would like to thank the C++ standards committee members, past and present, whose tireless work continues to shape this remarkable language. I am grateful to the open-source community for providing compilers (GCC, Clang, MSVC) and tools that make modern C++ development accessible to all. Special thanks to my readers and students whose questions and feedback have sharpened the clarity of this text.

Now, let us begin.

"The only way to learn a new programming language is by writing programs in it."
— Dennis Ritchie

Chapter 1

Introduction to Pointers in C++

1.1 What Are Pointers in C++?

Pointers are one of the most powerful and fundamental features of C++. They provide a level of control and flexibility that is unmatched by many higher-level programming languages. In this section, we will explore the definition of pointers, their importance in modern C++, the differences between pointers and references, and common misconceptions about pointers. We will also incorporate modern C++ concepts, including C++23 enhancements and a preview of C++26 features, to provide a comprehensive understanding of pointers in contemporary C++ programming.

1.1.1 Definition of Pointers

In C++, a **pointer** is a variable that stores the **memory address** of another variable or function. Unlike regular variables that hold data values directly, pointers hold the location in memory where the actual data resides. This allows programmers to indirectly access and manipulate data, forming the foundation of low-level programming and memory management in C++.
To declare a pointer, you specify the data type it points to, followed by an asterisk ($\star$), and then the pointer's name. For example:

```
int* ptr;          // Pointer to an integer
double* dptr;      // Pointer to a double
void* vptr;        // Pointer to an unknown type (void pointer)
```

Here, `ptr` is a pointer to an integer. The asterisk (`*`) indicates that `ptr` is a pointer, and `int` specifies that it points to an integer type. The pointer itself does not store an integer value but rather the address of an integer variable.

Important

Pointer Declaration Style: C++ allows multiple styles of pointer declaration: `int* ptr`, `int *ptr`, or `int * ptr`. The most common modern style is `int* ptr`, emphasizing that the type is "pointer to int." However, be cautious when declaring multiple pointers in one statement:

```
int* p1, p2;    // p1 is a pointer, p2 is an integer!
int *p1, *p2;   // Both are pointers
```

To initialize a pointer, you assign it the address of a variable using the **address-of operator (`&`)**. For example:

```
int x = 10;
int* ptr = &x;   // ptr now holds the memory address of x
```

To access the value stored at the memory address held by the pointer, you use the **dereference operator (`*`)**. For example:

```
int value = *ptr; // value will be 10
*ptr = 20;        // x now becomes 20 (modifying through pointer)
```

1.1.2 Importance of Pointers in Modern C++

Pointers remain a critical feature of C++, even in modern C++ (C++11 through C++23 and beyond). While modern C++ introduces safer alternatives like smart pointers, raw pointers are still widely used in specific scenarios. Below are key reasons why pointers are important in contemporary C++ programming:

1. **Dynamic Memory Allocation**:

    - Pointers enable **dynamic memory management**, allowing programs to allocate and deallocate memory at runtime. This is essential when the size of data structures is not known at compile time.

- The `new` operator allocates memory, and `delete` deallocates it. However, in modern C++, smart pointers are preferred:

```cpp
// Raw pointer approach (not recommended for owning pointers)
int* rawArr = new int[10];   // Allocate
delete[] rawArr;             // Deallocate

// Modern C++ approach (C++11 and later)
auto smartArr = std::make_unique<int[]>(10);  // Automatically
↪  deallocated
```

2. **Efficient Data Manipulation**:

   - Pointers allow direct memory access, enabling efficient code. Passing large structures or arrays via pointers avoids copying overhead.
   - For example:

```cpp
void processData(const int* data, std::size_t size) {
    for (std::size_t i = 0; i < size; ++i) {
        std::cout << data[i] << ' ';
    }
}

std::vector<int> vec = {1, 2, 3, 4, 5};
processData(vec.data(), vec.size()); // Pass pointer to internal array
```

3. **Functionality and Flexibility**:

   - Pointers enable advanced programming techniques: **function pointers**, **polymorphism**, and complex data structures like **linked lists**, **trees**, and **graphs**.
   - Modern C++ combines pointers with smart pointers for safe resource management:

```cpp
struct Node {
    int data;
    std::unique_ptr<Node> next;  // Unique ownership of next node
};
```

4. **Interfacing with Hardware and System Calls**:

   - In systems programming and embedded systems, pointers interact with hardware via memory-mapped I/O and direct memory access.
   - **C++23** introduces `std::out_ptr` and `std::inout_ptr` to safely interface smart pointers with C APIs that expect raw pointers:

```cpp
// C++23: Safely pass smart pointer to C function expecting raw pointer
void legacy_c_api(int** out_ptr);  // C function that allocates memory

std::unique_ptr<int, decltype(&std::free)> ptr(nullptr, std::free);
legacy_c_api(std::out_ptr(ptr));   // Automatically manages the raw
↪  pointer
```

5. **String and Array Handling**:

   - In C++, arrays and pointers are closely related. An array name decays to a pointer to its first element in most contexts.
   - However, modern C++ provides safer alternatives like `std::string`, `std::vector`, and `std::span` (C++20):

```cpp
int arr[5] = {1, 2, 3, 4, 5};
int* ptr = arr;              // Array-to-pointer decay

std::span<int> span(arr);    // C++20: A safer, non-owning view
```

1.1.3 Differences Between Pointers and References

While both pointers and references provide indirect access to variables, they have significant differences in syntax, usage, and behavior. Understanding these differences is crucial for effective C++ programming.

1. **Syntax and Usage**:

   - **Pointers** use `*` and `&` for dereferencing and address-of operations:

```cpp
int x = 10;
int* ptr = &x;     // Address-of operator
int value = *ptr;  // Dereference operator
```

   - **References** use `&` in declaration and require no explicit dereferencing:

```cpp
int x = 10;
int& ref = x;      // Reference declaration
int value = ref;   // Direct access (no dereference needed)
```

2. **Nullability**:

   - **Pointers** can be `nullptr`, indicating they point to no valid object:

```cpp
int* ptr = nullptr;  // Null pointer (C++11 and later)
```

- **References** must always refer to a valid object and cannot be null. This makes them inherently safer.

3. **Reassignment**:

   - **Pointers** can be reassigned to point to different memory locations:

```
int x = 10, y = 20;
int* ptr = &x;    // ptr points to x
ptr = &y;         // ptr now points to y
```

   - **References** cannot be reassigned after initialization; they always refer to the original object:

```
int x = 10, y = 20;
int& ref = x;     // ref refers to x
ref = y;          // Assigns y's value to x; ref still refers to x!
```

4. **Memory Management**:

   - **Pointers** require explicit memory management for dynamically allocated objects.
   - **References** are automatically bound to valid objects and do not involve manual memory management.

5. **Use Cases**:

   - **Pointers** are preferred for dynamic allocation, optional parameters, polymorphism, and when reassignment is needed.
   - **References** are ideal for function parameters (to avoid copying), for defining aliases, and when a non-nullable, non-reassignable indirection is desired.

Modern C++ Insight

Modern Guidance: In C++ Core Guidelines, references are preferred over pointers for function parameters when the argument must refer to a valid object. Use pointers when null is a valid state or when reassignment is required.

1.1.4 Common Misconceptions About Pointers

1. **Pointers Are Complicated and Dangerous**:

- While pointers can be challenging, modern C++ provides tools (smart pointers, `std::optional`, `std::span`) that make pointer usage safer. Understanding memory management and following best practices mitigates risks.

2. **Pointers and Arrays Are the Same**:

   - Arrays and pointers are distinct. An array name decays to a pointer in many contexts, but `sizeof` behavior differs, and arrays cannot be reassigned:

```
int arr[5] = {1, 2, 3, 4, 5};
int* ptr = arr;           // Decay: ptr points to first element
// arr = ptr;             // Error: array name is not assignable
std::size_t arrSize = sizeof(arr);  // 5 * sizeof(int)
std::size_t ptrSize = sizeof(ptr);  // Size of pointer (e.g., 8 bytes)
```

3. **Pointers Always Cause Memory Leaks**:

   - Memory leaks occur when dynamically allocated memory is not deallocated. With careful management—or better yet, using smart pointers—leaks can be eliminated:

```
auto ptr = std::make_unique<int>(42);  // No leak possible
```

4. **References Are Just Syntactic Sugar for Pointers**:

   - While references are often implemented as pointers under the hood, they have distinct semantics: references cannot be null, cannot be reassigned, and do not require explicit dereferencing. These constraints enable compiler optimizations and safer code.

5. **Pointers Are Obsolete in Modern C++**:

   - Pointers remain essential for low-level programming, performance-critical code, and interfacing with C libraries. However, modern C++ encourages using smart pointers for ownership and raw pointers only for non-owning observation.

1.1.5 Modern C++ Concepts in Pointers

Modern C++ (C++11 through C++23 and beyond) introduces features and best practices that enhance pointer safety and usability. Below are key modern concepts:

1. **Smart Pointers**:

- Smart pointers automatically manage the lifetime of dynamically allocated memory, preventing leaks and dangling pointers.
- **`std::unique_ptr`**: Exclusive ownership; cannot be copied, only moved.

```
auto ptr = std::make_unique<int>(42);  // C++14: make_unique
// auto copy = ptr;                    // Error: no copy
auto moved = std::move(ptr);           // Transfer ownership
```

- **`std::shared_ptr`**: Shared ownership with reference counting.

```
auto ptr1 = std::make_shared<int>(42);
auto ptr2 = ptr1;                      // Shared ownership
// Object destroyed when last shared_ptr is destroyed
```

- **`std::weak_ptr`**: Non-owning observer for shared_ptr, breaking circular references.

```
std::weak_ptr<int> weak = ptr1;
if (auto locked = weak.lock()) {       // Get shared_ptr if object still
↪  exists
    // Use locked
}
```

2. **C++23 Smart Pointer Enhancements**:

- `std::out_ptr` and `std::inout_ptr` allow smart pointers to interface safely with C APIs:

```
// Requires C++23
void c_api_alloc(int** out);  // Allocates memory

std::unique_ptr<int, decltype(&std::free)> ptr(nullptr, std::free);
c_api_alloc(std::out_ptr(ptr));  // Automatically manages the allocated
↪  memory
```

- `std::unique_ptr` now supports array types with `std::make_unique_for_overwrite` for default-initialized arrays (performance optimization).

3. **Move Semantics**:

- Move semantics enable efficient transfer of resource ownership, particularly with smart pointers:

```
std::unique_ptr<int> createResource() {
    return std::make_unique<int>(42);  // Returns by value (move
    ↪  semantics)
```

```
}

auto resource = createResource();      // No copying, efficient transfer
```

4. **`nullptr`**:

   - `nullptr` (C++11 and later) is type-safe and avoids the pitfalls of `NULL` (which is `0`) in overloaded contexts:

```
void f(int);
void f(int*);
f(NULL);      // Calls f(int) { ambiguous and dangerous!
f(nullptr);   // Calls f(int*) { correct
```

5. **Range-based For Loops and `std::span`**:

   - Range-based for loops simplify iteration, reducing pointer usage:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
for (const auto& elem : vec) {
    std::cout << elem << ' ';
}
```

   - `std::span` (C++20) provides a safe, non-owning view of contiguous memory, replacing raw pointer + size patterns:

```
void process(std::span<int> data) {
    for (int& elem : data) {
        elem *= 2;
    }
}

std::vector<int> vec = {1, 2, 3, 4, 5};
process(vec);  // No raw pointers needed
```

6. **Preview: C++26 Pointer Vocabulary Types**:

   - The C++26 draft proposes `std::observer_ptr`, a non-owning pointer vocabulary type that clearly expresses intent:

```
// Proposed for C++26
void useResource(std::observer_ptr<Resource> obs) {
    if (obs) {  // Check if not null
        obs->doSomething();
    }
}
```

- `std::ptr_len` is proposed as a standard type for pointer-length pairs, replacing many ad-hoc patterns:

```
// Proposed for C++26
std::ptr_len<const int> data = {myArray, mySize};
process(data);  // Safe, clear semantics
```

1.1.6 Summary

In this chapter, we established the foundational concepts of pointers in C++:

- Pointers store memory addresses and enable indirect data access.
- They remain essential for dynamic memory management, performance, systems programming, and implementing complex data structures.
- References offer safer, more constrained alternatives for many use cases.
- Modern C++ introduces smart pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`) and vocabulary types that dramatically improve pointer safety and expressiveness.
- C++23 adds `out_ptr`/`inout_ptr` for C API interoperability, and C++26 is expected to bring `observer_ptr` and `ptr_len` for clearer non-owning pointer semantics.

Understanding these concepts prepares you for the deeper exploration of pointer arithmetic, pointer-to-function, polymorphism, and advanced memory management techniques in the chapters ahead.

Exercise 1. ***Practice Exercises:***

1. *Declare a pointer to an integer, initialize it with the address of a variable, and modify the variable through the pointer.*
2. *Create a function that takes a pointer to an integer and doubles its value. Test it with a local variable.*
3. *Write code demonstrating that references cannot be reassigned, while pointers can.*
4. *Create a `std::unique_ptr<int>` and transfer ownership to another `unique_ptr` using `std::move`.*
5. *Research `std::span` and rewrite a function that takes a raw pointer and size to use `std::span` instead.*

1.2 Why Use Pointers in C++?

Pointers are a cornerstone of C++ programming, offering unparalleled control over memory and data structures. While modern C++ introduces safer alternatives like smart pointers and standard library containers, raw pointers remain essential in many scenarios. In this section, we will explore the **benefits of using pointers** and their **common use cases** in modern C++ programming. We will also incorporate the latest C++23 features and preview C++26 concepts to provide a comprehensive understanding of why pointers are still relevant and powerful.

1.2.1 Benefits of Using Pointers

Pointers provide several advantages that make them indispensable in C++ programming. Below are the key benefits:

1. **Direct Memory Access**:
   - Pointers allow direct memory manipulation, enabling low-level control essential for systems programming, embedded systems, and performance-critical applications.
   - For example, pointers can access specific memory locations such as hardware registers:

```
// Memory-mapped I/O (embedded systems)
volatile uint32_t* controlRegister =
↪  reinterpret_cast<uint32_t*>(0x40021000);
*controlRegister = 0x01;  // Enable peripheral clock
```

   - **C++23** enhances safety with `std::bit_cast` for type-punning operations, reducing reliance on unsafe `reinterpret_cast` in certain scenarios.

2. **Dynamic Memory Management**:
   - Pointers enable runtime memory allocation, essential when data structure sizes are unknown at compile time.
   - Traditional approach:

```
int* arr = new int[10];   // Allocate
delete[] arr;             // Deallocate (must not forget!)
```

   - Modern C++ approach with smart pointers (preferred):

```
auto arr = std::make_unique<int[]>(10);  // C++14: Automatically
↪  deallocated
```

- **C++23** introduces `std::make_unique_for_overwrite` and `std::make_shared_for_overwrite` for default-initialized arrays, optimizing performance when default initialization is unnecessary.

3. **Efficient Data Structures**:
   - Pointers are fundamental for implementing linked structures such as linked lists, trees, graphs, and hash tables.
   - Modern C++ combines pointers with smart pointers for safe ownership:

```cpp
struct TreeNode {
    int data;
    std::unique_ptr<TreeNode> left;   // Unique ownership of children
    std::unique_ptr<TreeNode> right;
    TreeNode(int val) : data(val) {}
};

// Tree with automatic cleanup
auto root = std::make_unique<TreeNode>(10);
root->left = std::make_unique<TreeNode>(5);
root->right = std::make_unique<TreeNode>(15);
// No manual deletion required
```

4. **Functionality and Flexibility**:
   - Pointers enable advanced techniques: function pointers for callbacks, polymorphism for runtime binding, and pointer-to-member for complex abstractions.
   - Function pointer example:

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

int compute(int x, int y, int (*op)(int, int)) {
    return op(x, y);
}

// C++23: std::function is often preferred for type-erased callables
std::function<int(int,int)> operation = add;
```

5. **Interfacing with C Libraries and System APIs**:
   - Many operating system APIs and C libraries use raw pointers extensively. Modern C++ provides safe bridges.

- **C++23** introduces `std::out_ptr` and `std::inout_ptr` for seamless smart pointer integration:

```cpp
// C API that allocates memory
extern "C" void create_resource(Resource** out);

std::unique_ptr<Resource, decltype(&destroy_resource)> resource(nullptr,
↪  destroy_resource);
create_resource(std::out_ptr(resource));  // Safely captures allocated
↪  memory
// resource automatically destroyed when out of scope
```

1.2.2 Common Use Cases for Pointers in C++

Pointers appear in a wide range of scenarios. Below are the most common use cases, with modern C++ best practices:

1. **Dynamic Arrays and Containers**:
   - While `std::vector` is preferred for most dynamic array needs, pointers are used when interfacing with legacy code or implementing custom containers.
   - Modern approach using `std::span` (C++20) for safe array views:

```cpp
void processData(std::span<int> data) {
    for (auto& elem : data) {
        elem *= 2;
    }
}

std::vector<int> vec = {1, 2, 3, 4, 5};
processData(vec);  // No raw pointers needed
```

2. **Polymorphism and Runtime Binding**:
   - Base class pointers (or smart pointers) enable polymorphic behavior through virtual functions.
   - Modern C++ recommends using `std::unique_ptr` or `std::shared_ptr` instead of raw pointers for owning polymorphic objects:

```cpp
class Shape {
public:
    virtual double area() const = 0;
```

```
    virtual ~Shape() = default;  // Virtual destructor essential
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() const override { return 3.14159 * radius * radius; }
};

// Using smart pointers for automatic cleanup
std::unique_ptr<Shape> shape = std::make_unique<Circle>(5.0);
std::cout << shape->area() << '\n';
```

3. **Resource Management (RAII)**:

   - Pointers are fundamental to RAII (Resource Acquisition Is Initialization), where resource lifetime is tied to object lifetime.
   - Modern C++ encapsulates this pattern in smart pointers and library types:

```
// File management with RAII
class FileHandle {
    std::FILE* file;
public:
    FileHandle(const char* filename, const char* mode)
        : file(std::fopen(filename, mode)) {}
    ~FileHandle() { if (file) std::fclose(file); }

    // Non-copyable, movable
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;
    FileHandle(FileHandle&& other) noexcept :
    ↪  file(std::exchange(other.file, nullptr)) {}

    std::FILE* get() const { return file; }
};

// Usage
FileHandle fh("data.txt", "r");
if (fh.get()) {
    // Use file - automatically closed when fh goes out of scope
}
```

4. **Function Pointers and Callbacks**:

   - Function pointers enable callback mechanisms, event handling, and strategy patterns.
   - Modern C++ offers more flexible alternatives:

```
// Traditional function pointer
void (*callback)(int) = [](int x) { std::cout << x << '\n'; };

// Modern alternatives:
// 1. std::function (type-erased, more flexible)
std::function<void(int)> modernCallback = [](int x) { std::cout << x <<
↪  '\n'; };

// 2. Templates for compile-time polymorphism (zero overhead)
template<typename Callable>
void execute(Callable&& callable, int value) {
    std::forward<Callable>(callable)(value);
}

execute([](int x) { std::cout << x * 2 << '\n'; }, 42);
```

5. **Low-Level Programming and Embedded Systems**:

   - Pointers are essential for hardware interaction, DMA controllers, and custom memory management.
   - **C++26 draft** proposes `std::observer_ptr` for non-owning pointer semantics, making intent explicit:

```
// Proposed C++26 syntax for non-owning hardware register access
void configure_timer(std::observer_ptr<volatile TimerRegisters> timer) {
    if (timer) {
        timer->control = 0x01;  // Safe access with null check
        timer->prescaler = 1000;
    }
}
```

6. **Optional Return Values**:

   - Pointers (especially `nullptr`) can represent optional or nullable return values.
   - Modern C++ provides better alternatives:

```cpp
// Raw pointer approach (traditional)
int* find(int value) { /* returns nullptr if not found */ }

// Modern alternatives:
// 1. std::optional (C++17) - safer, clearer intent
std::optional<int> findModern(int value) {
    // returns std::nullopt if not found
}

// 2. std::expected (C++23) - for operations that can fail with error
↪  info
std::expected<int, std::error_code> findWithError(int value) {
    // returns error code on failure
}
```

1.2.3 Modern C++ Enhancements for Pointers

Modern C++ (C++11 through C++23 and beyond) introduces features that dramatically improve pointer safety and expressiveness:

1. **Smart Pointers**:

   - **`std::unique_ptr`**: Exclusive ownership, zero overhead compared to raw pointers.

     ```cpp
     auto ptr = std::make_unique<MyClass>(args...);  // C++14
     // Cannot copy, only move
     auto moved = std::move(ptr);
     ```

   - **`std::shared_ptr`**: Shared ownership with thread-safe reference counting.

     ```cpp
     auto ptr1 = std::make_shared<MyClass>(args...);
     auto ptr2 = ptr1;  // Reference count increases
     // Object destroyed when last shared_ptr is destroyed
     ```

   - **`std::weak_ptr`**: Non-owning observer, breaks circular references.

     ```cpp
     std::weak_ptr<MyClass> weak = ptr1;
     if (auto locked = weak.lock()) {  // Get shared_ptr if still alive
         locked->doSomething();
     }
     ```

2. **C++23 Smart Pointer Utilities**:

   - `std::out_ptr` and `std::inout_ptr`: Safely pass smart pointers to C APIs that expect raw pointer outputs:

```cpp
// C API that returns allocated memory
extern "C" void get_data(void** out, size_t* size);

std::vector<uint8_t> data;
size_t size = 0;
get_data(std::out_ptr(data), &size);  // C++23: Automatically manages
↪  memory
```

- `std::make_unique_for_overwrite` and `std::make_shared_for_overwrite`:

```cpp
// Avoids value-initialization overhead when overwriting immediately
auto arr = std::make_unique_for_overwrite<int[]>(1000);
// Elements are default-initialized (uninitialized), faster for large
↪  arrays
```

3. **C++26 Preview: Vocabulary Types**:

- `std::observer_ptr`: A non-owning pointer with clear semantics, replacing raw pointers for observation:

```cpp
// Proposed C++26
void use_resource(std::observer_ptr<Resource> obs) {
    if (obs) {  // Explicit null check
        obs->doWork();
    }
    // No ownership concerns - observer_ptr never deletes
}
```

- `std::ptr_len`: Standard type for pointer-length pairs:

```cpp
// Proposed C++26
void process_buffer(std::ptr_len<const std::byte> buffer) {
    for (std::size_t i = 0; i < buffer.len(); ++i) {
        process(buffer.ptr()[i]);
    }
}
```

4. **`nullptr` and Type Safety**:

- `nullptr` (C++11) provides type-safe null pointer representation:

```cpp
void f(int);
void f(int*);
f(0);       // Calls f(int) - not what you want!
f(nullptr); // Calls f(int*) - correct
```

5. **Move Semantics and Perfect Forwarding**:

   - Move semantics enable efficient resource transfer:

```
std::unique_ptr<Widget> createWidget() {
    return std::make_unique<Widget>();  // Return value optimization +
    ↪  move
}

auto widget = createWidget();  // No copying, efficient
```

1.2.4 Advanced Use Cases and Techniques

1. **Custom Memory Allocators and Pools**:

   - Pointers enable custom memory management strategies for performance-critical applications.
   - Modern allocators can be integrated with standard containers:

```
// Fixed-size memory pool using pointers
template<typename T>
class PoolAllocator {
    std::vector<std::byte> storage;
    T* freeList{nullptr};

public:
    explicit PoolAllocator(std::size_t capacity)
        : storage(capacity * sizeof(T)) {
        // Initialize free list using pointer manipulation
        T* ptr = reinterpret_cast<T*>(storage.data());
        for (std::size_t i = 0; i < capacity - 1; ++i) {
            *reinterpret_cast<T**>(ptr) = ptr + 1;
            ++ptr;
        }
        *reinterpret_cast<T**>(ptr) = nullptr;
        freeList = reinterpret_cast<T*>(storage.data());
    }

    T* allocate() {
        if (!freeList) throw std::bad_alloc();
        T* result = freeList;
        freeList = *reinterpret_cast<T**>(freeList);
```

```
        return result;
    }

    void deallocate(T* ptr) {
        *reinterpret_cast<T**>(ptr) = freeList;
        freeList = ptr;
    }
};
```

2. **Pointer Arithmetic and Memory Optimization**:

   - Pointer arithmetic enables efficient buffer manipulation:

```
// Optimized memory copy using pointer arithmetic
void* fast_memcpy(void* dest, const void* src, std::size_t n) {
    auto* d = static_cast<std::byte*>(dest);
    const auto* s = static_cast<const std::byte*>(src);

    // Copy in 8-byte chunks where possible
    while (n >= 8) {
        *reinterpret_cast<uint64_t*>(d) = *reinterpret_cast<const
        ↪  uint64_t*>(s);
        d += 8;
        s += 8;
        n -= 8;
    }

    // Copy remaining bytes
    while (n--) {
        *d++ = *s++;
    }

    return dest;
}
```

3. **Interfacing with Legacy Code and C Libraries**:

   - Modern C++ provides tools to safely wrap legacy C interfaces:

```
// Wrapping a legacy C API with modern C++ safety
extern "C" {
    struct LegacyContext;
    LegacyContext* legacy_create();
    void legacy_destroy(LegacyContext*);
```

```
    int legacy_do_work(LegacyContext*, const char* input);
}

class LegacyWrapper {
    std::unique_ptr<LegacyContext, decltype(&legacy_destroy)> ctx;

public:
    LegacyWrapper() : ctx(legacy_create(), legacy_destroy) {
        if (!ctx) throw std::runtime_error("Failed to create context");
    }

    int doWork(const std::string& input) {
        return legacy_do_work(ctx.get(), input.c_str());
    }
};

// Usage - automatically cleans up
LegacyWrapper wrapper;
int result = wrapper.doWork("test");
```

1.2.5 Summary

- Pointers provide direct memory access, dynamic memory management, and enable complex data structures.
- They are essential for polymorphism, callback mechanisms, and interfacing with C libraries.
- Modern C++ (C++11 through C++23) introduces smart pointers that eliminate manual memory management risks.
- C++23 adds `out_ptr` utilities for safe C API interoperability and `make_unique_for_overwrite` for performance optimization.
- The upcoming C++26 standard is expected to introduce `observer_ptr` and `ptr_len` vocabulary types for clearer pointer semantics.
- While smart pointers are preferred for ownership, raw pointers remain valuable for non-owning observation and low-level operations.

Exercise 2. ***Practice Exercises:***

1. *Create a linked list using* `std::unique_ptr` *and demonstrate automatic cleanup.*
2. *Write a function that takes a* `std::span<int>` *and computes the sum of elements. Compare with a raw pointer version.*
3. *Using C++23, implement a wrapper for a C API that allocates memory, utilizing* `std::out_ptr`*.*
4. *Design a simple memory pool using raw pointers and compare its performance with standard allocation.*
5. *Research* `std::observer_ptr` *(C++26 draft) and explain how it improves code clarity over raw pointers for non-owning relationships.*

1.3 Memory Management in C++

Memory management is a critical aspect of C++ programming, especially when working with pointers. Understanding how memory is allocated, used, and deallocated is essential for writing efficient and bug-free programs. In this section, we will explore the **overview of stack and heap memory**, **how C++ manages memory**, and an **introduction to dynamic memory allocation**. We will also incorporate modern C++ concepts, including C++23 enhancements and C++26 preview features, to provide a comprehensive understanding of memory management in contemporary C++ programming.

1.3.1 Overview of Stack and Heap Memory

In C++, memory is divided into two primary regions: the **stack** and the **heap**. Each region has its own characteristics, advantages, and limitations.

1. **Stack Memory**:
    - The stack is a region of memory managed automatically by the compiler. It stores local variables, function parameters, and return addresses.
    - **Characteristics**:
        - **Fast Allocation and Deallocation**: Moving the stack pointer is extremely efficient.
        - **LIFO (Last-In, First-Out)**: Follows strict order; last allocated is first deallocated.
        - **Limited Size**: Typically small (e.g., 1-8 MB on most systems); exceeding capacity causes stack overflow.
        - **Automatic Management**: Variables are automatically destroyed when they go out of scope.
    - **Example**:

```
void function() {
    int x = 42;            // Allocated on stack
    std::array<int, 100> arr; // Also on stack
    // x and arr automatically destroyed when function exits
}
```

2. **Heap Memory**:

- The heap is a region of memory managed manually (or via smart pointers) by the programmer. It is used for dynamic allocation where size and lifetime are unknown at compile time.
- **Characteristics**:
  - **Flexible Size**: Can grow dynamically, limited only by available system memory.
  - **Manual Management**: Must be explicitly allocated and deallocated; failure leads to memory leaks.
  - **Slower Operations**: Allocation and deallocation involve more complex bookkeeping.
  - **Persistent Scope**: Memory remains allocated until explicitly freed, regardless of scope.
- **Example**:

```
void function() {
    int* ptr = new int(42);    // Allocated on heap
    // ... use ptr
    delete ptr;                 // Must explicitly deallocate
}
```

Important

Stack vs. Heap Summary :

Feature	Stack	Heap
Management	Automatic	Manual / Smart pointers
Speed	Very fast	Slower
Size	Limited (MB)	Large (GB)
Lifetime	Scope-bound	Programmer-controlled
Fragmentation	None	Possible

1.3.2 How C++ Manages Memory

C++ provides several mechanisms for memory management, ranging from automatic stack management to manual heap control and modern RAII techniques.

1. **Automatic Memory Management (Stack)**:

- Local variables are automatically allocated on entry to a scope and deallocated on exit. This is efficient and exception-safe.
- **Example**:

```
void foo() {
    std::string s = "Hello";  // Allocated on stack (the string object,
    ↪  not its data)
    // s is automatically destroyed when foo exits
}
```

2. **Manual Memory Management (Heap with `new`/`delete`)**:
   - The `new` operator allocates memory on the heap and optionally constructs objects.
   - The `delete` operator destroys objects and deallocates memory.
   - **Common Pitfalls**:
     - **Memory Leaks**: Forgetting to call `delete`.
     - **Dangling Pointers**: Using a pointer after `delete`.
     - **Double Deletion**: Calling `delete` twice on the same pointer.
     - **Mismatched Forms**: Using `delete` for array allocated with `new[]`.
   - **Example**:

```
int* p = new int(10);     // Allocate single int
delete p;                 // Correct

int* arr = new int[100]; // Allocate array
delete[] arr;             // Correct (note the [])

// Dangerous: delete arr;  // Undefined behavior!
```

3. **RAII (Resource Acquisition Is Initialization)**:
   - RAII is a fundamental C++ idiom where resource lifetime is tied to object lifetime. Resources are acquired in constructors and released in destructors.
   - This ensures exception safety and prevents resource leaks.
   - **Example**:

```
class FileHandle {
    std::FILE* file;
public:
```

```
    FileHandle(const char* name, const char* mode)
        : file(std::fopen(name, mode)) {
        if (!file) throw std::runtime_error("Cannot open file");
    }
    ~FileHandle() {
        if (file) std::fclose(file);
    }
    // Non-copyable, movable
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;
    FileHandle(FileHandle&& other) noexcept
        : file(std::exchange(other.file, nullptr)) {}
};

void processFile() {
    FileHandle fh("data.txt", "r");  // File opened
    // ... operations ...
}  // File automatically closed even if exception thrown
```

4. **Smart Pointers (Modern C++ Memory Management)**:

   - Smart pointers encapsulate raw pointers with automatic lifetime management.
   - **`std::unique_ptr`**: Exclusive ownership, non-copyable, movable.

     ```
     auto ptr = std::make_unique<int>(42);  // C++14
     // auto copy = ptr;                    // Error: no copy
     auto moved = std::move(ptr);           // Transfer ownership
     ```

   - **`std::shared_ptr`**: Shared ownership with reference counting.

     ```
     auto ptr1 = std::make_shared<int>(42);
     auto ptr2 = ptr1;                      // Shared ownership, count = 2
     // Object destroyed when last shared_ptr is destroyed
     ```

   - **`std::weak_ptr`**: Non-owning observer for shared_ptr.

     ```
     std::weak_ptr<int> weak = ptr1;
     if (auto locked = weak.lock()) {      // Safely acquire shared_ptr if
     ↪  object exists
         // Use locked
     }
     ```

1.3.3 Dynamic Memory Allocation

Dynamic memory allocation allows programs to allocate memory at runtime, essential when data structure sizes are unknown at compile time or when memory must outlive the allocating scope.

1. **Allocation with `new` and `new[]`**:

```
int* single = new int(10);         // Single object, initialized to 10
int* array = new int[100];         // Array of 100 uninitialized ints
int* defaultInit = new int();      // Value-initialized to 0
```

2. **Deallocation with `delete` and `delete[]`**:

```
delete single;     // Delete single object
delete[] array;    // Delete array
```

3. **Modern C++ Alternatives**:

   - Prefer `std::make_unique` and `std::make_shared` over raw `new`:

```
auto uptr = std::make_unique<int>(42);           // C++14
auto sptr = std::make_shared<MyClass>(args...);  // C++11
```

   - **C++23**: `std::make_unique_for_overwrite` for default-initialized arrays (performance optimization):

```
// Avoids value-initialization overhead when overwriting immediately
auto arr = std::make_unique_for_overwrite<int[]>(1000);
// Elements are uninitialized - faster for large arrays when you'll fill
↪  them
```

4. **Common Pitfalls and Prevention**:

   - **Memory Leaks**: Use smart pointers or RAII wrappers.
   - **Dangling Pointers**: Avoid storing raw pointers to managed objects; use `std::weak_ptr` or references.
   - **Double Deletion**: Smart pointers prevent this automatically.
   - **Exception Safety**: Smart pointers guarantee no leaks when exceptions occur.

1.3.4 Advanced Memory Management Techniques

1. **Custom Memory Allocators**:

   - Custom allocators allow optimization for specific allocation patterns and can be used with standard containers.
   - **Example: Simple linear allocator**:

```
class LinearAllocator {
    std::byte* buffer;
    std::size_t capacity;
    std::size_t offset{0};

public:
    LinearAllocator(std::size_t size)
        : buffer(static_cast<std::byte*>(std::malloc(size))),
        ↪  capacity(size) {}

    ~LinearAllocator() { std::free(buffer); }

    void* allocate(std::size_t size, std::size_t alignment =
    ↪  alignof(std::max_align_t)) {
        std::size_t alignedOffset = (offset + alignment - 1) &
        ↪  ~(alignment - 1);
        if (alignedOffset + size > capacity) return nullptr;
        offset = alignedOffset + size;
        return buffer + alignedOffset;
    }

    void reset() { offset = 0; }  // Reset for reuse

    // Non-copyable, movable
    LinearAllocator(const LinearAllocator&) = delete;
    LinearAllocator& operator=(const LinearAllocator&) = delete;
    LinearAllocator(LinearAllocator&& other) noexcept
        : buffer(std::exchange(other.buffer, nullptr)),
          capacity(std::exchange(other.capacity, 0)),
          offset(std::exchange(other.offset, 0)) {}
};
```

2. **Memory Pools**:

- Memory pools pre-allocate large blocks and serve fixed-size chunks, reducing fragmentation and allocation overhead.
- **Example: Fixed-size object pool**:

```
template<typename T>
class ObjectPool {
    union Node {
        T object;
        Node* next;
    };

    Node* freeList{nullptr};
    std::vector<std::unique_ptr<Node[]>> blocks;
    static constexpr std::size_t BLOCK_SIZE = 1024;

public:
    T* allocate() {
        if (!freeList) {
            allocateBlock();
        }
        Node* node = freeList;
        freeList = node->next;
        return std::launder(&node->object);
    }

    void deallocate(T* ptr) {
        Node* node = reinterpret_cast<Node*>(ptr);
        node->next = freeList;
        freeList = node;
    }

private:
    void allocateBlock() {
        auto block = std::make_unique<Node[]>(BLOCK_SIZE);
        for (std::size_t i = 0; i < BLOCK_SIZE - 1; ++i) {
            block[i].next = &block[i + 1];
        }
        block[BLOCK_SIZE - 1].next = nullptr;
        freeList = &block[0];
        blocks.push_back(std::move(block));
    }
};
```

3. **Placement New**:

   - Placement new constructs an object in pre-allocated memory, enabling custom memory management strategies.
   - **Example**:

```cpp
// Allocate raw memory
void* memory = std::malloc(sizeof(MyClass));

// Construct object in that memory
MyClass* obj = new (memory) MyClass(args...);

// Use obj...

// Manually call destructor
obj->~MyClass();

// Free raw memory
std::free(memory);
```

   - **C++23**: `std::destroy_at` provides a safer way to destruct objects:

```cpp
std::destroy_at(obj);  // C++17, safer than manual destructor call
```

1.3.5 Memory Management in Modern C++ (C++23 and Beyond)

Modern C++ introduces several features that enhance memory management safety and expressiveness:

1. **Smart Pointer Enhancements in C++23**:

   - `std::out_ptr` and `std::inout_ptr`: Safely pass smart pointers to C APIs:

```cpp
// C API that allocates memory
extern "C" void create_resource(void** out);

std::unique_ptr<Resource, decltype(&destroy_resource)> resource(nullptr,
↪  destroy_resource);
create_resource(std::out_ptr(resource));  // Automatically manages the
↪  allocated memory
```

   - `std::make_unique_for_overwrite` and `std::make_shared_for_overwrite`:

```cpp
// Performance optimization: skip value-initialization
auto buffer = std::make_unique_for_overwrite<uint8_t[]>(1024);
// Elements are uninitialized - you must fill them before reading
```

2. **C++26 Preview: Vocabulary Types**:

   - `std::observer_ptr`: A non-owning pointer with clear semantics, replacing raw pointers for observation:

   ```cpp
   // Proposed C++26
   void process(std::observer_ptr<const Widget> obs) {
       if (obs) {  // Explicit null check
           obs->doWork();
       }
       // observer_ptr never deletes
   }
   ```

   - `std::ptr_len`: Standard type for pointer-length pairs:

   ```cpp
   // Proposed C++26
   void process_buffer(std::ptr_len<const std::byte> buffer) {
       for (std::size_t i = 0; i < buffer.len(); ++i) {
           process(buffer.ptr()[i]);
       }
   }
   ```

3. **Stack Memory Safety (C++23)**:

   - `std::stacktrace` for debugging stack memory issues:

   ```cpp
   #include <stacktrace>

   void dangerous_recursion(int depth) {
       if (depth == 0) {
           std::cout << std::stacktrace::current() << '\n';  // Print call
           ↪  stack
           return;
       }
       dangerous_recursion(depth - 1);
   }
   ```

4. **Move Semantics and Perfect Forwarding**:

   - Move semantics enable efficient transfer of ownership without copying:

```cpp
std::unique_ptr<Widget> createWidget() {
    return std::make_unique<Widget>();  // RVO + move semantics
}

auto widget = createWidget();  // No copies, efficient
```

5. **Standard Library Containers**:
   - Containers like `std::vector`, `std::array`, and `std::string` eliminate manual memory management in most scenarios.
   - **C++23**: `std::vector` now supports `resize_for_overwrite` for default-initialized resizing:

```cpp
std::vector<int> vec;
vec.resize_for_overwrite(1000);  // Allocates without initializing
↪  elements
// Fill elements immediately for better performance
```

1.3.6 Best Practices for Memory Management

1. **Prefer Smart Pointers for Ownership**:
   - Use `std::unique_ptr` for exclusive ownership, `std::shared_ptr` for shared ownership. Avoid raw owning pointers.
2. **Use RAII Consistently**:
   - Encapsulate all resources (memory, files, locks, sockets) in RAII classes.
3. **Prefer Stack Allocation When Possible**:
   - Stack allocation is faster, safer, and automatically managed. Use heap only when necessary.
4. **Avoid Manual `new` and `delete`**:
   - Use `std::make_unique`, `std::make_shared`, containers, and RAII wrappers instead.
5. **Check for Null Pointers**:

- Always verify raw pointers before dereferencing. Use `std::observer_ptr` (C++26) to document non-owning pointers.

6. **Use `nullptr` Instead of `NULL` or `0`**:
   - `nullptr` is type-safe and avoids overload resolution issues.
7. **Address Sanitizers and Tools**:
   - Use compiler sanitizers (`-fsanitize=address`) and static analyzers to detect memory errors.

1.3.7 Summary

- Stack memory is fast, automatic, and scope-bound; heap memory is flexible but requires explicit management.
- RAII is the cornerstone of safe C++ memory management, tying resource lifetime to object lifetime.
- Smart pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`) automate memory management and prevent leaks.
- C++23 introduces `out_ptr` utilities and `make_unique_for_overwrite` for performance-critical code.
- The upcoming C++26 standard will add `observer_ptr` and `ptr_len` for clearer non-owning pointer semantics.
- Advanced techniques like custom allocators and memory pools enable optimization for specific use cases.
- Following best practices and using modern tools ensures memory safety and program correctness.

Exercise 3. ***Practice Exercises:***

1. *Create a RAII wrapper for a POSIX file descriptor (using `open()` and `close()`) that ensures the descriptor is closed when the wrapper goes out of scope.*

2. *Implement a simple memory pool that allocates fixed-size blocks (e.g., 64 bytes each) and handles allocation and deallocation.*

3. *Using C++23, write a function that safely interfaces with a C API that returns allocated memory, using* `std::out_ptr`.

4. *Compare the performance of* `std::vector<int>` *vs manually allocated arrays with* `new[]` *for large allocations. Explain the differences.*

5. *Research* `std::observer_ptr` *(C++26 draft) and write a small example demonstrating its usage for non-owning observation.*

Chapter 2

Basics of Pointers in C++

2.1 Declaring and Initializing Pointers

Pointers are one of the most powerful and fundamental features of C++. They allow programmers to directly manipulate memory, enabling efficient and flexible programming. In this section, we will explore the **syntax for declaring pointers**, **initializing pointers**, and provide **examples of declaring and using pointers** to different data types such as `int`, `char`, and `double`. We will also incorporate modern C++ concepts, including C++23 enhancements and C++26 preview features, to provide a comprehensive understanding of pointers in contemporary C++ programming.

2.1.1 Syntax for Declaring Pointers

A pointer is a variable that stores the memory address of another variable or function. To declare a pointer, you specify the data type it points to, followed by an asterisk (`*`), and then the pointer's name. The general syntax is:

```
data_type* pointer_name;
```

Here, `data_type` is the type of data the pointer will point to (e.g., `int`, `char`, `double`), and `pointer_name` is the name of the pointer variable.

Important

Pointer Declaration Style: Modern C++ style favors `int* ptr` (the asterisk attached to the type), emphasizing that the type is "pointer to int." However, be cautious when declaring multiple pointers:

```
int* p1, p2;    // p1 is a pointer, p2 is an integer (surprise!)
int *p1, *p2;   // Both are pointers
int* p1, *p2;   // Clear: both are pointers
```

Examples:

- Declaring a pointer to an `int`:

```
int* intPtr;
```

- Declaring a pointer to a `char`:

```
char* charPtr;
```

- Declaring a pointer to a `double`:

```
double* doublePtr;
```

- Declaring a pointer to a `const` type:

```
const int* constPtr;        // Pointer to const int (cannot modify pointed
↪  value)
int* const constPtr2;       // Const pointer to int (cannot change address)
const int* const constPtr3; // Const pointer to const int
```

2.1.2 Initializing Pointers

Once a pointer is declared, it should be initialized before use. Uninitialized pointers contain indeterminate addresses and lead to undefined behavior. There are several ways to initialize pointers:

1. **Null Pointers**:
    - A null pointer points to no valid memory location. In modern C++, the preferred way is `nullptr` (C++11 and later).
    - **Example**:

```
int* ptr = nullptr; // Type-safe null pointer
```

- `nullptr` is superior to `NULL` or `0` because it avoids overload resolution ambiguities:

```
void f(int);
void f(int*);
f(NULL);    // Calls f(int) - ambiguous and dangerous!
f(nullptr); // Calls f(int*) - correct
```

2. **Pointers to Existing Variables**:

   - Use the address-of operator (`&`) to obtain the memory address of an existing variable.
   - **Example**:

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

3. **Dynamic Memory Allocation (Raw)**:

   - Allocate memory on the heap using `new`. **Note**: In modern C++, prefer smart pointers for owning pointers.
   - **Example**:

```
int* ptr = new int(42);      // Single object, initialized to 42
int* arr = new int[100];     // Array of 100 uninitialized ints
delete ptr;                  // Must deallocate
delete[] arr;                // Must use array form
```

4. **Smart Pointers (Modern C++)**:

   - Smart pointers automatically manage memory lifetime, preventing leaks.
   - **`std::unique_ptr`**: Exclusive ownership, zero overhead.

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);  // C++14
```

   - **`std::shared_ptr`**: Shared ownership with reference counting.

```
std::shared_ptr<int> ptr = std::make_shared<int>(42);  // C++11
```

   - **C++23**: `std::make_unique_for_overwrite` for performance:

```
auto arr = std::make_unique_for_overwrite<int[]>(1000); // Uninitialized,
↪  faster
```

2.1.3 Examples: Declaring and Using Pointers to Different Data Types

Below are comprehensive examples demonstrating pointer usage with various data types, incorporating modern C++ concepts.

1. **Pointer to `int`**:

```cpp
int value = 42;
int* ptr = &value;

std::cout << "Address: " << ptr << '\n';      // Prints memory address
std::cout << "Value: " << *ptr << '\n';       // Prints 42

*ptr = 100;  // Modify through pointer
std::cout << "Modified value: " << value << '\n';  // Prints 100
```

2. **Pointer to `char`**:

```cpp
char ch = 'A';
char* ptr = &ch;

std::cout << "Character: " << *ptr << '\n';   // Prints 'A'

// String literals (C-style strings)
const char* str = "Hello, World!";
std::cout << str << '\n';  // Prints the entire string
```

3. **Pointer to `double`**:

```cpp
double pi = 3.14159;
double* ptr = &pi;

std::cout << "Value: " << *ptr << '\n';       // Prints 3.14159
*ptr = 2.71828;  // Modify through pointer
std::cout << "New value: " << pi << '\n';     // Prints 2.71828
```

4. **Null Pointer with Safety Check**:

```cpp
int* ptr = nullptr;

if (ptr == nullptr) {
    std::cout << "Pointer is null, safe to initialize\n";
    ptr = new int(42);  // Now allocate
}
```

```cpp
// Always check before dereferencing
if (ptr != nullptr) {
    std::cout << "Value: " << *ptr << '\n';
}

delete ptr;  // Don't forget to clean up
```

5. **Smart Pointer Example (Modern C++)**:

```cpp
// Unique ownership
auto uniqueInt = std::make_unique<int>(42);
std::cout << *uniqueInt << '\n';  // 42
// No manual delete needed

// Shared ownership
auto sharedInt = std::make_shared<int>(100);
auto sharedInt2 = sharedInt;  // Reference count = 2
std::cout << *sharedInt << '\n';  // 100
// Automatically deleted when last reference goes out of scope

// C++23: for-overwrite for performance
auto buffer = std::make_unique_for_overwrite<uint8_t[]>(1024);
// buffer elements are uninitialized - fill before reading!
std::fill_n(buffer.get(), 1024, uint8_t(0));
```

2.1.4 Modern C++ Concepts in Pointer Declaration and Initialization

Modern C++ (C++11 through C++23 and beyond) introduces several features that enhance pointer safety and usability:

1. **Smart Pointers**:

   - **`std::unique_ptr`**: Exclusive ownership, cannot be copied, only moved.

     ```cpp
     auto ptr = std::make_unique<MyClass>(args...);
     auto moved = std::move(ptr);  // Transfer ownership
     // ptr now empty
     ```

   - **`std::shared_ptr`**: Shared ownership with atomic reference counting.

     ```cpp
     auto ptr1 = std::make_shared<MyClass>(args...);
     auto ptr2 = ptr1;  // Reference count = 2
     ```

   - **`std::weak_ptr`**: Non-owning observer for shared_ptr, breaks circular references.

```
std::weak_ptr<MyClass> weak = ptr1;
if (auto locked = weak.lock()) {  // Safely acquire shared_ptr if object
↪  exists
    locked->doWork();
}
```

2. **C++23 Smart Pointer Utilities**:

   - `std::out_ptr` and `std::inout_ptr`: Safely pass smart pointers to C APIs:

   ```
   // C API that allocates memory
   extern "C" void create_buffer(void** out, size_t* size);

   std::vector<uint8_t> buffer;
   size_t size = 0;
   create_buffer(std::out_ptr(buffer), &size);  // C++23: Automatically
   ↪  manages memory
   ```

   - `std::make_unique_for_overwrite` and `std::make_shared_for_overwrite`:

   ```
   // Skip value-initialization for performance
   auto arr = std::make_unique_for_overwrite<int[]>(1000);
   // Elements are default-initialized (uninitialized) - faster
   ```

3. **C++26 Preview: Vocabulary Types**:

   - `std::observer_ptr`: Non-owning pointer with clear semantics:

   ```
   // Proposed C++26
   void process(std::observer_ptr<const Widget> obs) {
       if (obs) {  // Explicit null check
           obs->doWork();
       }
       // observer_ptr never deletes
   }
   ```

   - `std::ptr_len`: Standard type for pointer-length pairs:

   ```
   // Proposed C++26
   void process_buffer(std::ptr_len<const std::byte> buffer) {
       for (std::size_t i = 0; i < buffer.len(); ++i) {
           buffer.ptr()[i];  // Safe access
       }
   }
   ```

4. **`auto` for Pointer Declarations**:

- `auto` simplifies pointer declarations, especially with complex types:

```
auto ptr = new int(42);            // int* ptr
auto* ptr2 = new int(42);          // int* ptr2 (explicit)
auto smart = std::make_unique<int>(42);  // std::unique_ptr<int>
```

5. **Type Deduction with `decltype`**:

   - `decltype` can deduce pointer types for generic code:

```
int x = 42;
decltype(&x) ptr = &x;  // int* ptr
```

2.1.5 Best Practices for Declaring and Initializing Pointers

1. **Prefer Smart Pointers for Ownership**:

   - Use `std::unique_ptr` for exclusive ownership, `std::shared_ptr` for shared ownership. Reserve raw pointers for non-owning observation.

2. **Always Initialize Pointers**:

   - Uninitialized pointers cause undefined behavior. Initialize with `nullptr` if no valid address is available.

3. **Use `nullptr` Instead of `NULL` or `0`**:

   - `nullptr` is type-safe and avoids overload resolution issues.

4. **Check for Null Before Dereferencing**:

   - Always verify that a raw pointer is not null before dereferencing:

```
if (ptr != nullptr) {
    *ptr = 42;  // Safe
}
```

5. **Avoid Manual `new` and `delete`**:

   - Use `std::make_unique`, `std::make_shared`, and standard containers instead.

6. **Use `const` Correctly**:

   - Use `const` to express immutability:

```
const int* ptr = &x;       // Cannot modify pointed value
int* const ptr = &x;       // Cannot change address
const int* const ptr = &x; // Both
```

7. **Prefer References Over Raw Pointers for Non-nullable Parameters**:

   - Use references when the parameter must refer to a valid object:

```
void process(int& value);  // value must be valid
void process(int* value);  // value may be nullptr
```

2.1.6 Advanced Techniques and Use Cases

1. **Pointer to Pointer (Double Pointers)**:

   - Useful for functions that need to modify a pointer's address.
   - **Example**:

```
void allocateInt(int** ptr) {
    *ptr = new int(42);
}

int* p = nullptr;
allocateInt(&p);
std::cout << *p << '\n';  // 42
delete p;
```

   - Modern C++ alternative using references to smart pointers:

```
void allocateInt(std::unique_ptr<int>& ptr) {
    ptr = std::make_unique<int>(42);
}
```

2. **Pointer Arithmetic**:

   - Pointer arithmetic enables efficient array traversal:

```
int arr[] = {10, 20, 30, 40, 50};
int* ptr = arr;  // Points to first element

for (int i = 0; i < 5; ++i) {
    std::cout << *(ptr + i) << ' ';  // Access by arithmetic
}
```

   - **C++20**: `std::span` provides a safer alternative:

```
std::span<int> span(arr);
for (int& elem : span) {  // No pointer arithmetic needed
    std::cout << elem << ' ';
}
```

3. **Function Pointers**:

   - Pointers to functions enable callbacks and strategy patterns:

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

int compute(int x, int y, int (*operation)(int, int)) {
    return operation(x, y);
}

// C++17: std::invoke for unified call syntax
#include <functional>
int result = std::invoke(operation, x, y);
```

4. **Member Pointers**:

   - Pointers to class members enable advanced abstractions:

```
struct Point {
    int x, y;
};

int Point::*ptr = &Point::x;  // Pointer to member
Point p{10, 20};
std::cout << p.*ptr << '\n';   // 10
```

2.1.7 Summary

- Pointers store memory addresses and are declared with `type* name`.
- Always initialize pointers: with `nullptr`, address of variable, or dynamic allocation.
- `nullptr` is the type-safe null pointer representation in modern C++.
- Smart pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`) automate memory management and should be preferred for owning pointers.

- C++23 adds `make_unique_for_overwrite` for performance and `out_ptr` for C API interoperability.
- C++26 will introduce `observer_ptr` and `ptr_len` for clearer non-owning pointer semantics.
- Best practices: initialize pointers, check for null, prefer smart pointers, use `const` appropriately.

Exercise 4. ***Practice Exercises:***

1. *Declare and initialize a pointer to an integer, then modify the integer through the pointer. Print the value before and after.*
2. *Create a pointer to a `const int`. Try to modify the pointed value and observe the compiler error.*
3. *Using `std::unique_ptr`, create a pointer to a dynamically allocated `std::string`. Initialize it with "Hello, Modern C++!" and print its length.*
4. *Write a function that takes a `std::shared_ptr<int>` and returns a `std::weak_ptr<int>` to the same object. Demonstrate how to safely lock and use the weak pointer.*
5. *Research `std::observer_ptr` (C++26 draft) and write a small example showing how it differs from raw pointers for non-owning observation.*
6. *Implement a simple function that uses pointer arithmetic to reverse an array in place.*

2.2 Pointer Arithmetic

Pointer arithmetic is a powerful feature of C++ that allows you to perform arithmetic operations on pointers. This capability is particularly useful when working with arrays, dynamic memory, and low-level programming. However, it also requires careful attention to avoid undefined behavior. In this section, we will explore **adding and subtracting integers from pointers**, **pointer differences and comparisons**, and provide **examples of traversing arrays using pointer arithmetic**. We will also incorporate modern C++ concepts, such as `std::span`, smart pointers, and range-based for loops, to provide a comprehensive understanding of pointer arithmetic in contemporary C++ programming.

2.2.1 Adding and Subtracting Integers from Pointers

Pointer arithmetic involves adding or subtracting integers from pointers. When you perform arithmetic operations on a pointer, the pointer is adjusted by the size of the data type it points to (i.e., `sizeof(T)`). This allows you to navigate through arrays and memory blocks efficiently.

Important

Key Concept: Adding `n` to a pointer of type `T*` advances it by `n * sizeof(T)` bytes. The compiler handles the multiplication automatically.

1. **Adding Integers to Pointers**:
   - Adding an integer `n` to a pointer moves it forward by `n` elements.
   - **Example**:

```cpp
int arr[5] = {10, 20, 30, 40, 50};
int* ptr = arr;         // ptr points to arr[0]
ptr = ptr + 2;          // ptr now points to arr[2] (30)
std::cout << *ptr << '\n';  // Output: 30
```

2. **Subtracting Integers from Pointers**:
   - Subtracting an integer `n` from a pointer moves it backward by `n` elements.
   - **Example**:

```cpp
int arr[5] = {10, 20, 30, 40, 50};
int* ptr = arr + 4;     // ptr points to arr[4] (50)
```

```
ptr = ptr - 2;          // ptr now points to arr[2] (30)
std::cout << *ptr << '\n';  // Output: 30
```

3. **Increment and Decrement Operators**:
   - The ++ and −− operators provide convenient shorthand.
   - **Example**:

```
int arr[5] = {10, 20, 30, 40, 50};
int* ptr = arr;         // Points to arr[0]
ptr++;                  // Now points to arr[1] (20)
std::cout << *ptr << '\n';  // Output: 20
ptr--;                  // Back to arr[0] (10)
std::cout << *ptr << '\n';  // Output: 10
```

4. **Compound Assignment**:
   - Compound operators like += and −= work similarly.
   - **Example**:

```
int* ptr = arr;
ptr += 3;               // Equivalent to ptr = ptr + 3
std::cout << *ptr << '\n';  // Output: 40 (arr[3])
```

2.2.2 Pointer Differences and Comparisons

Pointer arithmetic also allows you to calculate the difference between two pointers and compare their relative positions.

1. **Pointer Differences**:
   - The difference between two pointers of the same type yields the number of elements between them. The result type is `std::ptrdiff_t` (a signed integer type).
   - **Example**:

```
int arr[5] = {10, 20, 30, 40, 50};
int* ptr1 = arr;          // Points to arr[0]
int* ptr2 = arr + 3;      // Points to arr[3]
std::ptrdiff_t diff = ptr2 - ptr1;  // diff = 3
std::cout << "Elements between: " << diff << '\n';
```

   - **Important**: Subtracting pointers that do not point into the same array (or one past the end) results in undefined behavior.

2. **Pointer Comparisons**:

   - Relational operators (<, >, <=, >=, ==, !=) can compare pointers from the same array.
   - **Example**:

```
int arr[5] = {10, 20, 30, 40, 50};
int* ptr1 = arr;          // arr[0]
int* ptr2 = arr + 3;      // arr[3]

if (ptr1 < ptr2) {
    std::cout << "ptr1 points to an earlier element\n";
}
if (ptr1 != nullptr) {
    std::cout << "ptr1 is not null\n";
}
```

   - **C++20**: Comparison with `nullptr` is well-defined; comparing pointers from different arrays yields unspecified results (though often implementation-defined).

2.2.3 Example: Traversing an Array Using Pointer Arithmetic

Pointer arithmetic is commonly used to traverse arrays. Below are multiple approaches, including modern alternatives.

1. **Traditional Pointer Traversal**:

```
#include <iostream>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int* ptr = arr;

    for (int i = 0; i < 5; ++i) {
        std::cout << "Element " << i << ": " << *ptr << '\n';
        ++ptr;  // Move to next element
    }

    return 0;
}
```

2. **Using Pointer Arithmetic with Bounds Check**:

```cpp
int arr[5] = {10, 20, 30, 40, 50};
int* begin = arr;
int* end = arr + 5;  // Pointer to one past the last element (safe for
↪  comparison)

for (int* ptr = begin; ptr != end; ++ptr) {
    std::cout << *ptr << ' ';
}
```

3. **Modern C++ Alternative: Range-based For Loop (Safer)**:

```cpp
int arr[5] = {10, 20, 30, 40, 50};
for (int& element : arr) {
    std::cout << element << ' ';
}
```

4. **C++20 Alternative: `std::span` (Bounds-Safe View)**:

```cpp
#include <span>

int arr[5] = {10, 20, 30, 40, 50};
std::span<int> span(arr);
for (int& element : span) {
    std::cout << element << ' ';
}
// Or with pointer arithmetic (still safe within span)
auto* ptr = span.data();
for (std::size_t i = 0; i < span.size(); ++i) {
    std::cout << ptr[i] << ' ';
}
```

2.2.4 Modern C++ Concepts in Pointer Arithmetic

Modern C++ provides safer alternatives and best practices that reduce the need for manual pointer arithmetic while preserving performance when needed.

1. **`std::span` (C++20)**:
    - `std::span` provides a safe, non-owning view of contiguous memory with bounds information.
    - **Example**:

```cpp
#include <span>

void process(std::span<int> data) {
    // Safe access with bounds checking in debug builds
    for (std::size_t i = 0; i < data.size(); ++i) {
        data[i] *= 2;  // Bounds-safe (if compiled with assertions)
    }

    // Or use pointer arithmetic with explicit bounds
    int* ptr = data.data();
    for (std::size_t i = 0; i < data.size(); ++i) {
        ptr[i] *= 2;
    }
}

std::vector<int> vec = {1, 2, 3, 4, 5};
process(vec);  // std::span can adapt to any contiguous container
```

2. **Range-based For Loops**:

   - Range-based for loops eliminate manual pointer arithmetic for iteration.
   - **Example**:

```cpp
std::vector<int> vec = {1, 2, 3, 4, 5};
for (int& elem : vec) {
    elem *= 2;  // No pointers needed
}
```

3. **Smart Pointers with Arrays**:

   - Smart pointers can manage dynamic arrays safely, though pointer arithmetic requires using `.get()` carefully.
   - **Example**:

```cpp
auto arr = std::make_unique<int[]>(5);
for (int i = 0; i < 5; ++i) {
    arr[i] = i * 10;  // operator[] is safe with unique_ptr<T[]>
}

// Pointer arithmetic (use with caution)
int* ptr = arr.get();
for (int i = 0; i < 5; ++i) {
    std::cout << *(ptr + i) << ' ';
```

```
}
```

4. **C++23: `std::mdspan` for Multidimensional Views**:

   - `std::mdspan` (C++23) provides a multidimensional view with safe indexing.
   - **Example**:

```
#include <mdspan>

int data[12] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
std::mdspan<int, std::extents<size_t, 3, 4>> matrix(data);

// Access elements with 2D indices instead of pointer arithmetic
for (size_t i = 0; i < matrix.extent(0); ++i) {
    for (size_t j = 0; j < matrix.extent(1); ++j) {
        std::cout << matrix[i, j] << ' ';  // No manual pointer
        ↪  arithmetic
    }
    std::cout << '\n';
}
```

5. **C++26 Preview: `std::ptr_len`**:

   - The proposed `std::ptr_len` type encapsulates pointer-length pairs, making pointer arithmetic safer.
   - **Example**:

```
// Proposed C++26
void process_buffer(std::ptr_len<const int> buffer) {
    // Safe iteration with bounds
    for (std::size_t i = 0; i < buffer.len(); ++i) {
        process(buffer.ptr()[i]);
    }

    // Or using pointer arithmetic with explicit bounds
    const int* ptr = buffer.ptr();
    for (std::size_t i = 0; i < buffer.len(); ++i) {
        process(*(ptr + i));
    }
}
```

2.2.5 Best Practices for Pointer Arithmetic

1. **Avoid Out-of-Bounds Access**:
    - Always ensure pointer arithmetic stays within the bounds of the array or allocated memory. Dereferencing pointers outside bounds is undefined behavior.
    - Use the *one-past-the-end* pointer for comparisons, but never dereference it.
2. **Prefer `std::span` and Containers**:
    - Use `std::span` (C++20) for non-owning views and standard containers for owning memory. They provide bounds information and safer iteration.
3. **Use Range-based For Loops**:
    - When iterating over containers, prefer range-based for loops over manual pointer arithmetic for clarity and safety.
4. **Check for Null Before Arithmetic**:
    - Adding to a null pointer is undefined behavior. Always verify pointers are not null before arithmetic.
5. **Use `std::ptrdiff_t` for Differences**:
    - Use `std::ptrdiff_t` (from `<cstddef>`) for pointer differences; it's the correct signed type.
6. **Limit Pointer Arithmetic to Arrays**:
    - Pointer arithmetic is only well-defined within the same array (or one past the end). Avoid arithmetic on pointers to individual objects unless they're part of an array.
7. **Consider `std::mdspan` for Multidimensional Data**:
    - For multidimensional arrays, `std::mdspan` (C++23) provides a safer, more expressive alternative to manual pointer arithmetic.

2.2.6 Advanced Techniques and Use Cases

1. **Pointer Arithmetic with Multidimensional Arrays**:
   - For static 2D arrays, pointer arithmetic can simulate row-major traversal.
   - **Example**:

```cpp
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

// Flattened traversal using pointer arithmetic
int* flat = &matrix[0][0];
for (int i = 0; i < 3 * 4; ++i) {
    std::cout << flat[i] << ' ';
}

// Or using pointer arithmetic with rows
for (int i = 0; i < 3; ++i) {
    int* row = matrix[i];
    for (int j = 0; j < 4; ++j) {
        std::cout << *(row + j) << ' ';
    }
    std::cout << '\n';
}
```

   - **C++23 Alternative**: `std::mdspan` provides a safer multidimensional view.

2. **Pointer Arithmetic with Dynamic Arrays**:
   - Pointer arithmetic works identically with dynamically allocated arrays.
   - **Example**:

```cpp
size_t size = 10;
int* arr = new int[size];
for (size_t i = 0; i < size; ++i) {
    arr[i] = i * 10;  // Indexing
}

// Pointer arithmetic traversal
int* ptr = arr;
```

```
int* end = arr + size;
while (ptr != end) {
    std::cout << *ptr++ << ' ';
}
delete[] arr;
```

- **Modern Alternative**: `std::vector` and `std::unique_ptr<T[]>` with `.get()`.

3. **Pointer Arithmetic in Generic Algorithms**:
   - Template algorithms often use pointer arithmetic for efficiency.
   - **Example: Custom `copy`**:

```
template<typename InputIt, typename OutputIt>
OutputIt my_copy(InputIt first, InputIt last, OutputIt d_first) {
    while (first != last) {
        *d_first++ = *first++;
    }
    return d_first;
}

int src[] = {1, 2, 3, 4, 5};
int dest[5];
my_copy(src, src + 5, dest);  // Works with pointers and iterators
```

4. **Pointer Arithmetic with Placement New and Custom Allocators**:
   - Low-level memory management often uses pointer arithmetic to navigate raw memory.
   - **Example**:

```
// Custom memory pool using pointer arithmetic
class PoolAllocator {
    std::byte* buffer;
    std::size_t capacity;
    std::size_t offset{0};

public:
    PoolAllocator(std::size_t size)
        : buffer(static_cast<std::byte*>(std::malloc(size))),
        ↪  capacity(size) {}

    ~PoolAllocator() { std::free(buffer); }
```

```
    void* allocate(std::size_t size, std::size_t alignment =
    ↪  alignof(std::max_align_t)) {
        std::size_t alignedOffset = (offset + alignment - 1) &
        ↪  ~(alignment - 1);
        if (alignedOffset + size > capacity) return nullptr;
        void* result = buffer + alignedOffset;
        offset = alignedOffset + size;
        return result;
    }

    void reset() { offset = 0; }  // Reset pointer arithmetic
};
```

2.2.7 Common Pitfalls and Undefined Behavior

1. **Out-of-Bounds Access**:

```
int arr[5] = {1, 2, 3, 4, 5};
int* ptr = arr + 5;   // One past the end (okay for comparison)
// *ptr = 6;          // UNDEFINED: dereferencing one-past-end
```

2. **Subtracting Pointers from Different Arrays**:

```
int a[5], b[5];
int* p = a;
int* q = b;
auto diff = q - p;     // UNDEFINED: pointers from different arrays
```

3. **Arithmetic on Null Pointers**:

```
int* ptr = nullptr;
ptr++;                 // UNDEFINED: arithmetic on null pointer
```

4. **Integer Overflow**:

    - Adding very large integers to pointers may wrap around, leading to undefined behavior.

2.2.8 Summary

- Pointer arithmetic advances pointers by multiples of `sizeof(T)` based on the pointed type.
- Adding `n` moves forward by `n` elements; subtracting moves backward.

- Pointer differences (`std::ptrdiff_t`) give the number of elements between two pointers.
- Comparisons are valid only for pointers within the same array (or one past the end).
- Modern C++ provides safer alternatives: `std::span` for contiguous views, range-based for loops for iteration, and `std::mdspan` (C++23) for multidimensional access.
- C++26 will introduce `std::ptr_len` for explicit pointer-length pairs.
- Best practices: avoid out-of-bounds access, prefer containers over raw arrays, check for null, and use standard library algorithms when possible.

Exercise 5. ***Practice Exercises:***

1. *Write a function that reverses an array in place using pointer arithmetic (without using indices).*
2. *Given a 2D array (e.g., `int matrix[3][4]`), write a function that prints all elements using pointer arithmetic to traverse in row-major order.*
3. *Implement a function that finds the maximum element in an array using pointer arithmetic. Return a pointer to the maximum element.*
4. *Using `std::span` (C++20), write a function that computes the sum of all elements. Compare the implementation with a raw pointer version.*
5. *Research `std::mdspan` (C++23) and write an example that creates a 2D view of a 1D array. Show how to access elements without manual pointer arithmetic.*
6. *Implement a simple circular buffer using pointer arithmetic and raw memory. Ensure proper wrap-around behavior.*

2.3 Dereferencing Pointers

Dereferencing pointers is a fundamental concept in C++ that allows you to access and modify the value stored at the memory address held by a pointer. This capability is essential for working with dynamic memory, passing data by reference, and implementing various algorithms and data structures. In this section, we will explore **accessing and modifying values through pointers** and provide **examples of swapping values and working with complex types**. We will also incorporate modern C++ concepts, such as smart pointers, references, and the new C++23/26 vocabulary types, to provide a comprehensive understanding of dereferencing pointers in contemporary C++ programming.

2.3.1 Accessing and Modifying Values Through Pointers

Dereferencing a pointer involves using the **dereference operator (*)** to access or modify the value stored at the memory address the pointer points to. This allows you to indirectly manipulate data, a key feature of pointer-based programming.

Important

Always Check for Null: Dereferencing a null pointer or a dangling pointer results in undefined behavior, typically causing a program crash. Always verify pointers before dereferencing.

1. **Accessing Values**:
   - Use the dereference operator (*) to read the value at the pointed address.
   - **Example**:

```
int x = 42;
int* ptr = &x;         // ptr holds address of x
int value = *ptr;      // value = 42
std::cout << "Value: " << value << '\n';
```

2. **Modifying Values**:
   - Use the dereference operator on the left side of assignment to modify the pointed value.
   - **Example**:

```
int x = 42;
int* ptr = &x;
*ptr = 100;            // x now equals 100
std::cout << "New value of x: " << x << '\n';
```

3. **Dereferencing Smart Pointers**:

   - Smart pointers (`std::unique_ptr`, `std::shared_ptr`) overload the `*` and `->` operators for seamless dereferencing.
   - **Example**:

```
auto ptr = std::make_unique<int>(42);
*ptr = 100;            // Modify through unique_ptr
std::cout << *ptr << '\n';  // Output: 100

auto shared = std::make_shared<std::string>("Hello");
shared->append(" World");  // Use -> operator with smart pointer
std::cout << *shared << '\n';  // Output: Hello World
```

4. **Dereferencing with `std::optional` and `std::expected`**:

   - Modern C++ provides vocabulary types that represent optional or fallible values, often used with pointers.
   - **C++17 `std::optional`**:

```
std::optional<int> opt = 42;
if (opt) {
    std::cout << *opt << '\n';  // Dereference optional
}
```

   - **C++23 `std::expected`**:

```
std::expected<int, std::error_code> result = 42;
if (result) {
    std::cout << *result << '\n';  // Dereference expected
}
```

2.3.2 Example: Swapping Values Using Pointers and References

A classic example demonstrating dereferencing is swapping two values. This illustrates how pointers enable direct memory manipulation.

1. **Swapping with Raw Pointers**:

```cpp
#include <iostream>

void swap(int* a, int* b) {
    int temp = *a;     // Dereference to get value
    *a = *b;           // Assign through pointer
    *b = temp;         // Assign through pointer
}

int main() {
    int x = 10, y = 20;
    std::cout << "Before: x = " << x << ", y = " << y << '\n';

    swap(&x, &y);      // Pass addresses

    std::cout << "After:  x = " << x << ", y = " << y << '\n';
    return 0;
}
```

2. **Swapping with References (Modern C++ Preferred)**:

```cpp
void swap(int& a, int& b) {
    int temp = a;      // No dereferencing needed
    a = b;
    b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(x, y);        // Cleaner syntax, safer
    // x = 20, y = 10
}
```

3. **Swapping with Generic Templates**:

```cpp
template<typename T>
void swap(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}
```

2.3.3 Modern C++ Concepts in Dereferencing Pointers

Modern C++ provides safer alternatives and best practices that reduce the risks associated with pointer dereferencing.

1. **References**:
   - References provide a safer, more intuitive alternative to pointers for aliasing. They must be initialized and cannot be null or reassigned.
   - **Example**:

```cpp
int x = 42;
int& ref = x;          // ref is an alias for x
ref = 100;             // Modifies x directly (no dereferencing)
std::cout << x << '\n';  // Output: 100
```

2. **Smart Pointers**:
   - Smart pointers automatically manage memory lifetime, preventing leaks and dangling pointers.
   - **`std::unique_ptr`**: Exclusive ownership.

```cpp
auto ptr = std::make_unique<MyClass>(args...);
ptr->method();         // Access member
(*ptr).method();       // Alternative syntax
```

   - **`std::shared_ptr`**: Shared ownership with reference counting.

```cpp
auto ptr1 = std::make_shared<MyClass>(args...);
auto ptr2 = ptr1;      // Shared ownership
ptr1->method();        // Both point to same object
```

   - **`std::weak_ptr`**: Non-owning observer that must be locked before dereferencing.

```cpp
std::weak_ptr<MyClass> weak = ptr1;
if (auto locked = weak.lock()) {  // Acquire shared_ptr if object exists
    locked->method();             // Safe dereference
}
```

3. **C++23: `std::out_ptr` and `std::inout_ptr`**:
   - These utilities enable safe dereferencing when interfacing with C APIs that expect raw pointer outputs.
   - **Example**:

```cpp
// C API that allocates memory
extern "C" void create_buffer(void** out, size_t* size);

std::vector<uint8_t> buffer;
size_t size = 0;
create_buffer(std::out_ptr(buffer), &size);  // C++23
// buffer now owns the allocated memory, automatically managed
```

4. **C++26 Preview: `std::observer_ptr`**:

   - Proposed non-owning pointer vocabulary type with explicit null checking.
   - **Example**:

```cpp
// Proposed C++26
void process(std::observer_ptr<const Widget> obs) {
    if (obs) {                    // Explicit null check
        obs->doWork();            // Safe dereference
    }
    // observer_ptr never deletes
}

// Usage
Widget widget;
process(std::observer_ptr(&widget));  // Clear non-owning intent
```

5. **Range-based For Loops and `std::span`**:

   - These constructs reduce the need for manual pointer dereferencing when iterating.
   - **Example**:

```cpp
std::vector<int> vec = {1, 2, 3, 4, 5};
for (int& elem : vec) {
    elem *= 2;        // No pointers, direct access
}

// With std::span (C++20)
void process(std::span<int> data) {
    for (int& elem : data) {
        elem *= 2;    // Safe, bounds-checked in debug
    }
}
```

2.3.4 Best Practices for Dereferencing Pointers

1. **Always Check for Null**:

```
int* ptr = getPointer();
if (ptr != nullptr) {
    *ptr = 42;  // Safe
}
```

2. **Prefer References Over Raw Pointers for Non-nullable Parameters**:

```
void process(int& value);  // Guaranteed non-null
void process(int* value);  // May be null, requires check
```

3. **Use Smart Pointers for Ownership**:

   - `std::unique_ptr` for exclusive ownership, `std::shared_ptr` for shared ownership. Reserve raw pointers for non-owning observation.

4. **Avoid Dangling Pointers**:

   - Never dereference a pointer after the pointed object has been destroyed. Smart pointers prevent this.

5. **Use `std::optional` for Optional Values**:

   - Instead of returning a raw pointer that may be null, consider `std::optional`.

6. **Use `std::span` for Array Views**:

   - Instead of passing raw pointer and size separately, use `std::span` (C++20).

2.3.5 Advanced Techniques and Use Cases

1. **Pointer to Pointer (Double Indirection)**:

   - Double pointers allow functions to modify pointer arguments.
   - **Example**:

```
void allocateAndInit(int** ptr, int value) {
    *ptr = new int(value);  // Modify the caller's pointer
}
```

```cpp
int* p = nullptr;
allocateAndInit(&p, 42);
std::cout << *p << '\n';  // 42
delete p;

// Modern alternative: reference to smart pointer
void allocateAndInit(std::unique_ptr<int>& ptr, int value) {
    ptr = std::make_unique<int>(value);
}
```

2. **Dereferencing with Pointer Arithmetic**:

   - Combine pointer arithmetic with dereferencing for efficient array operations.
   - **Example**:

```cpp
int arr[] = {10, 20, 30, 40, 50};
int* ptr = arr;
for (size_t i = 0; i < 5; ++i) {
    std::cout << *(ptr + i) << ' ';  // Dereference after arithmetic
}
// Output: 10 20 30 40 50
```

3. **Function Pointers**:

   - Dereferencing function pointers allows dynamic function calls.
   - **Example**:

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

int (*op)(int, int) = add;
std::cout << op(5, 3) << '\n';  // 8 (dereferencing implicit)

// Explicit dereferencing
std::cout << (*op)(5, 3) << '\n';  // Also 8

// Modern alternative: std::function
std::function<int(int,int)> func = multiply;
std::cout << func(5, 3) << '\n';  // 15
```

4. **Member Pointers**:

   - Pointers to class members require special dereferencing syntax.

- **Example**:

```
struct Point {
    int x, y;
};

int Point::*memberPtr = &Point::x;  // Pointer to member
Point p{10, 20};
std::cout << p.*memberPtr << '\n';  // 10 (dereference with .*)

Point* ptr = &p;
std::cout << ptr->*memberPtr << '\n';  // 10 (dereference with ->*)
```

5. **Placement New and Dereferencing**:

- Placement new constructs objects in pre-allocated memory; dereferencing accesses them.
- **Example**:

```
alignas(alignof(MyClass)) char buffer[sizeof(MyClass)];
MyClass* ptr = new (buffer) MyClass(args...);
ptr->method();          // Dereference to use object
ptr->~MyClass();        // Explicit destructor call
// No delete needed; memory was not allocated with new
```

2.3.6 Common Pitfalls and Undefined Behavior

1. **Dereferencing Null Pointer**:

```
int* ptr = nullptr;
*ptr = 42;  // UNDEFINED: program crash likely
```

2. **Dangling Pointer Dereference**:

```
int* ptr = new int(42);
delete ptr;
*ptr = 100;  // UNDEFINED: use after free
```

3. **Dereferencing Uninitialized Pointer**:

```
int* ptr;     // Uninitialized
*ptr = 42;    // UNDEFINED: indeterminate address
```

4. **Dereferencing Past Array Bounds**:

```cpp
int arr[5] = {1, 2, 3, 4, 5};
int* ptr = arr + 10;  // Far beyond bounds
*ptr = 42;             // UNDEFINED
```

2.3.7 Real-World Applications

1. **Dynamic Memory Management**:
   - Dereferencing is essential for manipulating dynamically allocated objects in custom containers, linked lists, trees, and graphs.
2. **Polymorphism**:
   - Base class pointers are dereferenced to call virtual functions, enabling runtime polymorphism.
3. **Low-Level System Programming**:
   - Hardware register access uses dereferenced pointers to read/write memory-mapped I/O.
4. **Callback Mechanisms**:
   - Function pointers are dereferenced to implement callbacks and event handlers.
5. **Interfacing with C Libraries**:
   - C APIs frequently use raw pointers; dereferencing is required to access data.

2.3.8 Summary

- Dereferencing with $\star$ accesses or modifies the value at a pointer's address.
- Always check for null before dereferencing raw pointers to avoid undefined behavior.
- References provide safer, non-nullable alternatives for aliasing.
- Smart pointers (`unique_ptr`, `shared_ptr`) manage ownership and can be dereferenced like raw pointers.
- C++23 introduces `out_ptr` utilities for safe C API integration.
- C++26 will add `observer_ptr` for clear non-owning pointer semantics.

- Best practices: prefer references, use smart pointers for ownership, check for null, and leverage modern vocabulary types like `std::span` and `std::optional`.

Exercise 6. ***Practice Exercises:***

1. *Write a function that takes a pointer to an integer and doubles its value. Demonstrate with both raw pointer and smart pointer versions.*
2. *Implement a generic swap function that works with any type using references. Compare with a pointer-based version.*
3. *Create a small linked list where each node contains a `std::shared_ptr` to the next node. Demonstrate traversal and dereferencing.*
4. *Using C++23's `std::out_ptr`, write a wrapper for a C API that returns a dynamically allocated string.*
5. *Research `std::observer_ptr` (C++26 draft) and write an example showing how it differs from raw pointers for non-owning observation.*
6. *Write a function that takes a `std::span<int>` and computes the sum of all elements. Compare with a version using raw pointers.*

2.4 Pointers and Arrays

Pointers and arrays are closely related in C++. Understanding this relationship is crucial for efficient memory management and data manipulation. In this section, we will explore the **relationship between pointers and arrays**, provide **examples of accessing array elements using pointers**, discuss **multi-dimensional arrays and pointers**, and demonstrate **traversing arrays with modern C++ alternatives**. We will also incorporate modern C++ concepts, such as `std::array`, `std::vector`, `std::span`, `std::mdspan`, and smart pointers, to provide a comprehensive understanding of pointers and arrays in contemporary C++ programming.

2.4.1 Relationship Between Pointers and Arrays

In C++, the name of an array is essentially a pointer to its first element. This relationship allows you to use pointer arithmetic to access and manipulate array elements. However, there are important distinctions to understand.

Important

Key Distinction: While an array name decays to a pointer in most contexts, an array is **not** a pointer. The `sizeof` operator reveals this difference:

```
int arr[5];
int* ptr = arr;
std::cout << sizeof(arr) << '\n';  // 5 * sizeof(int) = 20 (on typical
↪  systems)
std::cout << sizeof(ptr) << '\n';  // sizeof(int*) = 8 (on 64-bit systems)
```

1. **Array Name as a Pointer**:
   - The name of an array is a constant pointer to the first element.
   - **Example**:

     ```
     int arr[5] = {10, 20, 30, 40, 50};
     int* ptr = arr;           // ptr points to arr[0]
     std::cout << *ptr << '\n'; // Output: 10
     std::cout << arr[0] << '\n'; // Also 10
     ```

2. **Pointer Arithmetic with Arrays**:
   - Adding an integer to a pointer moves it to the corresponding element.

- **Example**:

```
int arr[5] = {10, 20, 30, 40, 50};
int* ptr = arr;
std::cout << *(ptr + 2) << '\n';  // Output: 30 (arr[2])
```

3. **Array Indexing Equivalence**:

- Array indexing `arr[i]` is equivalent to pointer arithmetic `*(arr + i)`.
- **Example**:

```
int arr[5] = {10, 20, 30, 40, 50};
std::cout << arr[2] << '\n';    // 30
std::cout << *(arr + 2) << '\n'; // 30 (identical)
std::cout << 2[arr] << '\n';     // 30 (valid but confusing!)
```

2.4.2 Accessing Array Elements Using Pointers

Below are examples of accessing array elements using pointers and pointer arithmetic.

1. **Traditional Pointer Traversal**:

```
#include <iostream>

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int* ptr = arr;

    for (int i = 0; i < 5; ++i) {
        std::cout << "Element " << i << ": " << *(ptr + i) << '\n';
    }
    return 0;
}
```

2. **Using Pointer with Bounds**:

```
int arr[5] = {10, 20, 30, 40, 50};
int* begin = arr;
int* end = arr + 5;  // One past the last element

for (int* ptr = begin; ptr != end; ++ptr) {
    std::cout << *ptr << ' ';
}
```

2.4.3 Multi-dimensional Arrays and Pointers

Multi-dimensional arrays are arrays of arrays. Understanding pointer relationships with multi-dimensional arrays is essential for advanced C++ programming.

1. **2D Array Memory Layout**:
   - A 2D array is stored in row-major order (all elements of row 0, then row 1, etc.).
   - **Example**:

```
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

// Pointer to first element
int* flat = &matrix[0][0];
std::cout << flat[1 * 4 + 2] << '\n';  // Access row 1, col 2 = 7
```

2. **Pointer to a Row**:
   - A 2D array name is a pointer to the first row (array of columns).
   - **Example**:

```
int matrix[3][4];
int (*rowPtr)[4] = matrix;     // Pointer to an array of 4 ints
rowPtr[1][2] = 42;              // Access using row pointer
```

3. **Array of Pointers (Jagged Arrays)**:
   - For variable-length rows, use an array of pointers.
   - **Example**:

```
int row0[] = {1, 2, 3};
int row1[] = {4, 5};
int row2[] = {6, 7, 8, 9};

int* jagged[] = {row0, row1, row2};
int rows = 3;

for (int i = 0; i < rows; ++i) {
    // Need to track sizes separately
```

```
        std::cout << "Row " << i << ": ";
        for (int j = 0; j < (i == 0 ? 3 : (i == 1 ? 2 : 4)); ++j) {
            std::cout << jagged[i][j] << ' ';
        }
        std::cout << '\n';
    }
```

2.4.4 Example: Traversing a 2D Array with Pointers

Below is an example of traversing a 2D array using pointer arithmetic.

```
#include <iostream>

int main() {
    constexpr int rows = 3, cols = 4;
    int matrix[rows][cols] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };

    // Method 1: Flattened pointer arithmetic
    int* flat = &matrix[0][0];
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            std::cout << *(flat + i * cols + j) << ' ';
        }
        std::cout << '\n';
    }

    // Method 2: Row pointer
    int (*rowPtr)[cols] = matrix;
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            std::cout << rowPtr[i][j] << ' ';
        }
        std::cout << '\n';
    }

    return 0;
}
```

2.4.5 Modern C++ Alternatives for Arrays

Modern C++ provides safer, more expressive alternatives to raw arrays and pointer arithmetic.

1. **`std::array` (Fixed-size Arrays)**:
    - Provides array semantics with STL container interface and bounds checking in debug builds.
    - **Example**:

```cpp
#include <array>

std::array<int, 5> arr = {10, 20, 30, 40, 50};

// Access via iterators
for (auto it = arr.begin(); it != arr.end(); ++it) {
    std::cout << *it << ' ';
}

// Range-based for loop
for (int& elem : arr) {
    elem *= 2;
}

// Bounds-safe access (with at() throwing on error)
try {
    arr.at(10) = 100;  // Throws std::out_of_range
} catch (const std::out_of_range& e) {
    std::cout << "Out of bounds: " << e.what() << '\n';
}
```

2. **`std::vector` (Dynamic Arrays)**:
    - Dynamically resizable array with automatic memory management.
    - **Example**:

```cpp
#include <vector>

std::vector<int> vec = {1, 2, 3, 4, 5};
vec.push_back(6);  // Automatically grows

// Access underlying data for C API interop
```

```cpp
int* data = vec.data();  // Pointer to first element
size_t size = vec.size();

// Range-based iteration
for (int& elem : vec) {
    elem *= 2;
}
```

3. **Smart Pointers for Dynamic Arrays**:

   - `std::unique_ptr<T[]>` for exclusive ownership of dynamic arrays.
   - **Example**:

```cpp
auto arr = std::make_unique<int[]>(10);  // C++14
arr[0] = 42;  // operator[] works with unique_ptr<T[]>
arr[1] = 84;

// Cannot copy, only move
auto moved = std::move(arr);
// Automatically deleted when moved goes out of scope

// C++23: make_unique_for_overwrite for performance
auto buffer = std::make_unique_for_overwrite<uint8_t[]>(1024);
// Elements are uninitialized - fill before reading!
std::fill_n(buffer.get(), 1024, uint8_t(0));
```

4. **`std::span` (C++20) - Non-owning Array Views**:

   - Provides a safe, bounds-checked view of contiguous memory.
   - **Example**:

```cpp
#include <span>

void process(std::span<int> data) {
    // Safe iteration with size information
    for (int& elem : data) {
        elem *= 2;
    }

    // Bounds-safe access (with assertions in debug)
    data[0] = 100;
}
```

```cpp
int arr[] = {1, 2, 3, 4, 5};
std::vector<int> vec = {1, 2, 3, 4, 5};

process(arr);  // Works with raw arrays
process(vec);  // Works with vectors
process(std::span<int>(arr + 1, 3));  // Subrange {2, 3, 4}
```

5. **`std::mdspan` (C++23) - Multi-dimensional Views**:

   - Provides a safe, non-owning view for multi-dimensional arrays without manual pointer arithmetic.
   - **Example**:

```cpp
#include <mdspan>
#include <vector>

int main() {
    std::vector<int> data(12);
    for (int i = 0; i < 12; ++i) data[i] = i + 1;

    // Create 3x4 view of the 1D data
    std::mdspan<int, std::extents<size_t, 3, 4>> matrix(data.data());

    // Access with 2D indices - no pointer arithmetic needed
    for (size_t i = 0; i < matrix.extent(0); ++i) {
        for (size_t j = 0; j < matrix.extent(1); ++j) {
            std::cout << matrix[i, j] << ' ';
        }
        std::cout << '\n';
    }

    // Subview (first two rows)
    auto sub = std::submdspan(matrix, std::tuple{0, 2},
    ↪  std::full_extent);

    return 0;
}
```

2.4.6 Best Practices for Pointers and Arrays

1. **Prefer Standard Library Containers**:

- Use `std::array` for fixed-size arrays, `std::vector` for dynamic arrays. They provide automatic memory management, bounds checking (in debug), and STL algorithm compatibility.

2. **Use `std::span` for Array Views**:
   - Instead of passing raw pointer and size separately, use `std::span` (C++20) for function parameters.
3. **Use `std::mdspan` for Multi-dimensional Access**:
   - For multi-dimensional arrays, `std::mdspan` (C++23) eliminates manual pointer arithmetic.
4. **Smart Pointers for Dynamic Arrays**:
   - Use `std::unique_ptr<T[]>` or `std::shared_ptr<T[]>` instead of raw `new[]`/`delete[]`.
5. **Avoid Out-of-Bounds Access**:
   - Always ensure pointer arithmetic stays within bounds. Use containers with bounds-checked access (`.at()`) in debug builds.
6. **Use Range-based For Loops**:
   - Prefer range-based for loops over manual pointer arithmetic for iteration.
7. **C++26 Preview: `std::ptr_len`**:
   - The proposed `std::ptr_len` provides a standard vocabulary type for pointer-length pairs, replacing many ad-hoc patterns.
   - **Example**:

```cpp
// Proposed C++26
void process(std::ptr_len<const int> buffer) {
    for (std::size_t i = 0; i < buffer.len(); ++i) {
        std::cout << buffer.ptr()[i] << ' ';
    }
}
```

2.4.7 Advanced Techniques

1. **Dynamic 2D Arrays with `std::vector`**:

```
size_t rows = 3, cols = 4;

// Option 1: Vector of vectors (simpler, may have cache issues)
std::vector<std::vector<int>> matrix(rows, std::vector<int>(cols, 0));
matrix[1][2] = 42;

// Option 2: Flat vector with 2D indexing (better cache locality)
std::vector<int> flat(rows * cols, 0);
auto idx = [cols](size_t i, size_t j) { return i * cols + j; };
flat[idx(1, 2)] = 42;

// Option 3: Using std::mdspan (C++23) with flat storage
std::mdspan<int, std::extents<size_t, std::dynamic_extent,
↪  std::dynamic_extent>>
    view(flat.data(), rows, cols);
view[1, 2] = 42;
```

2. **Pointer Arithmetic with Custom Allocators**:

```
class ArenaAllocator {
    std::vector<std::byte> storage;
    size_t offset{0};

public:
    explicit ArenaAllocator(size_t size) : storage(size) {}

    void* allocate(size_t size, size_t alignment = alignof(std::max_align_t))
    ↪  {
        size_t alignedOffset = (offset + alignment - 1) & ~(alignment - 1);
        if (alignedOffset + size > storage.size()) return nullptr;
        void* result = storage.data() + alignedOffset;
        offset = alignedOffset + size;
        return result;
    }

    void reset() { offset = 0; }
};

// Use with placement new
```

```
ArenaAllocator arena(1024);
int* arr = static_cast<int*>(arena.allocate(5 * sizeof(int)));
for (int i = 0; i < 5; ++i) {
    new (arr + i) int(i * 10);  // Placement new with pointer arithmetic
}
```

2.4.8 Common Pitfalls

1. **Array-to-Pointer Decay in Functions**:

```
void func(int arr[5]) {  // Actually decays to int* arr
    std::cout << sizeof(arr) << '\n';  // sizeof(int*), not 5*sizeof(int)
}
```

Solution: Use `std::array` or pass size explicitly, or use `std::span`.

2. **Out-of-Bounds Access**:

```
int arr[5];
int* ptr = arr + 10;  // Far beyond bounds
*ptr = 42;            // UNDEFINED
```

Solution: Use `std::array::at()` or `std::span` with bounds checking.

3. **Returning Pointer to Local Array**:

```
int* bad() {
    int arr[5] = {1, 2, 3, 4, 5};
    return arr;  // Dangling pointer!
}
```

Solution: Return `std::array`, `std::vector`, or `std::unique_ptr`.

4. **Using `delete` Instead of `delete[]`**:

```
int* arr = new int[10];
delete arr;  // UNDEFINED: mismatched new/delete
```

Solution: Use `delete[]` or better, use `std::unique_ptr<int[]>`.

2.4.9 Summary

- Array names decay to pointers to their first element in most contexts.
- Pointer arithmetic (`ptr + i`) provides efficient array traversal.
- Multi-dimensional arrays are stored in row-major order; flattened indexing with `flat[i * cols + j]` enables pointer-based access.
- Modern C++ alternatives: `std::array` (fixed), `std::vector` (dynamic), `std::span` (views), and `std::mdspan` (multi-dimensional views, C++23).
- Smart pointers (`unique_ptr<T[]>`) provide safe ownership of dynamic arrays.
- C++26 will introduce `std::ptr_len` for standardized pointer-length pairs.
- Best practices: prefer containers over raw arrays, use `std::span` for function parameters, and leverage range-based for loops.

Exercise 7. ***Practice Exercises:***

1. *Given a 1D array, write a function that reverses it using pointer arithmetic (no indexing).*
2. *Create a 3x4 matrix using `std::vector<int>` as flat storage. Write functions to access and modify elements using both flat indexing and `std::mdspan` (C++23).*
3. *Implement a function that takes a `std::span<int>` and returns the sum of all elements. Compare with a version using raw pointers.*
4. *Using `std::unique_ptr<int[]>`, dynamically allocate an array of 100 integers, fill it with values 0-99, and print them.*
5. *Research `std::mdspan` (C++23) and create a 2D view of a 1D vector. Demonstrate slicing to get a submatrix.*
6. *Write a function that accepts a `std::array<int, 10>` and returns a `std::array<int, 10>` with each element doubled. Compare with a raw array version.*

Chapter 3

Advanced Pointers in C++

3.1 Advanced Pointer Techniques in Functions

Pointers play a crucial role in C++ functions, enabling powerful features such as modifying function arguments, returning dynamically allocated memory, and implementing callbacks. In this section, we will explore **passing pointers as function arguments**, **returning pointers from functions**, and **function pointers**. We will also incorporate modern C++ concepts, such as smart pointers, `std::function`, lambda expressions, and C++23/26 enhancements, to provide a comprehensive understanding of advanced pointer techniques in contemporary C++ programming.

3.1.1 Passing Pointers as Function Arguments

Passing pointers as function arguments allows you to modify the original data outside the function. This is particularly useful when you need to update variables or work with large data structures without copying them.

1. **Modifying Function Arguments Using Pointers**:
    - By passing a pointer to a variable, you can modify the original variable within the function.
    - **Example**:

```
void increment(int* ptr) {
```

```
    if (ptr) {  // Always check for null
        (*ptr)++;  // Increment the value pointed to by ptr
    }
}

int main() {
    int x = 10;
    increment(&x);
    std::cout << "x after increment: " << x << '\n';  // Output: 11
    return 0;
}
```

2. **Passing Arrays to Functions**:

   - Arrays decay to pointers to their first element when passed to functions.
   - **Example**:

```
void printArray(const int* arr, size_t size) {
    for (size_t i = 0; i < size; ++i) {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    printArray(arr, 5);
    return 0;
}
```

3. **Modern C++: Passing with `std::span`**:

   - C++20 introduces `std::span` for safer array passing.
   - **Example**:

```
#include <span>

void printSpan(std::span<const int> data) {
    for (int value : data) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

```cpp
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    std::vector<int> vec = {1, 2, 3, 4, 5};

    printSpan(arr);  // Works with raw array
    printSpan(vec);  // Works with vector
    printSpan(std::span<const int>(arr + 1, 3));  // Subrange {2, 3, 4}
    return 0;
}
```

3.1.2 Returning Pointers from Functions

Returning pointers from functions allows you to return dynamically allocated memory or the address of a variable. However, care must be taken to avoid returning pointers to local variables.

Important

Critical Rule: Never return a pointer to a local variable. The local variable is destroyed when the function returns, leaving a dangling pointer.

1. **Returning a Dynamically Allocated Array (Traditional)**:
   - Returns a raw pointer; caller is responsible for deallocation.
   - **Example**:

```cpp
int* createArray(size_t size) {
    int* arr = new int[size];  // Dynamically allocate
    for (size_t i = 0; i < size; ++i) {
        arr[i] = static_cast<int>(i + 1);
    }
    return arr;  // Caller must delete[]
}

int main() {
    int* arr = createArray(5);
    for (size_t i = 0; i < 5; ++i) {
        std::cout << arr[i] << ' ';
    }
    delete[] arr;  // Caller responsible!
    return 0;
}
```

2. **Returning Smart Pointers (Modern C++ Preferred)**:

   - Smart pointers automatically manage memory, preventing leaks.
   - **Example with `std::unique_ptr`**:

```cpp
#include <memory>

std::unique_ptr<int[]> createArray(size_t size) {
    auto arr = std::make_unique<int[]>(size);
    for (size_t i = 0; i < size; ++i) {
        arr[i] = static_cast<int>(i + 1);
    }
    return arr;  // Ownership transferred to caller
}

int main() {
    auto arr = createArray(5);  // No manual delete needed
    for (size_t i = 0; i < 5; ++i) {
        std::cout << arr[i] << ' ';
    }
    return 0;  // arr automatically destroyed
}
```

   - **C++23: `std::make_unique_for_overwrite`** for performance:

```cpp
auto arr = std::make_unique_for_overwrite<int[]>(1000);
// Elements are uninitialized - faster for large arrays
std::fill_n(arr.get(), 1000, 42);
```

3. **Returning `std::vector` (Simplest for Dynamic Arrays)**:

   - `std::vector` provides automatic memory management and STL compatibility.
   - **Example**:

```cpp
#include <vector>

std::vector<int> createVector(size_t size) {
    std::vector<int> vec(size);
    for (size_t i = 0; i < size; ++i) {
        vec[i] = static_cast<int>(i + 1);
    }
    return vec;  // Efficient move semantics
}
```

```cpp
int main() {
    auto vec = createVector(5);
    for (int value : vec) {
        std::cout << value << ' ';
    }
    return 0;
}
```

3.1.3 Function Pointers

Function pointers allow you to store and call functions dynamically. They are useful for implementing callbacks, strategy patterns, and other advanced programming techniques.

1. **Declaring and Using Function Pointers**:
   - Syntax: `return_type (*pointer_name)(parameter_types).`
   - **Example**:

```cpp
void greet() { std::cout << "Hello!\n"; }
void farewell() { std::cout << "Goodbye!\n"; }

int main() {
    void (*funcPtr)() = greet;  // funcPtr points to greet
    funcPtr();                  // Calls greet()
    funcPtr = farewell;         // Reassign
    funcPtr();                  // Calls farewell()
    return 0;
}
```

2. **Implementing a Callback Mechanism**:
   - Function pointers enable flexible callbacks.
   - **Example**:

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

void process(int x, int y, int (*callback)(int, int)) {
    int result = callback(x, y);
    std::cout << "Result: " << result << '\n';
}
```

```
int main() {
    process(10, 20, add);       // Result: 30
    process(10, 20, multiply);  // Result: 200
    return 0;
}
```

3.1.4 Modern C++ Alternatives for Callbacks

Modern C++ provides safer, more flexible alternatives to raw function pointers.

1. **Lambda Expressions**:
   - Lambda expressions create anonymous functions with capture capabilities.
   - **Example**:

```
void process(int x, int y, int (*callback)(int, int)) {
    std::cout << "Result: " << callback(x, y) << '\n';
}

int main() {
    auto add = [](int a, int b) { return a + b; };
    process(10, 20, add);  // Lambda converts to function pointer (if
    ↪  stateless)

    // Capturing lambda cannot convert to function pointer
    int factor = 2;
    auto multiply = [factor](int a, int b) { return (a + b) * factor; };
    // process(10, 20, multiply);  // ERROR: capturing lambda
    return 0;
}
```

2. **`std::function` (Type-Erased Callable)**:
   - `std::function` can store any callable: function pointers, lambdas, function objects, and member functions.
   - **Example**:

```
#include <functional>

void process(int x, int y, std::function<int(int, int)> callback) {
    std::cout << "Result: " << callback(x, y) << '\n';
}
```

```
int main() {
    int factor = 2;
    auto multiply = [factor](int a, int b) { return (a + b) * factor; };

    process(10, 20, multiply);  // Works with capturing lambda
    process(10, 20, std::plus<int>{});  // Works with function object
    process(10, 20, [](int a, int b) { return a * b; });  // Stateless
    ↪  lambda
    return 0;
}
```

3. **`std::invoke` (C++17) for Unified Call Syntax**:

   - `std::invoke` provides a unified way to call any callable.
   - **Example**:

```
#include <functional>

template<typename Callable, typename... Args>
auto unifiedCall(Callable&& callable, Args&&... args) {
    return std::invoke(std::forward<Callable>(callable),
                       std::forward<Args>(args)...);
}

int main() {
    auto result = unifiedCall([](int a, int b) { return a + b; }, 10,
    ↪  20);
    std::cout << result << '\n';  // 30
    return 0;
}
```

4. **C++23: `std::move_only_function`**:

   - C++23 introduces `std::move_only_function` for move-only callables.
   - **Example**:

```
#include <functional>

std::move_only_function<int(int, int)> getCallback() {
    return [](int a, int b) { return a * b; };  // Move-only
}
```

```cpp
int main() {
    auto callback = getCallback();
    std::cout << callback(10, 20) << '\n';  // 200
    return 0;
}
```

3.1.5 Advanced Function Pointer Techniques

1. **Pointer to Pointer (Double Indirection)**:
    - Allows functions to modify pointer arguments.
    - **Example**:

```cpp
void allocateMemory(int** ptr, size_t size) {
    *ptr = new int[size];  // Modify caller's pointer
    for (size_t i = 0; i < size; ++i) {
        (*ptr)[i] = static_cast<int>(i + 1);
    }
}

int main() {
    int* arr = nullptr;
    allocateMemory(&arr, 5);
    for (size_t i = 0; i < 5; ++i) {
        std::cout << arr[i] << ' ';
    }
    delete[] arr;
    return 0;
}
```

    - **Modern alternative with reference to smart pointer**:

```cpp
void allocateMemory(std::unique_ptr<int[]>& ptr, size_t size) {
    ptr = std::make_unique<int[]>(size);
    for (size_t i = 0; i < size; ++i) {
        ptr[i] = static_cast<int>(i + 1);
    }
}
```

2. **Function Pointers in Data Structures**:
    - Store function pointers in containers for dynamic dispatch.
    - **Example**:

```cpp
#include <vector>

void greet() { std::cout << "Hello!\n"; }
void farewell() { std::cout << "Goodbye!\n"; }

int main() {
    std::vector<void(*)()> functions = {greet, farewell};
    for (auto func : functions) {
        func();  // Call each function
    }
    return 0;
}
```

3. **Member Function Pointers**:

   - Pointers to member functions require special syntax.
   - **Example**:

```cpp
class Calculator {
public:
    int add(int a, int b) const { return a + b; }
    int multiply(int a, int b) const { return a * b; }
};

int main() {
    Calculator calc;
    // Pointer to member function
    int (Calculator::*memberPtr)(int, int) const = &Calculator::add;

    int result = (calc.*memberPtr)(10, 20);  // Call via object
    std::cout << result << '\n';  // 30

    // Using std::invoke (C++17)
    result = std::invoke(memberPtr, calc, 10, 20);
    std::cout << result << '\n';  // 30
    return 0;
}
```

3.1.6 C++26 Preview: `std::observer_ptr` and Function Parameters

The proposed C++26 standard introduces `std::observer_ptr` for clear non-owning pointer semantics, which is particularly useful for function parameters.

```cpp
// Proposed C++26
#include <observer_ptr>

void process(std::observer_ptr<const int> data) {
    if (data) {  // Explicit null check
        std::cout << *data << '\n';
    }
    // observer_ptr never deletes
}

int main() {
    int value = 42;
    process(std::observer_ptr(&value));  // Clear non-owning intent

    std::observer_ptr<const int> nullPtr(nullptr);
    process(nullPtr);  // Safe - null check inside function
    return 0;
}
```

3.1.7 Best Practices for Advanced Pointer Techniques

1. **Prefer Smart Pointers for Ownership**:
    - Use `std::unique_ptr` for exclusive ownership, `std::shared_ptr` for shared ownership. Reserve raw pointers for non-owning observation.
2. **Use `std::span` for Array Parameters**:
    - Instead of pointer + size, use `std::span` (C++20) for safer array passing.
3. **Use `std::function` or Templates for Callbacks**:
    - `std::function` provides type erasure for flexibility. Templates provide zero-overhead compile-time polymorphism.
4. **Prefer Lambda Expressions for Short Callbacks**:
    - Lambdas are concise and can capture context.
5. **Avoid Returning Raw Pointers to Dynamically Allocated Memory**:
    - Return smart pointers or containers instead to prevent memory leaks.

6. **Always Check for Null Before Dereferencing**:
    - Raw pointers can be null. Always validate before use.
7. **Use `std::invoke` for Generic Callable Code**:
    - C++17's `std::invoke` handles function pointers, member pointers, and callable objects uniformly.

3.1.8 Common Pitfalls

1. **Returning Pointer to Local Variable**:

```
int* bad() {
    int x = 42;
    return &x;  // Dangling pointer!
}
```

2. **Dangling Pointer After Deletion**:

```
int* ptr = new int(42);
delete ptr;
*ptr = 100;  // Undefined behavior!
```

3. **Mismatched `new`/`delete`**:

```
int* arr = new int[10];
delete arr;  // Undefined: should be delete[]
```

4. **Null Pointer Dereference**:

```
int* ptr = nullptr;
*ptr = 42;  // Crash!
```

3.1.9 Summary

- Pointers as function arguments enable modification of original data and efficient array passing.
- Return smart pointers (`unique_ptr`, `shared_ptr`) or containers instead of raw pointers for dynamic memory.
- Function pointers enable dynamic dispatch and callbacks but have limitations with capturing lambdas.

- Modern alternatives: `std::function` (type-erased), lambda expressions (capturing), `std::invoke` (unified call), and `std::move_only_function` (C++23).
- C++20's `std::span` provides safe array views; C++26's `std::observer_ptr` will clarify non-owning semantics.
- Best practices: prefer smart pointers for ownership, use `std::span` for array parameters, and always validate raw pointers before dereferencing.

Exercise 8. ***Practice Exercises:***

1. *Write a function that takes a pointer to an integer and a size, and fills the array with values from 0 to size-1. Use both raw pointer and* `std::span` *versions.*
2. *Implement a function that returns a* `std::unique_ptr<int[]>` *containing a Fibonacci sequence of given length.*
3. *Create a callback system using* `std::function` *that can store and execute multiple callbacks with different signatures.*
4. *Write a generic function using* `std::invoke` *that can call any callable with any arguments.*
5. *Research* `std::move_only_function` *(C++23) and write an example demonstrating its use with move-only types.*
6. *Implement a simple command pattern using function pointers stored in a* `std::map` *where keys are strings and values are callbacks.*

3.2 Pointers and Structures/Classes

Pointers are not only used with primitive data types but also with user-defined types like structures and classes. In this section, we will explore **pointers to structures and classes**, **accessing class members via pointers**, and the **`this` pointer**. We will also incorporate modern C++ concepts, such as smart pointers, member initializer lists, and C++23/26 enhancements, to provide a comprehensive understanding of pointers in the context of structures and classes.

3.2.1 Pointers to Structures and Classes

Pointers to structures and classes allow you to dynamically allocate objects, pass them to functions, and manipulate their members. Below are the key aspects of using pointers with user-defined types.

1. **Declaring Pointers to Structures/Classes**:
   - A pointer to a structure or class is declared using `ClassName* ptr`.
   - **Example**:

```
class MyClass {
public:
    int data;
    void display() const {
        std::cout << "Data: " << data << '\n';
    }
};

int main() {
    MyClass obj;
    MyClass* ptr = &obj;       // ptr points to obj
    ptr->data = 42;            // Access member using ->
    ptr->display();            // Call member function
    return 0;
}
```

2. **Dynamic Allocation of Objects (Traditional)**:
   - Use `new` to allocate objects on the heap.
   - **Example**:

```
MyClass* ptr = new MyClass();  // Dynamically allocate
ptr->data = 100;
ptr->display();
delete ptr;                    // Must deallocate
```

3. **Smart Pointers for Automatic Memory Management (Modern C++)**:

   - Smart pointers automatically manage object lifetime.
   - **`std::unique_ptr`** for exclusive ownership:

```
auto ptr = std::make_unique<MyClass>();
ptr->data = 200;
ptr->display();  // Automatically destroyed when ptr goes out of scope
```

   - **`std::shared_ptr`** for shared ownership:

```
auto ptr1 = std::make_shared<MyClass>();
auto ptr2 = ptr1;  // Shared ownership
ptr1->data = 300;
ptr2->display();   // Same object
```

   - **C++23: `std::make_unique_for_overwrite`** for performance:

```
auto ptr = std::make_unique_for_overwrite<MyClass>();
// Object is default-initialized (not value-initialized) - faster
```

3.2.2 Accessing Class Members via Pointers

Accessing members through pointers uses the −> operator (arrow operator), which is syntactic sugar for dereferencing then member access.

1. **Arrow Operator vs. Dereference**:

```
MyClass obj;
MyClass* ptr = &obj;

// These are equivalent:
ptr->data = 42;
(*ptr).data = 42;   // Dereference then dot operator
```

2. **Accessing Members in Nested Structures**:

```
struct Address {
    std::string city;
    int zipCode;
```

```
};

struct Person {
    std::string name;
    Address* address;  // Pointer to nested structure
};

int main() {
    Address addr{"New York", 10001};
    Person person{"Alice", &addr};

    Person* personPtr = &person;
    // Access nested member through pointer chain
    std::cout << personPtr->address->city << '\n';  // "New York"
    return 0;
}
```

3.2.3 Example: Complete Class with Pointer Members

Below is a comprehensive example demonstrating pointers within classes, including dynamic allocation and smart pointer usage.

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

class Product {
    std::string name;
    double price;

public:
    Product(const std::string& n, double p) : name(n), price(p) {}

    void display() const {
        std::cout << "Product: " << name << ", Price: $" << price << '\n';
    }

    double getPrice() const { return price; }
};

class ShoppingCart {
    std::unique_ptr<Product> product;  // Exclusive ownership
```

```
    int quantity;

public:
    ShoppingCart(std::unique_ptr<Product> p, int q)
        : product(std::move(p)), quantity(q) {}

    double total() const {
        return product->getPrice() * quantity;
    }

    void display() const {
        product->display();
        std::cout << "Quantity: " << quantity << ", Total: $" << total() << '\n';
    }
};

int main() {
    auto laptop = std::make_unique<Product>("Laptop", 999.99);
    ShoppingCart cart(std::move(laptop), 2);
    cart.display();

    return 0;  // All memory automatically cleaned up
}
```

3.2.4 The `this` Pointer

The `this` pointer is a special pointer available in non-static member functions of a class. It points to the object for which the member function is called.

Important

Key Properties of `this`:

- `this` is an implicit parameter to all non-static member functions.
- Its type is `ClassName* const` (const pointer to non-const object) in non-const member functions.
- In const member functions, its type is `const ClassName* const`.
- `this` cannot be modified (it's a prvalue in modern C++).

1. **Disambiguating Member Names**:
   - Use `this` when parameter names shadow member names.
   - **Example**:

```
class Person {
    std::string name;
    int age;

public:
    Person(const std::string& name, int age) {
        this->name = name;  // Disambiguate
        this->age = age;
    }

    // Better: use member initializer list (no this needed)
    Person(const std::string& n, int a) : name(n), age(a) {}
};
```

2. **Returning the Current Object for Method Chaining (Fluent Interface)**:
   - Return `*this` to enable method chaining.
   - **Example**:

```
class Builder {
    std::string result;

public:
    Builder& append(const std::string& text) {
        result += text;
        return *this;  // Return reference to current object
    }

    Builder& addSpace() {
        result += ' ';
        return *this;
    }

    std::string build() const {
        return result;
    }
};
```

```cpp
int main() {
    std::string s = Builder{}
        .append("Hello")
        .addSpace()
        .append("World")
        .build();
    std::cout << s << '\n';  // "Hello World"
    return 0;
}
```

3. **Comparing `this` with Another Object**:

   - Useful in copy assignment operators to prevent self-assignment.
   - **Example**:

```cpp
class MyClass {
    int* data;

public:
    MyClass& operator=(const MyClass& other) {
        if (this != &other) {  // Check for self-assignment
            delete data;
            data = new int(*other.data);
        }
        return *this;
    }
};
```

3.2.5 Modern C++ Concepts in Pointers and Classes

Modern C++ introduces several features that enhance safety and expressiveness when using pointers with classes.

1. **Smart Pointers with Custom Deleters**:

   - Smart pointers can manage resources beyond memory using custom deleters.
   - **Example**:

```cpp
#include <cstdio>

struct FileHandle {
    std::FILE* file;
```

```
    explicit FileHandle(const char* filename, const char* mode)
        : file(std::fopen(filename, mode)) {}

    void close() {
        if (file) std::fclose(file);
        file = nullptr;
    }
};

// Custom deleter for std::unique_ptr
auto fileDeleter = [](FileHandle* fh) {
    fh->close();
    delete fh;
};

int main() {
    std::unique_ptr<FileHandle, decltype(fileDeleter)>
        fh(new FileHandle("data.txt", "r"), fileDeleter);

    if (fh->file) {
        // Use file
    }
    // Automatically closed when fh goes out of scope
    return 0;
}
```

2. **`std::weak_ptr` for Breaking Circular References**:

   - `std::weak_ptr` observes a `shared_ptr` without increasing reference count.
   - **Example**:

```
class Node {
public:
    int value;
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;  // Weak pointer prevents cycles

    Node(int v) : value(v) {}
};

int main() {
    auto node1 = std::make_shared<Node>(1);
```

```
    auto node2 = std::make_shared<Node>(2);

    node1->next = node2;
    node2->prev = node1;  // Weak pointer - no cycle

    // Safe traversal
    if (auto prev = node2->prev.lock()) {
        std::cout << prev->value << '\n';  // 1
    }
    return 0;
}
```

3. **CRTP (Curiously Recurring Template Pattern) with `this`**:

   - CRTP uses `this` for compile-time polymorphism.
   - **Example**:

```
template<typename Derived>
class Base {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
};

class Derived : public Base<Derived> {
public:
    void implementation() {
        std::cout << "Derived implementation\n";
    }
};
```

4. **C++23: `std::out_ptr` with Classes**:

   - Safely pass smart pointers to C APIs that return pointers to structures.
   - **Example**:

```
// C API that creates a resource
extern "C" {
    struct Resource;
    void create_resource(Resource** out);
    void destroy_resource(Resource*);
}
```

```
std::unique_ptr<Resource, decltype(&destroy_resource)>
    resource(nullptr, destroy_resource);
create_resource(std::out_ptr(resource));  // C++23
// resource now owns the allocated Resource
```

5. **C++26 Preview: `std::observer_ptr` for Class Pointers**:
    - `std::observer_ptr` provides clear non-owning semantics.
    - **Example**:

```
// Proposed C++26
class Widget {
public:
    void process(std::observer_ptr<const Data> data) {
        if (data) {
            // Use data safely
        }
    }
};

int main() {
    Data d;
    Widget w;
    w.process(std::observer_ptr(&d));  // Clear: non-owning
    return 0;
}
```

3.2.6 Best Practices for Pointers and Structures/Classes

1. **Prefer Smart Pointers for Ownership**:
    - Use `std::unique_ptr` for exclusive ownership, `std::shared_ptr` for shared ownership. Reserve raw pointers for non-owning observation.
2. **Use Member Initializer Lists**:
    - Initialize members directly in the constructor initializer list for better performance and clarity.
3. **Use `const` Correctly**:

- Mark member functions as `const` when they don't modify the object. This affects the `this` pointer type.

4. **Avoid Raw Pointers for Ownership**:
   - Minimize manual `new` and `delete`. Smart pointers prevent leaks and dangling pointers.
5. **Check for Null Before Dereferencing**:
   - Always verify raw pointers are not null before using `->` or `*`.
6. **Use `this` Explicitly When Needed**:
   - Use `this` to disambiguate member names and in method chaining. Otherwise, it's often implicit.
7. **Break Cycles with `std::weak_ptr`**:
   - When using `std::shared_ptr` in cyclic structures (e.g., trees, graphs), use `std::weak_ptr` to break cycles.

3.2.7 Advanced Techniques

1. **Pointer-to-Member (Member Pointers)**:
   - Pointers to class members allow indirect access to members.
   - **Example**:

```
struct Point {
    int x, y;
};

int main() {
    int Point::*memberPtr = &Point::x;  // Pointer to member
    Point p{10, 20};

    std::cout << p.*memberPtr << '\n';  // 10 (dereference with .*)

    Point* ptr = &p;
    std::cout << ptr->*memberPtr << '\n';  // 10 (dereference with ->*)
    return 0;
}
```

2. **Pointer-to-Member Function**:

   - Store and call member functions indirectly.
   - **Example**:

```
class Calculator {
public:
    int add(int a, int b) const { return a + b; }
    int multiply(int a, int b) const { return a * b; }
};

int main() {
    Calculator calc;
    // Pointer to member function
    int (Calculator::*func)(int, int) const = &Calculator::add;

    int result = (calc.*func)(10, 20);  // 30
    std::cout << result << '\n';

    // Using std::invoke (C++17)
    result = std::invoke(func, calc, 10, 20);
    std::cout << result << '\n';
    return 0;
}
```

3. **Placement New with Classes**:

   - Construct objects in pre-allocated memory.
   - **Example**:

```
#include <new>
#include <memory>

class MyClass {
    std::string name;
public:
    MyClass(const std::string& n) : name(n) {
        std::cout << "Constructed: " << name << '\n';
    }
    ~MyClass() {
        std::cout << "Destroyed: " << name << '\n';
    }
};
```

```
int main() {
    alignas(MyClass) char buffer[sizeof(MyClass)];

    // Construct in buffer
    MyClass* ptr = new (buffer) MyClass("Placement");

    // Use object
    // ...

    // Explicit destructor call
    ptr->~MyClass();

    // No delete needed - memory not allocated with new
    return 0;
}
```

3.2.8 Common Pitfalls

1. **Dangling Pointer to Destroyed Object**:

```
MyClass* ptr = new MyClass();
delete ptr;
ptr->display();  // UNDEFINED: dangling pointer
```

2. **Circular References with `shared_ptr`**:

```
struct Node {
    std::shared_ptr<Node> next;
    // Memory leak if circular: use weak_ptr instead
};
```

3. **Returning `this` from Const Member Function Incorrectly**:

```
class MyClass {
public:
    MyClass* getThis() const {
        return this;  // ERROR: returning const MyClass* as MyClass*
    }

    const MyClass* getThis() const {
        return this;  // Correct
    }
```

```
};
```

4. **Forgetting Virtual Destructor in Base Classes**:

```
class Base {
public:
    ~Base() {}  // Non-virtual: derived destructor not called!
};

class Derived : public Base {
    int* data;
public:
    Derived() : data(new int(42)) {}
    ~Derived() { delete data; }  // Never called via Base*
};

Base* ptr = new Derived();
delete ptr;  // Memory leak!
```

3.2.9 Summary

- Pointers to classes enable dynamic allocation, polymorphism, and efficient passing of objects.
- The arrow operator (`->`) provides convenient member access through pointers.
- Smart pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`) automate memory management and prevent leaks.
- The `this` pointer refers to the current object, useful for disambiguation, method chaining, and self-assignment checks.
- C++23 adds `std::out_ptr` for smart pointer/C API interoperability.
- C++26 will introduce `std::observer_ptr` for clear non-owning pointer semantics.
- Best practices: prefer smart pointers for ownership, use member initializer lists, break cycles with `weak_ptr`, and always declare virtual destructors in polymorphic base classes.

Exercise 9. ***Practice Exercises:***

1. *Create a `LinkedList` class using `std::unique_ptr` for nodes. Implement `insert`, `remove`, and `display` methods.*

2. *Implement a* `Builder` *class for constructing complex objects using method chaining (return* `*this`*).*

3. *Create a bidirectional linked list using* `std::shared_ptr` *for forward links and* `std::weak_ptr` *for backward links to prevent cycles.*

4. *Write a class with a member function that takes a* `std::observer_ptr` *parameter (simulate with raw pointer) and demonstrate safe usage.*

5. *Implement a simple reference-counted resource using* `std::shared_ptr` *and demonstrate proper cleanup.*

6. *Using C++23's* `std::out_ptr`*, write a wrapper for a C API that creates and destroys a custom structure.*

3.3 Dynamic Memory Management

Dynamic memory management is one of the most powerful yet challenging aspects of C++ programming. It allows developers to allocate and deallocate memory at runtime, enabling the creation of flexible and efficient programs. However, improper management of dynamic memory can lead to severe issues such as memory leaks, dangling pointers, and undefined behavior. In this section, we will explore dynamic memory management in depth, covering the use of `new` and `delete`, common pitfalls, and modern C++ techniques to manage memory safely and efficiently.

3.3.1 Using `new` and `delete`

Allocating and Deallocating Memory

In C++, dynamic memory allocation is performed using the `new` operator, and deallocation is done using the `delete` operator. These operators interact with the **heap**, a region of memory reserved for dynamic allocation. Unlike stack memory, which is automatically managed, heap memory requires explicit management by the programmer.

Important

Key Rule: Every `new` must be paired with exactly one `delete`, and every `new[]` must be paired with exactly one `delete[]`. Mismatched forms lead to undefined behavior.

Allocating Memory with `new`

The `new` operator allocates memory for a single object or an array and returns a pointer to the allocated memory.

```
int* ptr = new int;           // Allocates memory for a single integer
↪  (uninitialized)
int* ptrVal = new int(42);    // Allocates and initializes to 42
int* arr = new int[10];       // Allocates memory for an array of 10 integers
int* arrZero = new int[10](); // Value-initialized to zero
```

The `new` operator also calls constructors for user-defined types:

```
class MyClass {
public:
    MyClass() { std::cout << "Constructor called!\n"; }
    ~MyClass() { std::cout << "Destructor called!\n"; }
```

```
};

MyClass* obj = new MyClass();  // Allocates memory and calls constructor
// Output: Constructor called!
```

Deallocating Memory with `delete`

When dynamically allocated memory is no longer needed, it must be deallocated:

```
delete ptr;       // Deallocates a single object
delete[] arr;     // Deallocates an array
delete obj;       // Calls destructor, then deallocates
```

Example: Creating and Deleting Dynamic Arrays

```
#include <iostream>

int main() {
    // Allocate memory for an array of 5 integers
    int* arr = new int[5];

    // Populate the array
    for (int i = 0; i < 5; ++i) {
        arr[i] = i * 10;
    }

    // Print the array
    for (int i = 0; i < 5; ++i) {
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';

    // Deallocate the memory
    delete[] arr;

    return 0;
}
```

3.3.2 Common Pitfalls

Memory Leaks

A **memory leak** occurs when dynamically allocated memory is not deallocated, leading to gradual memory exhaustion.

```
void createMemoryLeak() {
    int* ptr = new int(42);
    // Forget to delete ptr - memory leak!
}
```

Dangling Pointers

A **dangling pointer** points to memory that has already been deallocated. Accessing it is undefined behavior.

```
int* createDanglingPointer() {
    int* ptr = new int(42);
    delete ptr;
    return ptr;  // Dangling pointer!
}
```

Double Deletion

Calling `delete` twice on the same pointer leads to undefined behavior (typically program crash).

```
int* ptr = new int(42);
delete ptr;
delete ptr;  // Undefined behavior!
```

Mismatched `new` and `delete`

```
int* arr = new int[10];
delete arr;   // Undefined: should be delete[]
```

3.3.3 Fixing Memory Leaks and Dangling Pointers with Modern C++

Modern C++ provides powerful tools to eliminate these issues.

Using `std::unique_ptr` for Exclusive Ownership

`std::unique_ptr` automatically deallocates memory when it goes out of scope. It cannot be copied, ensuring unique ownership.

```
#include <memory>

void noMemoryLeak() {
    auto ptr = std::make_unique<int>(42);  // Automatically deallocated
    // No manual delete needed
}  // ptr destroyed, memory freed
```

For arrays:

```
auto arr = std::make_unique<int[]>(10);  // C++14
arr[0] = 42;  // operator[] works with unique_ptr<T[]>
// Automatically deallocated
```

C++23: `std::make_unique_for_overwrite` for performance:

```
auto arr = std::make_unique_for_overwrite<int[]>(1000);
// Elements are default-initialized (uninitialized) - faster
std::fill_n(arr.get(), 1000, 42);
```

Using `std::shared_ptr` for Shared Ownership

`std::shared_ptr` allows multiple pointers to share ownership. Memory is deallocated when the last `shared_ptr` is destroyed.

```
auto ptr1 = std::make_shared<int>(42);
auto ptr2 = ptr1;  // Both share ownership
// Memory freed when both ptr1 and ptr2 go out of scope
```

Using `std::weak_ptr` to Break Circular References

Circular references with `shared_ptr` cause memory leaks. Use `weak_ptr` to break cycles.

```
class Node {
public:
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;  // Weak pointer prevents cycles
};

auto node1 = std::make_shared<Node>();
auto node2 = std::make_shared<Node>();

node1->next = node2;
node2->prev = node1;  // No cycle - safe
```

RAII (Resource Acquisition Is Initialization)

RAII ties resource lifetime to object lifetime. Resources are acquired in constructors and released in destructors.

```
class FileHandle {
    std::FILE* file;
public:
    FileHandle(const char* filename, const char* mode)
        : file(std::fopen(filename, mode)) {
```

```
        if (!file) throw std::runtime_error("Cannot open file");
    }

    ~FileHandle() {
        if (file) std::fclose(file);
    }

    // Non-copyable, movable
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;
    FileHandle(FileHandle&& other) noexcept
        : file(std::exchange(other.file, nullptr)) {}

    std::FILE* get() const { return file; }
};

void useFile() {
    FileHandle fh("data.txt", "r");  // File opened
    // Use file...
}  // File automatically closed, even if exception thrown
```

3.3.4 Modern C++ Techniques for Dynamic Memory Management

1. **Prefer Standard Containers Over Raw Arrays**:
   - `std::vector`, `std::array`, and `std::string` manage memory automatically.
   - **Example**:

     ```
     std::vector<int> vec = {1, 2, 3, 4, 5};  // No new/delete needed
     vec.push_back(6);  // Automatically grows
     ```

2. **C++20: `std::span` for Non-Owning Views**:
   - `std::span` provides a safe view of contiguous memory without ownership.
   - **Example**:

     ```
     #include <span>

     void process(std::span<int> data) {
         for (int& elem : data) {
             elem *= 2;
         }
     ```

```
}

std::vector<int> vec = {1, 2, 3, 4, 5};
process(vec);  // Safe, no ownership transfer
```

3. **C++23: `std::out_ptr` and `std::inout_ptr`**:
   - Safely interface with C APIs that return allocated memory.
   - **Example**:

```
// C API that allocates memory
extern "C" void create_buffer(void** out, size_t* size);

std::vector<uint8_t> buffer;
size_t size = 0;
create_buffer(std::out_ptr(buffer), &size);  // C++23
// buffer now owns the allocated memory
```

4. **Move Semantics for Efficient Ownership Transfer**:
   - Move semantics transfer resources without copying.
   - **Example**:

```
std::unique_ptr<int> createResource() {
    return std::make_unique<int>(42);  // RVO + move
}

auto resource = createResource();  // Efficient transfer
```

5. **Memory Sanitizers and Detection Tools**:
   - Use **AddressSanitizer** (`-fsanitize=address`) and **LeakSanitizer** during development.
   - **Valgrind** (Linux) and **Dr. Memory** (Windows) for comprehensive memory analysis.
   - **C++23: `std::stacktrace`** for debugging memory-related crashes.

3.3.5 C++26 Preview: Vocabulary Types for Memory Management

The proposed C++26 standard introduces several types that enhance memory safety:
`std::observer_ptr` for clear non-owning semantics:

```cpp
// Proposed C++26
void process(std::observer_ptr<const int> data) {
    if (data) {  // Explicit null check
        std::cout << *data << '\n';
    }
    // observer_ptr never deletes
}
```

`std::ptr_len` for pointer-length pairs:

```cpp
// Proposed C++26
void process_buffer(std::ptr_len<const std::byte> buffer) {
    for (std::size_t i = 0; i < buffer.len(); ++i) {
        buffer.ptr()[i];  // Safe bounded access
    }
}
```

3.3.6 Best Practices for Dynamic Memory Management

1. **Prefer Stack Allocation When Possible**:
    - Stack allocation is faster, automatically managed, and cache-friendly.
2. **Use Smart Pointers for Heap Allocation**:
    - `std::unique_ptr` for exclusive ownership, `std::shared_ptr` for shared ownership.
3. **Avoid Manual `new` and `delete`**:
    - Use `std::make_unique` and `std::make_shared` instead.
4. **Use Standard Containers**:
    - `std::vector`, `std::array`, and `std::string` eliminate manual memory management.
5. **Implement RAII for Custom Resources**:
    - Encapsulate resources (files, sockets, mutexes) in RAII classes.
6. **Use `std::span` for Non-Owning Views**:

- Replace raw pointer + size parameters with `std::span`.

7. **Enable Compiler Sanitizers in Development**:
   - Use `-fsanitize=address,undefined` to catch memory errors early.

3.3.7 Comparison: Raw Pointers vs. Smart Pointers vs. Containers

Feature	Raw Pointers	Smart Pointers	Containers
Memory Management	Manual	Automatic	Automatic
Ownership	Unclear	Explicit	Container-owned
Exception Safety	Risky	Strong	Strong
Overhead	Minimal	Minimal (unique) or small (shared)	Moderate
Flexibility	High	High	High
Safety	Low	High	High

3.3.8 Common Pitfalls Summary

1. **Memory Leak**:

```
int* p = new int(42);
// No delete
```

2. **Dangling Pointer**:

```
int* p = new int(42);
delete p;
*p = 10;  // Undefined
```

3. **Double Deletion**:

```
delete p;
delete p;  // Undefined
```

4. **Mismatched `new`/`delete`**:

```
int* p = new int[10];
delete p;  // Undefined
```

5. **Circular Reference with `shared_ptr`**:

```cpp
struct Node {
    std::shared_ptr<Node> next;
    std::shared_ptr<Node> prev;  // Cycle! Use weak_ptr
};
```

3.3.9 Real-World Applications

1. **Game Development**: Dynamic allocation for game objects, levels, and assets.
2. **Database Systems**: Memory pools and custom allocators for query processing.
3. **Embedded Systems**: Careful heap management with limited memory.
4. **High-Performance Computing**: Custom allocators for cache efficiency.
5. **Network Servers**: Connection pools and buffer management.

3.3.10 Summary

- Dynamic memory with `new` and `delete` gives flexibility but requires careful manual management.
- Common pitfalls: memory leaks, dangling pointers, double deletion, and mismatched `new`/`delete`.
- Modern C++ solutions: smart pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`), RAII, and standard containers.
- C++20 adds `std::span` for non-owning views; C++23 adds `out_ptr` utilities and `make_unique_for_overwrite`.
- C++26 will introduce `observer_ptr` and `ptr_len` for clearer pointer semantics.
- Best practices: prefer stack allocation, use smart pointers for heap, implement RAII, enable sanitizers, and leverage standard library containers.

Exercise 10. ***Practice Exercises:***

1. *Write a program that demonstrates a memory leak, then fix it using `std::unique_ptr`.*
2. *Create a circular linked list that causes a memory leak with `std::shared_ptr`, then fix it using `std::weak_ptr`.*

3. *Implement a RAII wrapper for a POSIX file descriptor (using* `open()` *and* `close()`*).*

4. *Using* `std::make_unique_for_overwrite` *(C++23), allocate an array of 1 million integers and measure performance compared to* `std::make_unique`*.*

5. *Research AddressSanitizer and write a small program with a buffer overflow. Compile with* `-fsanitize=address` *and observe the error report.*

6. *Implement a custom memory pool that allocates fixed-size blocks and manages them using raw pointers, then compare with* `std::vector`*.*

Chapter 4

Modern C++ and Smart Pointers

4.1 Introduction to Smart Pointers

In modern C++, memory management is one of the most critical aspects of writing efficient, safe, and maintainable code. Traditional raw pointers, while powerful, come with significant risks, such as memory leaks, dangling pointers, and double deletions. These issues can lead to undefined behavior, crashes, and security vulnerabilities. To address these challenges, C++11 introduced **smart pointers**, which are a cornerstone of modern C++ programming. Smart pointers provide automatic memory management, ensuring that resources are properly deallocated when they are no longer needed. This section introduces the concept of smart pointers, explains why they are essential, explores the three main types of smart pointers in C++, and demonstrates how to replace raw pointers with `std::unique_ptr` through practical examples.

4.1.1 Why Smart Pointers?

Raw pointers in C++ require manual memory management, which can lead to several common pitfalls:

1. **Memory Leaks**: If memory allocated on the heap is not explicitly deallocated using `delete`, it remains occupied even after the program finishes using it. Over time, this can exhaust available memory, leading to performance degradation or program crashes.

2. **Dangling Pointers**: When a pointer points to memory that has already been deallocated, accessing or modifying it leads to undefined behavior. This can cause crashes, data corruption, or security vulnerabilities.

3. **Double Deletion**: Accidentally deleting the same memory location twice can corrupt the program's state and cause crashes. This often happens when multiple pointers point to the same resource, and the programmer loses track of ownership.

4. **Exception Safety**: If an exception is thrown before `delete` is called, the memory may never be released. This can lead to memory leaks in exception-prone code paths.

5. **Ownership Ambiguity**: Raw pointers do not convey ownership semantics. It is unclear whether a raw pointer owns the resource it points to or merely observes it. This ambiguity can lead to bugs and maintenance challenges.

Important

The Golden Rule of Modern C++: If you use `new`, you are likely doing something wrong. Prefer smart pointers and containers for dynamic memory management.

Smart pointers address these issues by wrapping raw pointers in objects that automatically manage their lifetime. They ensure that memory is deallocated when it is no longer needed, even in the presence of exceptions. This makes code safer, more robust, and easier to maintain. Smart pointers also provide clear ownership semantics, making it easier to reason about resource management in complex programs.

4.1.2 Types of Smart Pointers

C++ provides three types of smart pointers, each designed for specific use cases:

4.1.2.1 `std::unique_ptr`

`std::unique_ptr` is a smart pointer that enforces **exclusive ownership** of a dynamically allocated object. It ensures that only one `unique_ptr` can point to a resource at any given time. When the `unique_ptr` goes out of scope, the resource it manages is automatically deallocated.

- **Key Features**:

  - **Exclusive Ownership**: A `unique_ptr` cannot be copied, only moved. This ensures that there is only one owner of the resource at any time.
  - **Lightweight and Efficient**: `unique_ptr` has minimal overhead, making it as efficient as raw pointers in most cases.
  - **Custom Deleters**: You can specify a custom deleter function or lambda to handle specialized cleanup, such as closing file handles or releasing other resources.
  - **Exception Safety**: `unique_ptr` ensures that resources are released even if an exception is thrown.
  - **Array Support**: Specialization for arrays with `unique_ptr<T[]>` provides `operator[]`.

- **Use Cases**:
  - Managing resources with single ownership.
  - Returning dynamically allocated objects from functions.
  - Ensuring exception safety in resource management.
  - Implementing the Pimpl (Pointer to Implementation) idiom.
  - Managing dynamically allocated arrays.

- **Example**:

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

void exampleUniquePtr() {
    // Create a unique_ptr to manage an integer (C++11 style)
    std::unique_ptr<int> ptr1(new int(42));

    // Prefer make_unique (C++14 and later)
    auto ptr2 = std::make_unique<int>(42);  // Safer and more efficient

    std::cout << *ptr2 << '\n';  // Access the value

    // Transfer ownership
    auto ptr3 = std::move(ptr2);
    if (!ptr2) {
        std::cout << "ptr2 is now null\n";
```

```
    }

    // Array version
    auto arr = std::make_unique<int[]>(10);
    arr[0] = 42;  // operator[] works with unique_ptr<T[]>

}  // All memory automatically freed
```

- **Advanced Usage**:
  - **Custom Deleters**:

```
auto fileDeleter = [](FILE* f) {
    if (f) std::fclose(f);
};
std::unique_ptr<FILE, decltype(fileDeleter)> filePtr(
    std::fopen("data.txt", "r"), fileDeleter);
```

  - **C++23: `std::make_unique_for_overwrite` for performance**:

```
auto arr = std::make_unique_for_overwrite<int[]>(1000);
// Elements are default-initialized (uninitialized) - faster
std::fill_n(arr.get(), 1000, 42);
```

4.1.2.2 `std::shared_ptr`

`std::shared_ptr` is a smart pointer that implements **shared ownership**. Multiple `shared_ptr` instances can point to the same resource, and the resource is deallocated only when the last `shared_ptr` pointing to it is destroyed or reset. This is achieved using **reference counting**.

- **Key Features**:
  - **Shared Ownership**: Multiple `shared_ptr` instances can manage the same resource.
  - **Reference Counting**: Tracks the number of `shared_ptr` instances pointing to the resource.
  - **Thread Safety**: Reference counting is thread-safe (atomic operations), but accessing the underlying resource is not.
  - **Custom Deleters**: Supports custom deleters for specialized cleanup.
  - **Slightly Higher Overhead**: Due to reference counting, `shared_ptr` has more overhead than `unique_ptr`.

- **Use Cases**:
  - Managing resources with shared ownership.
  - Data structures like graphs or trees where multiple nodes may reference the same data.
  - Implementing caches or observer patterns.
  - Resource pools where objects are shared.
- **Example**:

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

void exampleSharedPtr() {
    // Create a shared_ptr (C++11 style)
    std::shared_ptr<int> ptr1(new int(42));

    // Prefer make_shared (C++11)
    auto ptr2 = std::make_shared<int>(42);  // More efficient

    auto ptr3 = ptr2;  // Share ownership, reference count = 2

    std::cout << *ptr2 << '\n';
    std::cout << "Use count: " << ptr2.use_count() << '\n';  // 2

    ptr2.reset();  // Release ownership
    std::cout << "Use count: " << ptr3.use_count() << '\n';  // 1
}  // ptr3 goes out of scope, memory freed
```

- **Advanced Usage**:
  - **Aliasing Constructor**: Share ownership of one object while pointing to another.

```cpp
struct Foo { int value; };
auto fooPtr = std::make_shared<Foo>(Foo{42});
std::shared_ptr<int> aliasPtr(fooPtr, &fooPtr->value);
// Both share ownership of the Foo object
```

  - **C++20: `std::make_shared` for arrays**:

```cpp
auto arr = std::make_shared<int[]>(10);  // C++20
arr[0] = 42;
```

  - **C++23: `std::make_shared_for_overwrite`**:

```cpp
auto arr = std::make_shared_for_overwrite<int[]>(1000);
// Default-initialized (uninitialized) - faster
```

4.1.2.3 `std::weak_ptr`

`std::weak_ptr` is a smart pointer that provides **non-owning (weak) references** to a resource managed by a `std::shared_ptr`. Unlike `shared_ptr`, `weak_ptr` does not increment the reference count, which helps break circular references that can lead to memory leaks.

- **Key Features**:
  - **Non-Owning Reference**: Does not affect the reference count.
  - **Used to Observe**: Can be used to observe a resource without extending its lifetime.
  - **Convertible to `shared_ptr`**: Can be converted to a `shared_ptr` using `lock()` to safely access the resource.
  - **Prevents Circular References**: Helps break circular dependencies between `shared_ptr` instances.
- **Use Cases**:
  - Breaking circular references between `shared_ptr` instances.
  - Caching mechanisms where the cached object should not prevent its deletion.
  - Observer patterns where observers do not need to extend the lifetime of the subject.
  - Avoiding memory leaks in parent-child relationships.
- **Example**:

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

void exampleWeakPtr() {
    auto sharedPtr = std::make_shared<int>(42);
    std::weak_ptr<int> weakPtr = sharedPtr;  // Non-owning reference

    // Safely access the resource
    if (auto lockedPtr = weakPtr.lock()) {
        std::cout << "Value: " << *lockedPtr << '\n';
    } else {
        std::cout << "Resource no longer exists\n";
    }
```

```
    sharedPtr.reset();  // Release the resource

    if (weakPtr.expired()) {
        std::cout << "Resource has been deleted\n";
    }
}
```

- **Advanced Usage: Breaking Circular References**:

```
struct Node {
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;  // Weak pointer prevents cycles
    int value;

    Node(int v) : value(v) {}
};

void exampleCircular() {
    auto node1 = std::make_shared<Node>(1);
    auto node2 = std::make_shared<Node>(2);

    node1->next = node2;
    node2->prev = node1;  // Weak reference - no cycle

    // When node1 goes out of scope, node2 can be properly destroyed
    // because prev is a weak_ptr, not a shared_ptr
}
```

4.1.3 Example: Replacing Raw Pointers with `std::unique_ptr`

To demonstrate the benefits of smart pointers, let's consider a scenario where raw pointers are used to manage dynamic memory. We will then refactor the code to use `std::unique_ptr`.

4.1.3.1 Raw Pointer Example

```
#include <iostream>

void rawPointerExample() {
    int* rawPtr = new int(42);  // Dynamically allocate memory
    std::cout << *rawPtr << '\n';

    // What if an exception occurs here? Memory leak!
```

```
    delete rawPtr;  // Must manually deallocate
}
```

4.1.3.2 Problems with Raw Pointers

1. **Memory Leak Risk**: If `delete` is forgotten, the memory is leaked.
2. **Exception Unsafe**: If an exception is thrown before `delete`, the memory is leaked.
3. **Manual Management**: The programmer must manually track the lifetime of the resource.
4. **No Clear Ownership**: It's unclear who owns the memory.

4.1.3.3 Refactored with `std::unique_ptr`

```
#include <memory>
#include <iostream>

void uniquePtrExample() {
    // Modern C++: Use make_unique
    auto smartPtr = std::make_unique<int>(42);
    std::cout << *smartPtr << '\n';

    // If an exception occurs here, memory is still freed!

}  // Memory automatically deallocated when smartPtr goes out of scope
```

4.1.3.4 Benefits of `std::unique_ptr`

1. **Automatic Cleanup**: Memory is automatically deallocated when the `unique_ptr` goes out of scope.
2. **Exception Safety**: Resources are released even if an exception is thrown.
3. **Clear Ownership**: The ownership semantics are explicit, making the code easier to understand.
4. **No Manual `delete`**: Eliminates the risk of forgetting to deallocate.

4.1.4 Modern C++ Enhancements for Smart Pointers

1. **C++14: `std::make_unique`**:

```
auto ptr = std::make_unique<MyClass>(args...);  // C++14
```

2. **C++17: `std::shared_ptr` for Arrays**:

```
auto arr = std::make_shared<int[]>(10);  // C++20, but improved in C++17
```

3. **C++20: `std::make_shared` for Arrays**:

```
auto arr = std::make_shared<int[]>(10);  // C++20
```

4. **C++23: `std::make_unique_for_overwrite` and `std::make_shared_for_overwrite`**:

```
// Skip value-initialization for performance
auto arr = std::make_unique_for_overwrite<int[]>(1000);
auto sharedArr = std::make_shared_for_overwrite<int[]>(1000);
```

5. **C++23: `std::out_ptr` and `std::inout_ptr`**:

```
// Safely interface with C APIs
extern "C" void create_resource(void** out);
std::unique_ptr<Resource, decltype(&destroy_resource)> resource(nullptr,
↪  destroy_resource);
create_resource(std::out_ptr(resource));  // C++23
```

4.1.5 C++26 Preview: `std::observer_ptr`

The proposed C++26 standard introduces `std::observer_ptr`, a vocabulary type for non-owning pointers that clearly expresses intent.

```
// Proposed C++26
#include <observer_ptr>

class Widget {
public:
    void process(std::observer_ptr<const Data> data) {
        if (data) {  // Explicit null check
            // Use data safely
        }
    }
```

```
};

int main() {
    Data d;
    Widget w;
    w.process(std::observer_ptr(&d));  // Clear: non-owning observation
    return 0;
}
```

4.1.6 Choosing the Right Smart Pointer

Requirement	Recommended	Why
Single ownership	`std::unique_ptr`	Zero overhead, exclusive ownership
Shared ownership	`std::shared_ptr`	Reference counting, multiple owners
Non-owning observation	Raw pointer or `std::observer_ptr` (C++26)	No lifetime management
Breaking cycles	`std::weak_ptr`	Prevents memory leaks
Dynamic arrays	`std::unique_ptr<T[]>` or `std::vector`	Automatic management
Large objects	`std::shared_ptr` with `make_shared`	Single allocation for object + control block

4.1.7 Best Practices

1. **Prefer `std::make_unique` and `std::make_shared`**:
    - Safer (exception-safe), more efficient (single allocation for `make_shared`).
2. **Use `std::unique_ptr` as Default**:
    - Use `shared_ptr` only when shared ownership is truly needed.
3. **Avoid Circular References with `shared_ptr`**:
    - Use `std::weak_ptr` to break cycles.
4. **Use `std::unique_ptr` for Factory Functions**:

- Return `unique_ptr` to transfer ownership clearly.

5. **Never Use Smart Pointers for Non-Owning Pointers**:
   - Use raw pointers or `std::observer_ptr` (C++26) for observation.
6. **Be Aware of `shared_ptr` Overhead**:
   - Reference counting adds memory and performance overhead.

4.1.8 Summary

- Smart pointers are essential tools in modern C++ that eliminate manual memory management risks.
- `std::unique_ptr` provides exclusive ownership with zero overhead.
- `std::shared_ptr` enables shared ownership with reference counting.
- `std::weak_ptr` breaks circular references in shared ownership scenarios.
- C++14 added `make_unique`; C++17/20 improved array support; C++23 added `make_unique_for_overwrite` and `out_ptr`.
- C++26 will introduce `std::observer_ptr` for clear non-owning semantics.
- Best practices: prefer `make_unique`/`make_shared`, use `unique_ptr` as default, break cycles with `weak_ptr`.

Exercise 11. ***Practice Exercises:***

1. *Rewrite a function that manually allocates an array with `new[]` to use `std::unique_ptr<int[]>` instead.*
2. *Create a graph where nodes have edges to other nodes. Use `std::shared_ptr` for edges and `std::weak_ptr` to prevent cycles.*
3. *Implement a factory function that returns `std::unique_ptr<Base>` to a derived class. Demonstrate proper usage.*
4. *Using C++23's `std::make_unique_for_overwrite`, allocate a large array and measure the performance difference compared to `std::make_unique`.*

5. *Research* `std::observer_ptr` *(C++26 draft) and write a small example showing how it improves code clarity over raw pointers for non-owning relationships.*

6. *Create a simple cache that uses* `std::weak_ptr` *to store objects that can be reclaimed when memory pressure occurs.*

4.2 Using `std::unique_ptr`

`std::unique_ptr` is one of the most commonly used smart pointers in modern C++. It enforces **exclusive ownership** of a dynamically allocated resource, ensuring that only one `unique_ptr` can own the resource at any given time. This section delves into the ownership semantics of `std::unique_ptr`, demonstrates how to manage dynamic arrays, explores the use of custom deleters, and provides practical examples, including managing file handles with a custom deleter. We will also cover advanced topics such as using `std::unique_ptr` with polymorphism, integrating it with STL containers, and leveraging it in modern C++ contexts.

Important

Golden Rule: Prefer `std::make_unique` (C++14) over direct `new` for creating `unique_ptr` instances. It is safer (exception-safe) and more concise.

4.2.1 Ownership Semantics

The primary feature of `std::unique_ptr` is its **exclusive ownership** model. This means that a `unique_ptr` cannot be copied, only moved. When a `unique_ptr` is moved, ownership of the resource is transferred to the new `unique_ptr`, and the original `unique_ptr` is set to `nullptr`. This ensures that there is only one owner of the resource at any time, preventing issues like double deletion.

Important

Key Properties:

- **Exclusive Ownership**: Only one `unique_ptr` can own a resource at a time.
- **Move-Only**: Cannot be copied, but can be moved using `std::move`.
- **Automatic Cleanup**: When the `unique_ptr` goes out of scope, the resource it owns is automatically deallocated.
- **Exception Safety**: Resources are released even if an exception is thrown.
- **Zero Overhead**: `unique_ptr` has no space or time overhead compared to raw pointers (when using default deleter).

Example:

```
#include <memory>
#include <iostream>

void ownershipSemanticsExample() {
    // C++11 style (not recommended for modern code)
    std::unique_ptr<int> ptr1(new int(42));

    // Modern C++: prefer make_unique (C++14)
    auto ptr2 = std::make_unique<int>(42);  // Cleaner and exception-safe

    std::cout << *ptr2 << '\n';

    // Transfer ownership
    auto ptr3 = std::move(ptr2);

    if (!ptr2) {
        std::cout << "ptr2 is now null\n";
    }

    std::cout << *ptr3 << '\n';  // ptr3 now owns the resource
}  // ptr3 goes out of scope, memory automatically freed
```

4.2.2 Managing Dynamic Arrays with `std::unique_ptr`

`std::unique_ptr` can also manage dynamically allocated arrays using the `T[]` specialization. This ensures that the array is properly deallocated with `delete[]` when the `unique_ptr` goes out of scope.

Important

Array Specialization:

- Use `std::unique_ptr<T[]>` for array management.
- Provides `operator[]` for element access.
- Automatically uses `delete[]` for deallocation.

Example:

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

void dynamicArrayExample() {
    // C++11 style (not recommended)
    // std::unique_ptr<int[]> arr(new int[5]{1, 2, 3, 4, 5});

    // Modern C++: prefer make_unique for arrays (C++14)
    auto arr = std::make_unique<int[]>(5);

    // Initialize array
    for (int i = 0; i < 5; ++i) {
        arr[i] = i + 1;
    }

    // Access and modify array elements
    for (int i = 0; i < 5; ++i) {
        std::cout << arr[i] << ' ';
        arr[i] *= 2;
    }
    std::cout << '\n';

    // Print modified array
    for (int i = 0; i < 5; ++i) {
```

```
        std::cout << arr[i] << ' ';
    }
    std::cout << '\n';
}  // Array automatically deallocated
```

C++23: `std::make_unique_for_overwrite` for Performance

When you need to allocate an array that will be immediately overwritten, using `make_unique_for_overwrite` avoids unnecessary value-initialization, improving performance.

```
#include <memory>
#include <algorithm>

void performanceArrayExample() {
    // C++23: Default-initialized (uninitialized) - faster
    auto arr = std::make_unique_for_overwrite<int[]>(1000000);

    // Immediately fill with values
    std::fill_n(arr.get(), 1000000, 42);

    // Use arr...
}  // Automatically deallocated
```

4.2.3 Custom Deleters

By default, `std::unique_ptr` uses `delete` (or `delete[]` for arrays) to deallocate the resource it owns. However, you can specify a **custom deleter** to handle specialized cleanup, such as closing file handles, releasing mutexes, or freeing resources allocated by custom allocators.

Important

Custom Deleters:

- Can be functions, function objects, or lambdas.
- Passed as the second template argument and constructor argument.
- Enable `unique_ptr` to manage non-memory resources.
- The deleter type is part of the `unique_ptr` type, so different deleters produce different types.

Example: Using a Custom Deleter for File Handles

```
#include <memory>
#include <iostream>
#include <cstdio>

void fileHandleExample() {
    // Custom deleter for FILE* (lambda)
    auto fileDeleter = [](FILE* file) {
        if (file) {
            std::cout << "Closing file\n";
            std::fclose(file);
        }
    };

    // Create unique_ptr with custom deleter
    std::unique_ptr<FILE, decltype(fileDeleter)> filePtr(
        std::fopen("example.txt", "w"), fileDeleter);

    if (filePtr) {
        std::cout << "File opened successfully\n";
        std::fprintf(filePtr.get(), "Hello, Modern C++!");
    } else {
        std::cerr << "Failed to open file\n";
    }

    // File automatically closed when filePtr goes out of scope
}
```

Example: Custom Deleter with Function Object

```
struct FileDeleter {
    void operator()(FILE* file) const {
        if (file) {
            std::cout << "Closing file\n";
            std::fclose(file);
        }
    }
};

void fileHandleWithFunctor() {
    std::unique_ptr<FILE, FileDeleter> filePtr(
        std::fopen("example.txt", "w"));
```

```
    if (filePtr) {
        std::fprintf(filePtr.get(), "Hello, World!");
    }
}  // File automatically closed
```

4.2.4 Advanced Custom Deleters with Arrays

Custom deleters can also be used with `std::unique_ptr<T[]>` for specialized array cleanup.

```
#include <memory>
#include <iostream>

void customDeleterArrayExample() {
    // Custom deleter for arrays
    auto arrayDeleter = [](int* arr) {
        std::cout << "Deleting array with custom deleter\n";
        delete[] arr;  // Must use delete[] for arrays
    };

    // Create unique_ptr with custom deleter
    std::unique_ptr<int[], decltype(arrayDeleter)> arr(
        new int[5]{1, 2, 3, 4, 5}, arrayDeleter);

    // Access and modify array elements
    for (int i = 0; i < 5; ++i) {
        std::cout << arr[i] << ' ';
        arr[i] *= 2;
    }
    std::cout << '\n';
}  // Array deallocated using custom deleter
```

4.2.5 Using `std::unique_ptr` with Polymorphism

`std::unique_ptr` can manage polymorphic objects, ensuring that the correct destructor is called when the `unique_ptr` goes out of scope.

Important

Key Point: Always declare the base class destructor as `virtual` when using polymorphism with smart pointers.

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

class Base {
public:
    virtual void print() const {
        std::cout << "Base class\n";
    }
    virtual ~Base() = default;  // Virtual destructor is essential!
};

class Derived : public Base {
public:
    void print() const override {
        std::cout << "Derived class\n";
    }
    ~Derived() {
        std::cout << "Derived destructor\n";
    }
};

class AnotherDerived : public Base {
public:
    void print() const override {
        std::cout << "AnotherDerived class\n";
    }
};

void polymorphismExample() {
    // Factory function returning unique_ptr to base
    auto createShape = [](int type) -> std::unique_ptr<Base> {
        if (type == 1) {
            return std::make_unique<Derived>();
        } else {
            return std::make_unique<AnotherDerived>();
        }
    };

    auto ptr = createShape(1);
    ptr->print();  // Calls Derived::print()

    auto ptr2 = createShape(2);
```

```
    ptr2->print();  // Calls AnotherDerived::print()

    // Both objects automatically destroyed with correct destructors
}
```

4.2.6 Integrating `std::unique_ptr` with STL Containers

`std::unique_ptr` works seamlessly with STL containers like `std::vector`. Since `unique_ptr` is move-only, you must use `std::move` or emplace functions to insert elements.

```
#include <memory>
#include <vector>
#include <iostream>
#include <algorithm>

void stlContainerExample() {
    std::vector<std::unique_ptr<int>> vec;

    // Add elements using make_unique and move
    vec.push_back(std::make_unique<int>(10));
    vec.push_back(std::make_unique<int>(20));
    vec.push_back(std::make_unique<int>(30));

    // C++17: emplace_back with make_unique
    vec.emplace_back(std::make_unique<int>(40));

    // Access elements
    std::cout << "Elements: ";
    for (const auto& ptr : vec) {
        std::cout << *ptr << ' ';
    }
    std::cout << '\n';

    // Modify elements using range-based for loop
    for (auto& ptr : vec) {
        *ptr *= 2;
    }

    // Transfer ownership from vector to local unique_ptr
    auto extracted = std::move(vec[0]);
    std::cout << "Extracted value: " << *extracted << '\n';
```

```
    // Remove element from vector
    vec.erase(vec.begin());

    // All remaining elements automatically deleted when vector goes out of scope
}
```

4.2.7 Using `std::unique_ptr` in Factory Functions

`std::unique_ptr` is ideal for factory functions that return dynamically allocated objects.

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

class Logger {
public:
    virtual void log(const std::string& message) = 0;
    virtual ~Logger() = default;
};

class FileLogger : public Logger {
    std::string filename;
public:
    explicit FileLogger(std::string name) : filename(std::move(name)) {}
    void log(const std::string& message) override {
        std::cout << "[FileLogger:" << filename << "] " << message << '\n';
    }
};

class ConsoleLogger : public Logger {
public:
    void log(const std::string& message) override {
        std::cout << "[Console] " << message << '\n';
    }
};

// Factory function returns unique_ptr (clear ownership transfer)
std::unique_ptr<Logger> createLogger(const std::string& type) {
    if (type == "file") {
        return std::make_unique<FileLogger>("app.log");
    } else {
        return std::make_unique<ConsoleLogger>();
```

```
    }
}

void factoryExample() {
    auto logger = createLogger("file");
    logger->log("Application started");

    logger = createLogger("console");
    logger->log("Switched to console");
}
```

4.2.8 C++23: `std::out_ptr` for C API Interoperability

C++23 introduces `std::out_ptr` and `std::inout_ptr` to safely interface with C APIs that expect raw pointer outputs.

```
#include <memory>
#include <vector>

// C API that allocates memory
extern "C" {
    struct Buffer;
    void create_buffer(Buffer** out, size_t* size);
    void destroy_buffer(Buffer* buf);
}

// Modern C++ wrapper using unique_ptr
std::unique_ptr<Buffer, decltype(&destroy_buffer)> createBuffer() {
    std::unique_ptr<Buffer, decltype(&destroy_buffer)> buffer(nullptr,
    ↪  destroy_buffer);
    size_t size = 0;

    // C++23: Safely pass unique_ptr to C API
    create_buffer(std::out_ptr(buffer), &size);

    return buffer;
}

void cApiExample() {
    auto buffer = createBuffer();
    // Use buffer...
    // Automatically destroyed with destroy_buffer
```

```
}
```

4.2.9 Using `std::unique_ptr` in Multithreaded Environments

While `std::unique_ptr` itself is not thread-safe, it can be used in multithreaded environments with proper synchronization.

```
#include <memory>
#include <thread>
#include <mutex>
#include <vector>
#include <iostream>

class ThreadSafeContainer {
    std::vector<std::unique_ptr<int>> data;
    mutable std::mutex mtx;

public:
    void add(std::unique_ptr<int> value) {
        std::lock_guard<std::mutex> lock(mtx);
        data.push_back(std::move(value));
    }

    std::unique_ptr<int> removeLast() {
        std::lock_guard<std::mutex> lock(mtx);
        if (data.empty()) return nullptr;
        auto result = std::move(data.back());
        data.pop_back();
        return result;
    }

    void print() const {
        std::lock_guard<std::mutex> lock(mtx);
        for (const auto& ptr : data) {
            std::cout << *ptr << ' ';
        }
        std::cout << '\n';
    }
};

void multithreadedExample() {
    ThreadSafeContainer container;
```

```
    auto producer = [&container]() {
        for (int i = 0; i < 100; ++i) {
            container.add(std::make_unique<int>(i));
        }
    };

    auto consumer = [&container]() {
        for (int i = 0; i < 50; ++i) {
            auto val = container.removeLast();
            if (val) {
                std::cout << "Consumed: " << *val << '\n';
            }
        }
    };

    std::thread t1(producer);
    std::thread t2(consumer);
    std::thread t3(consumer);

    t1.join();
    t2.join();
    t3.join();
}
```

4.2.10 C++26 Preview: `std::observer_ptr` as Alternative for Non-Owning Observation

The proposed C++26 standard introduces `std::observer_ptr` for non-owning observation, clarifying intent where raw pointers were previously used.

```
// Proposed C++26
#include <observer_ptr>

class Widget {
public:
    void process(std::observer_ptr<const Data> data) {
        if (data) {
            // Use data safely - no ownership concerns
        }
    }
```

```
};

void observerPtrExample() {
    Data d;
    Widget w;
    w.process(std::observer_ptr(&d));  // Clear non-owning intent
}
```

4.2.11 Best Practices for `std::unique_ptr`

1. **Always Prefer `std::make_unique`**:
    - Safer (exception-safe) and more concise than `new`.
    - Use `std::make_unique_for_overwrite` (C++23) when default-initialization is acceptable.
2. **Use `unique_ptr` as Default Smart Pointer**:
    - Only use `shared_ptr` when shared ownership is actually needed.
3. **Return `unique_ptr` from Factory Functions**:
    - Clearly transfers ownership to the caller.
4. **Use `std::move` to Transfer Ownership**:
    - Explicitly shows ownership transfer in code.
5. **Use Custom Deleters for Non-Memory Resources**:
    - File handles, sockets, mutexes, etc.
6. **Never Use `unique_ptr` for Non-Owning Pointers**:
    - Use raw pointers or `std::observer_ptr` (C++26) for observation.

4.2.12 Common Pitfalls

1. **Using `unique_ptr` with Wrong Deleter**:

```
// Wrong: using delete for array
std::unique_ptr<int[]> arr(new int[10]);  // OK
std::unique_ptr<int> arr(new int[10]);    // Wrong! Will use delete not
↪  delete[]
```

2. **Attempting to Copy `unique_ptr`**:

```
auto ptr1 = std::make_unique<int>(42);
auto ptr2 = ptr1;  // Error: unique_ptr is not copyable
```

3. **Returning `unique_ptr` to Local Variable**:

```
std::unique_ptr<int> bad() {
    int x = 42;
    return std::unique_ptr<int>(&x);  // Wrong! x is on stack
}
```

4. **Forgetting Virtual Destructor in Polymorphic Base Classes**:

```
class Base {
public:
    ~Base() {}  // Non-virtual - derived destructor won't be called!
};
```

4.2.13 Summary

- `std::unique_ptr` provides exclusive ownership with zero overhead.
- Move-only semantics prevent accidental copying and ensure clear ownership transfer.
- Array specialization (`unique_ptr<T[]>`) provides safe dynamic array management.
- Custom deleters enable management of non-memory resources like file handles.
- Works seamlessly with polymorphism, STL containers, and factory functions.
- C++14 added `make_unique`; C++20 improved array support; C++23 added `make_unique_for_overwrite` and `out_ptr`.
- C++26 will introduce `std::observer_ptr` for clear non-owning semantics.
- Best practices: prefer `make_unique`, use `unique_ptr` as default, return from factories, and never use for non-owning observation.

Exercise 12. ***Practice Exercises:***

1. *Implement a* `unique_ptr` *wrapper for a POSIX file descriptor (using* `open()` *and* `close()`*) with a custom deleter.*
2. *Create a factory function that returns* `std::unique_ptr<Base>` *to different derived classes based on a string parameter.*
3. *Write a program that stores* `std::unique_ptr<Widget>` *in a* `std::vector`*, then transfers ownership of specific elements out of the vector.*
4. *Using C++23's* `std::make_unique_for_overwrite`*, allocate a large array and fill it efficiently.*
5. *Implement a simple resource pool that manages* `std::unique_ptr` *instances and reuses them.*
6. *Research* `std::out_ptr` *(C++23) and write a wrapper for a C API that returns allocated memory.*

4.3 Using `std::shared_ptr`

`std::shared_ptr` is a smart pointer that implements **shared ownership** of a dynamically allocated resource. Unlike `std::unique_ptr`, which enforces exclusive ownership, `std::shared_ptr` allows multiple pointers to share ownership of the same resource. The resource is deallocated only when the last `shared_ptr` pointing to it is destroyed or reset. This section explores the shared ownership semantics of `std::shared_ptr`, demonstrates how to share resources between multiple objects, discusses the issue of circular references, and shows how to break them using `std::weak_ptr`. We will also cover advanced topics such as custom deleters, aliasing constructors, thread safety considerations, and performance implications.

Important

Golden Rule: Prefer `std::make_shared` over direct `new` for creating `shared_ptr` instances. It is safer (exception-safe) and more efficient (single allocation for the object and control block).

4.3.1 Shared Ownership Semantics

The primary feature of `std::shared_ptr` is its **shared ownership** model. This means that multiple `shared_ptr` instances can point to the same resource, and the resource is deallocated only when the last `shared_ptr` is destroyed or reset. This is achieved using **reference counting**, where each `shared_ptr` increments a reference count when it is copied and decrements it when it is destroyed or reset.

Important

Key Properties:

- **Shared Ownership**: Multiple `shared_ptr` instances can manage the same resource.
- **Reference Counting**: Tracks the number of `shared_ptr` instances pointing to the resource.
- **Automatic Cleanup**: The resource is deallocated when the reference count drops to zero.
- **Thread Safety**: Reference counting is thread-safe (atomic operations), but accessing the underlying resource is not.
- **Overhead**: `shared_ptr` has more overhead than `unique_ptr` due to reference counting and control block allocation.

Example:

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

void sharedOwnershipExample() {
    // C++11 style (not recommended for modern code)
    std::shared_ptr<int> ptr1(new int(42));

    // Modern C++: prefer make_shared (C++11)
    auto ptr2 = std::make_shared<int>(42);

    std::cout << "Use count: " << ptr2.use_count() << '\n';  // 1

    // Share ownership
    auto ptr3 = ptr2;  // Reference count = 2
    std::cout << "Use count: " << ptr2.use_count() << '\n';  // 2

    // Access the resource
    std::cout << *ptr2 << ' ' << *ptr3 << '\n';

    ptr2.reset();  // Release ownership, count = 1
    std::cout << "Use count after reset: " << ptr3.use_count() << '\n';
```

```
}  // ptr3 goes out of scope, memory freed
```

4.3.2 Sharing Resources Between Multiple Objects

`std::shared_ptr` is particularly useful when multiple objects need to share access to the same resource. For example, in a graph or tree data structure, multiple nodes may reference the same data.

Example: Shared Node in a Graph

```
#include <memory>
#include <iostream>
#include <vector>

class GraphNode {
public:
    int value;
    std::vector<std::shared_ptr<GraphNode>> neighbors;

    explicit GraphNode(int val) : value(val) {
        std::cout << "Node " << value << " created\n";
    }

    ~GraphNode() {
        std::cout << "Node " << value << " destroyed\n";
    }

    void addNeighbor(std::shared_ptr<GraphNode> neighbor) {
        neighbors.push_back(neighbor);
    }
};

void sharedResourceExample() {
    auto node1 = std::make_shared<GraphNode>(1);
    auto node2 = std::make_shared<GraphNode>(2);
    auto node3 = std::make_shared<GraphNode>(3);

    // node2 and node3 both have references to node1
    node2->addNeighbor(node1);
    node3->addNeighbor(node1);

    std::cout << "node1 use count: " << node1.use_count() << '\n';  // 3
```

```
    // node1 remains alive as long as any node references it
}
```

4.3.3 Circular References and `std::weak_ptr`

One of the challenges with `std::shared_ptr` is the potential for **circular references**. A circular reference occurs when two or more `shared_ptr` instances reference each other, creating a cycle. This prevents the reference count from dropping to zero, leading to memory leaks.

Important

Circular Reference Problem: When objects form a cycle using `shared_ptr`, their reference counts never reach zero, causing memory leaks. Use `std::weak_ptr` to break cycles.

Example of Circular Reference (Memory Leak):

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

class Node {
public:
    std::shared_ptr<Node> next;
    Node() { std::cout << "Node created\n"; }
    ~Node() { std::cout << "Node destroyed\n"; }
};

void circularReferenceExample() {
    auto node1 = std::make_shared<Node>();
    auto node2 = std::make_shared<Node>();

    // Create a circular reference
    node1->next = node2;
    node2->next = node1;

    std::cout << "node1 use count: " << node1.use_count() << '\n';  // 2
    std::cout << "node2 use count: " << node2.use_count() << '\n';  // 2

    // Destructors never called! Memory leak!
}
```

4.3.4 Breaking Circular References with `std::weak_ptr`

To break circular references, C++ provides `std::weak_ptr`. A `weak_ptr` is a non-owning reference to a resource managed by a `shared_ptr`. It does not increment the reference count, allowing the resource to be deallocated when the last `shared_ptr` is destroyed.

Important

Key Properties of `std::weak_ptr`:

- **Non-Owning Reference**: Does not affect the reference count.
- **Convertible to `shared_ptr`**: Use `lock()` to safely obtain a `shared_ptr` if the resource still exists.
- **Expiration Check**: Use `expired()` to check if the resource has been destroyed.
- **Prevents Circular References**: Essential for breaking cycles in shared ownership graphs.

Example: Breaking Circular Reference with `weak_ptr`

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

class Node {
public:
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;  // Use weak_ptr to break circular reference

    Node() { std::cout << "Node created\n"; }
    ~Node() { std::cout << "Node destroyed\n"; }
};

void weakPtrExample() {
    auto node1 = std::make_shared<Node>();
    auto node2 = std::make_shared<Node>();

    // Create bidirectional links with weak_ptr for one direction
    node1->next = node2;
    node2->prev = node1;  // weak_ptr - no cycle
```

```
    std::cout << "node1 use count: " << node1.use_count() << '\n';  // 1 (only
    ↪  node1 owns itself)
    std::cout << "node2 use count: " << node2.use_count() << '\n';  // 1 (only
    ↪  node2 owns itself)

    // Safely access through weak_ptr
    if (auto locked = node2->prev.lock()) {
        std::cout << "Accessed node1 through weak_ptr\n";
    }

}  // Both nodes properly destroyed
```

4.3.5 Custom Deleters with `std::shared_ptr`

Like `std::unique_ptr`, `std::shared_ptr` supports custom deleters. This allows you to specify a custom cleanup function when the resource is no longer needed.

```
#include <memory>
#include <iostream>
#include <cstdio>

void customDeleterExample() {
    // Custom deleter for FILE*
    auto fileDeleter = [](FILE* file) {
        if (file) {
            std::cout << "Closing file\n";
            std::fclose(file);
        }
    };

    // shared_ptr with custom deleter
    std::shared_ptr<FILE> filePtr(std::fopen("example.txt", "w"), fileDeleter);

    if (filePtr) {
        std::fprintf(filePtr.get(), "Hello, World!");
    }

}  // File automatically closed
```

4.3.6 Aliasing Constructor

The **aliasing constructor** of `std::shared_ptr` allows a `shared_ptr` to share ownership of one object while pointing to another. This is useful when you want to manage a subobject or a member of a larger object.

```
#include <memory>
#include <iostream>

struct Foo {
    int value;
    std::string name;

    Foo(int v, std::string n) : value(v), name(std::move(n)) {}
};

void aliasingConstructorExample() {
    auto fooPtr = std::make_shared<Foo>(42, "answer");

    // shared_ptr to member, sharing ownership of the whole Foo
    std::shared_ptr<int> valuePtr(fooPtr, &fooPtr->value);
    std::shared_ptr<std::string> namePtr(fooPtr, &fooPtr->name);

    std::cout << "Value: " << *valuePtr << '\n';        // 42
    std::cout << "Name: " << *namePtr << '\n';          // answer

    // Both valuePtr and namePtr share ownership with fooPtr
    std::cout << "Use count: " << fooPtr.use_count() << '\n';  // 3

}  // All destroyed when last shared_ptr goes out of scope
```

4.3.7 Thread Safety Considerations

While `std::shared_ptr` ensures that reference counting is thread-safe, accessing the underlying resource is not inherently thread-safe. If multiple threads need to access the resource, you must use additional synchronization mechanisms.

Important

Thread Safety Rules:

- Reference count operations are thread-safe (atomic).
- Access to the underlying object is **not** thread-safe.
- Multiple threads can safely create, copy, and destroy `shared_ptr` instances pointing to the same object.
- Concurrent reads/writes to the pointed object require external synchronization.

```cpp
#include <memory>
#include <thread>
#include <mutex>
#include <iostream>
#include <vector>

class ThreadSafeCounter {
    std::shared_ptr<int> counter;
    mutable std::mutex mtx;

public:
    ThreadSafeCounter() : counter(std::make_shared<int>(0)) {}

    void increment() {
        std::lock_guard<std::mutex> lock(mtx);
        ++(*counter);
    }

    int get() const {
        std::lock_guard<std::mutex> lock(mtx);
        return *counter;
    }

    // Return a shared_ptr that can be safely copied to other threads
    std::shared_ptr<int> getSharedPtr() const {
        std::lock_guard<std::mutex> lock(mtx);
        return counter;  // shared_ptr copy is thread-safe
    }
};
```

```
void threadFunction(std::shared_ptr<int> ptr, int iterations) {
    for (int i = 0; i < iterations; ++i) {
        // Access to the pointed object is NOT thread-safe!
        // Need external synchronization if modifying
        // This example is unsafe without additional locks
    }
}

void safeThreadFunction(ThreadSafeCounter& counter, int iterations) {
    for (int i = 0; i < iterations; ++i) {
        counter.increment();  // Thread-safe through internal mutex
    }
}

void threadSafetyExample() {
    ThreadSafeCounter counter;

    std::vector<std::thread> threads;
    for (int i = 0; i < 10; ++i) {
        threads.emplace_back(safeThreadFunction, std::ref(counter), 10000);
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Final count: " << counter.get() << '\n';  // 100000
}
```

4.3.8 Using `std::weak_ptr` for Caching

`std::weak_ptr` is also useful in caching mechanisms where the cached object should not prevent its deletion. For example, you can use `weak_ptr` to store cached resources and convert them to `shared_ptr` when needed.

```
#include <memory>
#include <iostream>
#include <unordered_map>
#include <string>
```

```
class Resource {
public:
    int id;
    std::string data;

    Resource(int i, std::string d) : id(i), data(std::move(d)) {
        std::cout << "Resource " << id << " created\n";
    }

    ~Resource() {
        std::cout << "Resource " << id << " destroyed\n";
    }
};

class ResourceCache {
    std::unordered_map<int, std::weak_ptr<Resource>> cache;

public:
    std::shared_ptr<Resource> getResource(int id, const std::string& data) {
        auto it = cache.find(id);
        if (it != cache.end()) {
            if (auto resource = it->second.lock()) {
                std::cout << "Resource " << id << " found in cache\n";
                return resource;  // Return existing resource
            }
        }

        // Create new resource and store weak_ptr in cache
        auto resource = std::make_shared<Resource>(id, data);
        cache[id] = resource;
        std::cout << "Resource " << id << " created and cached\n";
        return resource;
    }

    void cleanup() {
        // Remove expired entries
        for (auto it = cache.begin(); it != cache.end();) {
            if (it->second.expired()) {
                std::cout << "Removing expired entry for resource " << it->first
                ↪  << '\n';
                it = cache.erase(it);
            } else {
```

```
                ++it;
            }
        }
    }

    size_t size() const { return cache.size(); }
};

void cachingExample() {
    ResourceCache cache;

    {
        auto res1 = cache.getResource(1, "Data for resource 1");
        auto res2 = cache.getResource(2, "Data for resource 2");

        std::cout << "Cache size: " << cache.size() << '\n';

        auto res1Again = cache.getResource(1, "Data for resource 1");  // From
        ↪  cache
    }  // res1, res2, res1Again destroyed

    cache.cleanup();  // Remove expired entries
    std::cout << "Cache size after cleanup: " << cache.size() << '\n';
}
```

4.3.9 Modern C++ Enhancements for `std::shared_ptr`

1. **C++14: `std::make_shared` for arrays** (improved in C++20):

   ```
   // C++20: make_shared for arrays
   auto arr = std::make_shared<int[]>(10);  // C++20
   arr[0] = 42;
   ```

2. **C++20: `std::make_shared` for arrays with custom deleters**:

   ```
   // C++20: array support with make_shared
   auto arr = std::make_shared<int[]>(10);  // Uses delete[]
   ```

3. **C++23: `std::make_shared_for_overwrite`**:

   ```
   // C++23: Default-initialized (uninitialized) - faster for large arrays
   auto arr = std::make_shared_for_overwrite<int[]>(1000000);
   std::fill_n(arr.get(), 1000000, 42);
   ```

4. **C++23: `std::atomic<std::shared_ptr>`**:

```
#include <atomic>

// C++20: atomic operations on shared_ptr
std::atomic<std::shared_ptr<int>> atomicPtr;

void atomicSharedPtrExample() {
    auto ptr = std::make_shared<int>(42);
    atomicPtr.store(ptr);  // Atomic store
    auto loaded = atomicPtr.load();  // Atomic load
}
```

4.3.10 C++26 Preview: `std::observer_ptr` for Non-Owning Observation

The proposed C++26 standard introduces `std::observer_ptr` for clear non-owning semantics, complementing `std::weak_ptr` for cases where you don't need to break cycles.

```
// Proposed C++26
#include <observer_ptr>

class Document {
    std::shared_ptr<Data> data;

public:
    // Non-owning observer parameter
    void process(std::observer_ptr<const Data> obs) {
        if (obs) {
            // Use data without ownership concerns
        }
    }

    std::observer_ptr<Data> getData() {
        return std::observer_ptr(data.get());  // Clear: non-owning
    }
};
```

4.3.11 Performance Considerations

Aspect	Impact
Control Block	Separate allocation unless using `make_shared`
Reference Counting	Atomic operations add overhead
Copy Operations	Increment/decrement atomic counter
Move Operations	No atomic operations, faster than copy
Destruction	Atomic decrement, may trigger deletion

Important

Performance Tip: Use `std::make_shared` when possible to combine the object and control block in a single allocation, reducing memory overhead and improving cache locality.

4.3.12 Best Practices for `std::shared_ptr`

1. **Prefer `std::make_shared`**:
   - Single allocation for object and control block.
   - Exception-safe and more efficient.
2. **Use `shared_ptr` Only When Shared Ownership is Needed**:
   - Default to `unique_ptr` for exclusive ownership.
   - `shared_ptr` adds overhead you shouldn't pay unnecessarily.
3. **Break Cycles with `std::weak_ptr`**:
   - Always use `weak_ptr` for back references in cyclic structures.
4. **Use `weak_ptr` for Caching**:
   - Allows resources to be reclaimed when no longer in active use.
5. **Avoid Creating `shared_ptr` from Raw Pointers**:
   - Use `make_shared` instead of `shared_ptr(new T(...))`.
   - Multiple `shared_ptr` from same raw pointer leads to double deletion.

6. **Be Aware of Thread Safety Boundaries**:

   - Reference counting is thread-safe; object access is not.
   - Use external synchronization for object access across threads.

4.3.13 Common Pitfalls

1. **Creating Multiple `shared_ptr` from Same Raw Pointer**:

```
int* raw = new int(42);
std::shared_ptr<int> p1(raw);
std::shared_ptr<int> p2(raw);  // Double deletion! Each thinks it owns the
↪  pointer
```

   Solution: Always use `make_shared` or copy from existing `shared_ptr`.

2. **Circular References**:

```
struct Node {
    std::shared_ptr<Node> next;
    std::shared_ptr<Node> prev;  // Cycle!
};
```

   Solution: Use `std::weak_ptr` for one direction.

3. **Using `shared_ptr` with Arrays (Pre-C++20)**:

```
// Pre-C++20: This uses delete, not delete[] - undefined behavior!
std::shared_ptr<int> arr(new int[10]);
```

   Solution: Use `std::make_shared<int[]>(10)` (C++20) or custom deleter.

4. **Performance Overhead in Hot Paths**:

   - `shared_ptr` copy involves atomic operations.
   - For performance-critical code, prefer `unique_ptr` or raw pointers for non-owning access.

4.3.14 Summary

- `std::shared_ptr` provides shared ownership with reference counting.
- Multiple `shared_ptr` instances can manage the same resource; cleanup occurs when the last reference is destroyed.
- `std::weak_ptr` breaks circular references and provides non-owning observation.
- Custom deleters, aliasing constructors, and thread-safe reference counting are advanced features.
- C++20 added array support for `make_shared`; C++23 added `make_shared_for_overwrite` and `atomic<shared_ptr>`.
- C++26 will introduce `std::observer_ptr` for clear non-owning semantics.
- Best practices: prefer `make_shared`, use `weak_ptr` for cycles, avoid creating multiple `shared_ptr` from raw pointers, and be mindful of thread safety and performance overhead.

Exercise 13. ***Practice Exercises:***

1. *Create a directed graph where nodes use `std::shared_ptr` for outgoing edges and `std::weak_ptr` for incoming edges to prevent cycles.*
2. *Implement a thread-safe shared resource using `std::shared_ptr` and `std::mutex` that can be safely accessed from multiple threads.*
3. *Design a cache system using `std::weak_ptr` that automatically removes expired entries.*
4. *Using the aliasing constructor, create a `shared_ptr` to a member of a structure that shares ownership with the containing object.*
5. *Research `std::atomic<std::shared_ptr>` (C++20/23) and write an example demonstrating lock-free shared pointer operations.*
6. *Compare the performance of `std::make_shared` vs `std::shared_ptr(new T(...))` with a large number of allocations.*

Chapter 5

Pointers and Object-Oriented Programming

5.1 Pointers and Polymorphism

Polymorphism is one of the core principles of object-oriented programming (OOP). It allows objects of different types to be treated as objects of a common base type, enabling flexible and reusable code. In C++, pointers play a crucial role in implementing polymorphism, particularly runtime polymorphism. This section explores the use of pointers in polymorphism, including base and derived class pointers, virtual functions, and the vtable mechanism. We will also provide practical examples to demonstrate these concepts, along with advanced topics such as multiple inheritance, virtual inheritance, and modern C++ practices for polymorphic code.

Important

Modern C++ Principle: Prefer smart pointers (`std::unique_ptr` and `std::shared_ptr`) over raw pointers for managing polymorphic objects. They ensure proper cleanup and exception safety.

5.1.1 Base and Derived Class Pointers

In C++, a pointer to a base class can point to an object of a derived class. This is a fundamental feature that enables polymorphism. By using base class pointers, you can write code that works with any derived class, providing flexibility and extensibility.

Important

Key Concepts:

- **Upcasting**: Converting a derived class pointer to a base class pointer is implicit and safe.
- **Downcasting**: Converting a base class pointer to a derived class pointer requires explicit casting (`static_cast` or `dynamic_cast`) and must be done carefully.
- **Object Slicing**: Assigning a derived class object to a base class variable (not pointer) slices off the derived parts—always use pointers or references for polymorphism.

Example: Base and Derived Class Pointers

```
#include <iostream>
#include <memory>

class Base {
public:
    void show() {
        std::cout << "Base class show()\n";
    }

    virtual ~Base() = default;  // Virtual destructor essential for polymorphism
};

class Derived : public Base {
public:
    void show() {
        std::cout << "Derived class show()\n";
    }

    void derivedOnly() {
        std::cout << "Derived-specific function\n";
```

```
    }
};

void baseAndDerivedPointersExample() {
    Derived derivedObj;

    // Upcasting: Base pointer to Derived object (implicit, safe)
    Base* basePtr = &derivedObj;
    basePtr->show();  // Calls Base::show() (non-virtual)

    // Upcasting with smart pointers
    std::unique_ptr<Base> smartPtr = std::make_unique<Derived>();
    smartPtr->show();  // Calls Base::show()

    // Downcasting: Base pointer to Derived pointer (requires cast)
    Base* basePtr2 = &derivedObj;
    Derived* derivedPtr = static_cast<Derived*>(basePtr2);  // Safe if we know the
    ↪  type
    derivedPtr->derivedOnly();

    // Runtime-checked downcasting with dynamic_cast
    Base* basePtr3 = new Derived();
    if (Derived* derivedPtr3 = dynamic_cast<Derived*>(basePtr3)) {
        derivedPtr3->derivedOnly();  // Safe: checks type at runtime
    }
    delete basePtr3;
}
```

5.1.2 Runtime Polymorphism with Virtual Functions

Runtime polymorphism is achieved in C++ using **virtual functions**. When a base class pointer points to a derived class object and a virtual function is called, the derived class's version of the function is executed. This is known as **dynamic binding** or **late binding**.

Important

Virtual Function Rules:

- Declare a function as `virtual` in the base class to enable runtime polymorphism.
- Use `override` in the derived class to explicitly indicate overriding (C++11).
- Use `final` to prevent further overriding (C++11).
- A virtual destructor is essential in base classes to ensure proper cleanup of derived objects.

Example: Runtime Polymorphism

```
#include <iostream>
#include <memory>
#include <vector>

class Shape {
public:
    virtual void draw() const {
        std::cout << "Drawing a generic shape\n";
    }

    virtual double area() const = 0;  // Pure virtual function

    virtual ~Shape() = default;  // Virtual destructor
};

class Circle : public Shape {
    double radius;
public:
    explicit Circle(double r) : radius(r) {}

    void draw() const override {
        std::cout << "Drawing a circle with radius " << radius << '\n';
    }

    double area() const override {
        return 3.14159 * radius * radius;
    }
```

```
};

class Square : public Shape {
    double side;
public:
    explicit Square(double s) : side(s) {}

    void draw() const override {
        std::cout << "Drawing a square with side " << side << '\n';
    }

    double area() const override {
        return side * side;
    }
};

class Triangle final : public Shape {  // final: cannot be further derived
    double base, height;
public:
    Triangle(double b, double h) : base(b), height(h) {}

    void draw() const override {
        std::cout << "Drawing a triangle with base " << base
                  << " and height " << height << '\n';
    }

    double area() const override {
        return 0.5 * base * height;
    }
};

void runtimePolymorphismExample() {
    // Modern C++: using smart pointers
    std::vector<std::unique_ptr<Shape>> shapes;
    shapes.push_back(std::make_unique<Circle>(5.0));
    shapes.push_back(std::make_unique<Square>(4.0));
    shapes.push_back(std::make_unique<Triangle>(3.0, 6.0));

    for (const auto& shape : shapes) {
        shape->draw();  // Calls the appropriate draw() based on actual type
        std::cout << "Area: " << shape->area() << '\n';
    }
```

```
}  // All shapes automatically destroyed
```

5.1.3 Virtual Functions and the Vtable Mechanism

Virtual functions are implemented in C++ using a mechanism called the **vtable** (virtual table). The vtable is a table of function pointers created for each class with virtual functions. It allows the correct function to be called at runtime based on the actual object type.

Important

Vtable Implementation Details:

- Each class with virtual functions has a vtable (static) containing pointers to its virtual functions.
- Each object contains a hidden pointer (vptr) to its class's vtable.
- When a virtual function is called, the vptr is used to look up the correct function in the vtable.
- This indirection adds a small runtime overhead (one pointer dereference per virtual call).
- Multiple inheritance creates more complex vtable structures.

Conceptual Example: Understanding Vtable

```
#include <iostream>

class Base {
public:
    virtual void func1() { std::cout << "Base::func1()\n"; }
    virtual void func2() { std::cout << "Base::func2()\n"; }
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    void func1() override { std::cout << "Derived::func1()\n"; }
    void func2() override { std::cout << "Derived::func2()\n"; }
```

```
    void derivedOnly() { std::cout << "Derived::derivedOnly()\n"; }
};

void vtableConceptExample() {
    Base* basePtr = new Derived();

    // Virtual function calls:
    basePtr->func1();  // Calls Derived::func1() - resolved via vtable
    basePtr->func2();  // Calls Derived::func2() - resolved via vtable

    // Under the hood (conceptual):
    // 1. Access vptr from object
    // 2. Look up function pointer in vtable at appropriate index
    // 3. Call through function pointer

    delete basePtr;
}
```

5.1.4 Simulating the Vtable Mechanism

To better understand the vtable mechanism, let's simulate how it works using function pointers.

```
#include <iostream>
#include <memory>

// Simulating vtable with function pointers
class Base {
public:
    using FuncPtr = void(*)();  // Function pointer type

    // Simulated vtable: array of function pointers
    static FuncPtr vtable[2];

    // Simulated vptr: pointer to vtable
    FuncPtr* vptr;

    Base() : vptr(vtable) {}

    void callFunc1() { vptr[0](); }
    void callFunc2() { vptr[1](); }

    virtual ~Base() = default;
```

```
};

// Base's vtable
void BaseFunc1() { std::cout << "Base::func1()\n"; }
void BaseFunc2() { std::cout << "Base::func2()\n"; }
Base::FuncPtr Base::vtable[2] = {BaseFunc1, BaseFunc2};

class Derived : public Base {
public:
    static FuncPtr vtable[2];

    Derived() {
        vptr = vtable;  // Override vptr to point to Derived's vtable
    }
};

// Derived's vtable
void DerivedFunc1() { std::cout << "Derived::func1()\n"; }
void DerivedFunc2() { std::cout << "Derived::func2()\n"; }
Base::FuncPtr Derived::vtable[2] = {DerivedFunc1, DerivedFunc2};

void simulateVtableExample() {
    Base* basePtr = new Derived();
    basePtr->callFunc1();  // Calls DerivedFunc1()
    basePtr->callFunc2();  // Calls DerivedFunc2()
    delete basePtr;
}
```

5.1.5 Modern C++ Practices for Polymorphic Code

5.1.5.1 Using Smart Pointers for Polymorphic Objects

Modern C++ strongly recommends using smart pointers instead of raw pointers for managing polymorphic objects.

```
#include <memory>
#include <vector>
#include <iostream>

class Widget {
public:
    virtual void render() const = 0;
```

```
    virtual ~Widget() = default;
};

class Button : public Widget {
public:
    void render() const override {
        std::cout << "Rendering button\n";
    }
};

class TextBox : public Widget {
public:
    void render() const override {
        std::cout << "Rendering text box\n";
    }
};

class Window {
    std::vector<std::unique_ptr<Widget>> widgets;

public:
    void addWidget(std::unique_ptr<Widget> widget) {
        widgets.push_back(std::move(widget));
    }

    void render() const {
        for (const auto& widget : widgets) {
            widget->render();  // Polymorphic call
        }
    }
};

void modernPolymorphismExample() {
    Window window;
    window.addWidget(std::make_unique<Button>());
    window.addWidget(std::make_unique<TextBox>());
    window.render();
}
```

5.1.5.2 Factory Pattern with Smart Pointers

Factories are a common pattern for creating polymorphic objects.

```cpp
#include <memory>
#include <string>
#include <unordered_map>
#include <functional>

class Document {
public:
    virtual void open() = 0;
    virtual void save() = 0;
    virtual ~Document() = default;
};

class PDFDocument : public Document {
public:
    void open() override { std::cout << "Opening PDF\n"; }
    void save() override { std::cout << "Saving PDF\n"; }
};

class WordDocument : public Document {
public:
    void open() override { std::cout << "Opening Word document\n"; }
    void save() override { std::cout << "Saving Word document\n"; }
};

class DocumentFactory {
    using Creator = std::function<std::unique_ptr<Document>()>;
    std::unordered_map<std::string, Creator> creators;

public:
    void registerType(const std::string& type, Creator creator) {
        creators[type] = std::move(creator);
    }

    std::unique_ptr<Document> create(const std::string& type) {
        auto it = creators.find(type);
        if (it != creators.end()) {
            return it->second();
        }
        return nullptr;
    }
};
```

```
void factoryPatternExample() {
    DocumentFactory factory;
    factory.registerType("pdf", []() { return std::make_unique<PDFDocument>(); });
    factory.registerType("word", []() { return std::make_unique<WordDocument>();
    ↪  });

    auto doc = factory.create("pdf");
    if (doc) {
        doc->open();
        doc->save();
    }
}
```

5.1.6 Multiple Inheritance and Virtual Functions

In multiple inheritance, a class can inherit from more than one base class. This can lead to complex vtable structures and potential ambiguity.

```
#include <iostream>

class Drawable {
public:
    virtual void draw() = 0;
    virtual ~Drawable() = default;
};

class Resizable {
public:
    virtual void resize(int factor) = 0;
    virtual ~Resizable() = default;
};

class Rectangle : public Drawable, public Resizable {
    int width, height;

public:
    Rectangle(int w, int h) : width(w), height(h) {}

    void draw() override {
        std::cout << "Drawing rectangle " << width << "x" << height << '\n';
    }
```

```
    void resize(int factor) override {
        width *= factor;
        height *= factor;
        std::cout << "Resized to " << width << "x" << height << '\n';
    }
};

void multipleInheritanceExample() {
    Rectangle rect(10, 20);

    Drawable* drawPtr = &rect;
    Resizable* resizePtr = &rect;

    drawPtr->draw();      // Calls Rectangle::draw()
    resizePtr->resize(2); // Calls Rectangle::resize()

    // Note: The vptr mechanism handles different base class vtables
    // The object contains multiple vptrs (one for each base class)
}
```

5.1.7 Virtual Inheritance

Virtual inheritance resolves the ”diamond problem” where a class inherits from two classes that both inherit from a common base class.

```
#include <iostream>

class Animal {
public:
    virtual void speak() = 0;
    virtual ~Animal() = default;
};

class Mammal : virtual public Animal {
public:
    void speak() override { std::cout << "Mammal sound\n"; }
};

class Winged : virtual public Animal {
public:
    void speak() override { std::cout << "Winged creature sound\n"; }
};
```

```
class Bat : public Mammal, public Winged {
public:
    void speak() override {
        std::cout << "Bat screeches\n";
    }
};

void virtualInheritanceExample() {
    Bat bat;
    Animal* animalPtr = &bat;

    animalPtr->speak();  // Calls Bat::speak()

    // Without virtual inheritance, there would be ambiguity
    // With virtual inheritance, only one Animal subobject exists
}
```

5.1.8 Performance Considerations

Aspect	Impact
Virtual function call	One extra pointer dereference + indirect call
Vtable	One vptr per object, one vtable per class with virtual functions
Multiple inheritance	Multiple vptrs per object
Virtual inheritance	Additional indirection for base class access

Important

Performance Tips:

- Virtual functions add minimal overhead (typically 1-2 additional instructions).
- Use `final` on classes that won't be derived from to enable devirtualization optimizations.
- For performance-critical code, consider compile-time polymorphism with templates (CRTP) as an alternative.

5.1.9 Common Pitfalls

1. **Object Slicing**:

```
Derived derived;
Base base = derived;  // Slicing! Derived parts lost
```

Solution: Use pointers or references.

2. **Non-Virtual Destructor**:

```
class Base { ~Base() {} };  // Non-virtual destructor
Base* ptr = new Derived();
delete ptr;  // Undefined behavior - Derived destructor not called
```

Solution: Always declare virtual destructor in polymorphic base classes.

3. **Downcasting Without Type Check**:

```
Base* ptr = new Base();
Derived* derived = static_cast<Derived*>(ptr);  // Unsafe!
```

Solution: Use `dynamic_cast` for runtime-checked downcasting.

4. **Calling Virtual Functions in Constructor/Destructor**:

```
class Base {
public:
    Base() { virtualFunc(); }  // Calls Base::virtualFunc(), not Derived's!
};
```

Solution: Avoid virtual calls in constructors/destructors.

5. **Circular References with Shared Pointers**:

```
struct Node {
    std::shared_ptr<Node> parent;
    std::shared_ptr<Node> child;  // Potential cycle
};
```

Solution: Use `std::weak_ptr` for back references.

5.1.10 C++23 and C++26 Enhancements

C++23: `std::out_ptr` for C API Interoperability

```
// Safely interface with C APIs that expect raw pointers
extern "C" {
    struct Widget;
    void create_widget(Widget** out);
    void destroy_widget(Widget*);
}

std::unique_ptr<Widget, decltype(&destroy_widget)>
    createWidget() {
    std::unique_ptr<Widget, decltype(&destroy_widget)> ptr(nullptr,
    ↪  destroy_widget);
    create_widget(std::out_ptr(ptr));  // C++23
    return ptr;
}
```

C++26 Preview: `std::observer_ptr` for Non-Owning Polymorphic References

```
// Proposed C++26
#include <observer_ptr>

void processWidget(std::observer_ptr<Widget> widget) {
    if (widget) {
        widget->render();  // Non-owning observation
    }
}
```

5.1.11 Summary

- Base class pointers can point to derived class objects, enabling polymorphism (upcasting).
- Virtual functions enable runtime polymorphism through dynamic dispatch.
- The vtable mechanism provides the implementation: each object has a vptr to its class's vtable.
- Modern C++ practices: prefer smart pointers over raw pointers, use `override` and `final`, and always declare virtual destructors.
- Multiple inheritance and virtual inheritance introduce additional complexity in vtable structures.

- Performance overhead of virtual functions is minimal; use `final` to enable devirtualization.
- C++23 adds `std::out_ptr` for C API interoperability; C++26 will introduce `std::observer_ptr` for non-owning semantics.

Exercise 14. ***Practice Exercises:***

1. *Create a hierarchy of geometric shapes with virtual functions for area and perimeter. Use smart pointers to manage them in a container.*
2. *Implement a simple GUI framework with a `Widget` base class and derived classes (`Button`, `Label`, `TextBox`). Use a factory pattern with `std::unique_ptr`.*
3. *Explore the vtable layout using compiler-specific extensions (e.g., `-fdump-class-hierarchy` in GCC) for a class with multiple virtual functions.*
4. *Implement a safe downcasting mechanism using `dynamic_cast` and demonstrate its runtime type checking.*
5. *Create a class hierarchy that demonstrates virtual inheritance to resolve the diamond problem. Show how the virtual function calls are resolved.*
6. *Using C++23's `std::out_ptr`, create a wrapper for a C library that creates polymorphic objects.*

5.2 Pointers and Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit properties and behaviors from another class. Pointers play a crucial role in working with inheritance, particularly when dealing with base and derived class objects. This section explores how to access derived class objects via base class pointers and demonstrates the use of `dynamic_cast` for safe downcasting. We will also cover best practices, potential pitfalls, and advanced topics such as virtual inheritance, multiple inheritance, and the role of pointers in polymorphism.

Important

Core Principle: A pointer to a base class can point to any object of a class derived from that base. This is the foundation of runtime polymorphism in C++.

5.2.1 Accessing Derived Class Objects via Base Class Pointers

In C++, a pointer to a base class can point to an object of a derived class. This powerful feature enables polymorphism and allows you to write flexible and reusable code. However, accessing derived class members through a base class pointer requires careful handling, as the base class pointer only provides access to the base class interface.

Important

Key Concepts:

- **Upcasting**: Converting a derived class pointer to a base class pointer is implicit and safe. It's called upcasting because you move *up* the inheritance hierarchy.
- **Downcasting**: Converting a base class pointer to a derived class pointer is explicit and requires casting. It's called downcasting because you move *down* the inheritance hierarchy.
- **Object Slicing**: Assigning a derived class object to a base class variable (not pointer) slices off the derived parts—always use pointers or references for polymorphism.

Example: Base and Derived Class Pointers

```
#include <iostream>
```

```cpp
#include <memory>

class Base {
public:
    void nonVirtualShow() {
        std::cout << "Base::nonVirtualShow()\n";
    }

    virtual void virtualShow() {
        std::cout << "Base::virtualShow()\n";
    }

    virtual ~Base() = default;  // Virtual destructor essential
};

class Derived : public Base {
public:
    void nonVirtualShow() {
        std::cout << "Derived::nonVirtualShow()\n";
    }

    void virtualShow() override {
        std::cout << "Derived::virtualShow()\n";
    }

    void derivedOnly() {
        std::cout << "Derived::derivedOnly()\n";
    }
};

void accessingDerivedObjectsExample() {
    Derived derivedObj;
    Base* basePtr = &derivedObj;  // Upcasting: implicit and safe

    // Non-virtual function: resolved at compile-time based on pointer type
    basePtr->nonVirtualShow();  // Calls Base::nonVirtualShow()

    // Virtual function: resolved at runtime based on object type
    basePtr->virtualShow();     // Calls Derived::virtualShow()

    // basePtr->derivedOnly();   // Error: Base pointer cannot access derived
    ↪  members
```

```
    // Modern C++: using smart pointers
    std::unique_ptr<Base> smartPtr = std::make_unique<Derived>();
    smartPtr->virtualShow();     // Calls Derived::virtualShow()
}
```

5.2.2 Downcasting with `static_cast` and `dynamic_cast`

To access derived class members through a base class pointer, you need to downcast. C++ provides two main casting operators for downcasting:

1. **`static_cast`**:
    - Compile-time cast with no runtime type checking.
    - Fast but unsafe if the actual object type is not the target type.
    - Use only when you are absolutely certain of the object's type.
2. **`dynamic_cast`**:
    - Runtime type-checked cast that returns `nullptr` for pointers or throws `std::bad_cast` for references on failure.
    - Requires the base class to be polymorphic (have at least one virtual function).
    - Safe but has a small runtime overhead.

Example: `static_cast` vs `dynamic_cast`

```
#include <iostream>
#include <memory>
#include <typeinfo>

class Base {
public:
    virtual void show() { std::cout << "Base::show()\n"; }
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    void show() override { std::cout << "Derived::show()\n"; }
```

```
    void derivedFunction() { std::cout << "Derived::derivedFunction()\n"; }
};

class Other : public Base {
public:
    void show() override { std::cout << "Other::show()\n"; }
    void otherFunction() { std::cout << "Other::otherFunction()\n"; }
};

void downcastingExample() {
    // Safe case: basePtr actually points to a Derived object
    Base* basePtr = new Derived();

    // static_cast: fast but no runtime check
    Derived* derivedPtr1 = static_cast<Derived*>(basePtr);
    derivedPtr1->derivedFunction();  // Works correctly

    // dynamic_cast: runtime-checked, safe
    Derived* derivedPtr2 = dynamic_cast<Derived*>(basePtr);
    if (derivedPtr2) {
        derivedPtr2->derivedFunction();  // Works correctly
    }

    // Dangerous case: basePtr2 points to Other, not Derived
    Base* basePtr2 = new Other();

    // static_cast: compiles but leads to undefined behavior!
    Derived* derivedPtr3 = static_cast<Derived*>(basePtr2);
    derivedPtr3->derivedFunction();  // UNDEFINED BEHAVIOR - crash likely!

    // dynamic_cast: safe, returns nullptr
    Derived* derivedPtr4 = dynamic_cast<Derived*>(basePtr2);
    if (derivedPtr4) {
        derivedPtr4->derivedFunction();  // This block is skipped
    } else {
        std::cout << "dynamic_cast failed: not a Derived object\n";
    }

    delete basePtr;
    delete basePtr2;
}
```

5.2.3 Handling Invalid Casts

When using `dynamic_cast`, it's essential to handle cases where the cast fails.

```
#include <iostream>
#include <memory>
#include <typeinfo>

class Shape {
public:
    virtual void draw() const = 0;
    virtual ~Shape() = default;
};

class Circle : public Shape {
public:
    void draw() const override { std::cout << "Drawing circle\n"; }
    void setRadius(double r) { radius = r; }
private:
    double radius{0};
};

class Square : public Shape {
public:
    void draw() const override { std::cout << "Drawing square\n"; }
    void setSide(double s) { side = s; }
private:
    double side{0};
};

void handleInvalidCastsExample() {
    // Using pointers
    Shape* shapePtr = new Square();

    // Attempt to cast to Circle - will fail
    Circle* circlePtr = dynamic_cast<Circle*>(shapePtr);
    if (circlePtr) {
        circlePtr->setRadius(5.0);  // Not executed
        std::cout << "Cast successful\n";
    } else {
        std::cout << "Cast failed: shapePtr is not a Circle\n";
    }
```

```
    // Using references - throws exception on failure
    Square square;
    Shape& shapeRef = square;

    try {
        Circle& circleRef = dynamic_cast<Circle&>(shapeRef);
        circleRef.setRadius(5.0);  // Never reached
    } catch (const std::bad_cast& e) {
        std::cout << "Exception: " << e.what() << '\n';
    }

    delete shapePtr;
}
```

5.2.4 Multiple Inheritance and `dynamic_cast`

In multiple inheritance, `dynamic_cast` can cast across inheritance hierarchies, not just up and down.

```
#include <iostream>
#include <memory>

class Drawable {
public:
    virtual void draw() = 0;
    virtual ~Drawable() = default;
};

class Serializable {
public:
    virtual void serialize() = 0;
    virtual ~Serializable() = default;
};

class Widget : public Drawable, public Serializable {
public:
    void draw() override {
        std::cout << "Drawing widget\n";
    }

    void serialize() override {
```

```
        std::cout << "Serializing widget\n";
    }

    void widgetOnly() {
        std::cout << "Widget-specific function\n";
    }
};

void multipleInheritanceDynamicCastExample() {
    Widget widget;

    // Cast from Drawable* to Widget*
    Drawable* drawPtr = &widget;
    Widget* widgetPtr = dynamic_cast<Widget*>(drawPtr);
    if (widgetPtr) {
        widgetPtr->widgetOnly();  // Works
    }

    // Cast across inheritance hierarchies: Drawable* to Serializable*
    Serializable* serialPtr = dynamic_cast<Serializable*>(drawPtr);
    if (serialPtr) {
        serialPtr->serialize();  // Works - dynamic_cast can cross hierarchies
    }

    // Demonstration of cross-casting
    std::cout << "Addresses:\n";
    std::cout << "  Widget:     " << &widget << '\n';
    std::cout << "  Drawable:   " << drawPtr << '\n';
    std::cout << "  Serializable: " << serialPtr << '\n';
    // Note: The addresses may be different due to multiple inheritance layout
}
```

5.2.5 Virtual Inheritance and `dynamic_cast`

Virtual inheritance resolves the ”diamond problem” where a class inherits from two classes that both inherit from a common base class. `dynamic_cast` works correctly with virtual inheritance.

```
#include <iostream>
#include <memory>

class Animal {
public:
```

```
    virtual void speak() { std::cout << "Animal sound\n"; }
    virtual ~Animal() = default;
};

class Mammal : virtual public Animal {
public:
    void speak() override { std::cout << "Mammal sound\n"; }
};

class Winged : virtual public Animal {
public:
    void speak() override { std::cout << "Winged sound\n"; }
};

class Bat : public Mammal, public Winged {
public:
    void speak() override {
        std::cout << "Bat screeches\n";
    }

    void echolocate() {
        std::cout << "Bat echolocating\n";
    }
};

void virtualInheritanceDynamicCastExample() {
    Bat bat;

    // Upcasting to any base class works
    Animal* animalPtr = &bat;
    Mammal* mammalPtr = &bat;
    Winged* wingedPtr = &bat;

    // Downcasting from Animal to Bat
    Bat* batPtr1 = dynamic_cast<Bat*>(animalPtr);
    if (batPtr1) {
        batPtr1->echolocate();  // Works
    }

    // Cross-casting between siblings
    // Cast from Mammal* to Winged* (both share Animal base)
    Winged* wingedFromMammal = dynamic_cast<Winged*>(mammalPtr);
```

```
    if (wingedFromMammal) {
        std::cout << "Cross-cast successful\n";
        wingedFromMammal->speak();  // Calls Bat::speak()
    }

    // With virtual inheritance, there is only one Animal subobject
    std::cout << "Addresses:\n";
    std::cout << "  Bat:     " << &bat << '\n';
    std::cout << "  Animal:  " << animalPtr << '\n';
    std::cout << "  Mammal:  " << mammalPtr << '\n';
    std::cout << "  Winged:  " << wingedPtr << '\n';
}
```

5.2.6 Modern C++ Practices with Smart Pointers and Inheritance

Modern C++ encourages using smart pointers instead of raw pointers for managing polymorphic objects.

```
#include <memory>
#include <vector>
#include <iostream>

class Logger {
public:
    virtual void log(const std::string& msg) = 0;
    virtual ~Logger() = default;
};

class ConsoleLogger : public Logger {
public:
    void log(const std::string& msg) override {
        std::cout << "[Console] " << msg << '\n';
    }
};

class FileLogger : public Logger {
    std::string filename;
public:
    explicit FileLogger(std::string name) : filename(std::move(name)) {}

    void log(const std::string& msg) override {
        std::cout << "[File:" << filename << "] " << msg << '\n';
```

```
    }
};

class LoggerManager {
    std::vector<std::unique_ptr<Logger>> loggers;

public:
    void addLogger(std::unique_ptr<Logger> logger) {
        loggers.push_back(std::move(logger));
    }

    void logAll(const std::string& msg) {
        for (const auto& logger : loggers) {
            logger->log(msg);
        }
    }

    // Safe downcasting with smart pointers
    ConsoleLogger* getConsoleLogger() {
        for (auto& logger : loggers) {
            // Use get() to obtain raw pointer for dynamic_cast
            if (auto console = dynamic_cast<ConsoleLogger*>(logger.get())) {
                return console;
            }
        }
        return nullptr;
    }
};

void smartPointerInheritanceExample() {
    LoggerManager manager;
    manager.addLogger(std::make_unique<ConsoleLogger>());
    manager.addLogger(std::make_unique<FileLogger>("app.log"));

    manager.logAll("Application started");

    if (auto console = manager.getConsoleLogger()) {
        console->log("Direct console message");
    }
}
```

5.2.7 Best Practices for Pointers and Inheritance

1. **Use `dynamic_cast` for Safe Downcasting**:
   - Always use `dynamic_cast` when downcasting polymorphic types.
   - Handle invalid casts gracefully by checking for `nullptr` (for pointers) or catching `std::bad_cast` (for references).
2. **Avoid C-Style Casts**:
   - C-style casts (e.g., `(Derived*)basePtr`) are unsafe and should be avoided.
   - Use C++ casting operators: `static_cast`, `dynamic_cast`, `reinterpret_cast`, and `const_cast`.
3. **Prefer Polymorphism Over Downcasting**:
   - Use virtual functions and polymorphism to avoid the need for downcasting whenever possible.
   - Frequent downcasting can indicate a design flaw—consider refactoring.
4. **Declare Virtual Destructors in Base Classes**:
   - Always declare a virtual destructor in polymorphic base classes to ensure proper cleanup of derived objects.
5. **Use `override` and `final` Keywords**:
   - Use `override` to explicitly mark overriding functions (C++11).
   - Use `final` to prevent further overriding (C++11).
6. **Prefer Smart Pointers for Ownership**:
   - Use `std::unique_ptr` and `std::shared_ptr` to manage polymorphic object lifetimes.

5.2.8 Common Pitfalls

1. **Object Slicing**:

```
Derived derived;
Base base = derived;  // Slicing! Derived parts lost
```

   Solution: Use pointers or references.

2. **Non-Virtual Destructor**:

```
class Base { ~Base() {} };  // Non-virtual
Base* ptr = new Derived();
delete ptr;  // Undefined behavior - Derived destructor not called
```

   Solution: Always declare virtual destructor in polymorphic base classes.

3. **Downcasting Without Type Check**:

```
Base* ptr = new Base();
Derived* derived = static_cast<Derived*>(ptr);  // Unsafe!
```

   Solution: Use `dynamic_cast` for runtime-checked downcasting.

4. **Calling Virtual Functions in Constructor/Destructor**:

```
class Base {
public:
    Base() { virtualFunc(); }  // Calls Base::virtualFunc(), not Derived's!
};
```

   Solution: Avoid virtual calls in constructors/destructors.

5. **Multiple Inheritance Ambiguity**:

```
class A { public: void f(); };
class B { public: void f(); };
class C : public A, public B {};  // C::f() is ambiguous
```

   Solution: Resolve with scope resolution or virtual inheritance.

5.2.9 C++23 and C++26 Enhancements

C++23: `std::out_ptr` for C API Interoperability

```
// Safely interface with C APIs that create polymorphic objects
extern "C" {
    struct Widget;
    void create_widget(Widget** out);
    void destroy_widget(Widget*);
}

std::unique_ptr<Widget, decltype(&destroy_widget)>
    createWidget() {
    std::unique_ptr<Widget, decltype(&destroy_widget)> ptr(nullptr,
    ↪  destroy_widget);
    create_widget(std::out_ptr(ptr));  // C++23
    return ptr;
}
```

C++26 Preview: `std::observer_ptr` for Non-Owning Polymorphic References

```
// Proposed C++26
#include <observer_ptr>

void processShape(std::observer_ptr<Shape> shape) {
    if (shape) {
        shape->draw();  // Non-owning observation
    }
}
```

5.2.10 Summary

- Base class pointers can point to derived class objects (upcasting), enabling polymorphism.
- Downcasting is required to access derived class members and can be done safely with `dynamic_cast`.
- `dynamic_cast` performs runtime type checking and returns `nullptr` on failure for pointers.
- Multiple inheritance and virtual inheritance are supported by `dynamic_cast`, including cross-casting.

- Modern C++ practices: use smart pointers, `override`, `final`, and always declare virtual destructors.
- Best practices: prefer polymorphism over downcasting, use `dynamic_cast` for safety, and avoid object slicing.
- C++23 adds `std::out_ptr` for C API interoperability; C++26 will introduce `std::observer_ptr` for non-owning semantics.

Exercise 15. ***Practice Exercises:***

1. *Create a class hierarchy for vehicles (`Vehicle`, `Car`, `Truck`, `Motorcycle`) with virtual functions. Demonstrate upcasting and safe downcasting.*
2. *Implement a simple plugin system where plugins are loaded via base class pointers. Use `dynamic_cast` to access plugin-specific functionality.*
3. *Create a multiple inheritance hierarchy demonstrating the diamond problem. Show how virtual inheritance resolves it and how `dynamic_cast` works across the hierarchy.*
4. *Write a function that takes a `Base*` and safely attempts to cast it to various derived types, reporting which type it actually is.*
5. *Implement a polymorphic container using `std::vector<std::unique_ptr<Base>>` and demonstrate iteration with polymorphic calls.*
6. *Using C++23's `std::out_ptr`, create a wrapper for a C library that creates and destroys polymorphic objects.*

5.3 Pointers and Abstract Classes

Abstract classes are a powerful feature of C++ that allow you to define interfaces and enforce specific behaviors in derived classes. An abstract class cannot be instantiated directly; instead, it serves as a base class for other classes. Pointers to abstract classes are particularly useful for implementing polymorphism, enabling you to write flexible and reusable code. This section explores how to use pointers to abstract classes, demonstrates how to implement an interface with pointers, and provides practical examples to illustrate these concepts. We will also cover advanced topics such as smart pointers, the factory pattern, multiple inheritance, and virtual inheritance.

Important

Core Principle: Abstract classes define contracts that derived classes must fulfill. Pointers to abstract classes enable runtime polymorphism and interface-based programming.

5.3.1 Using Pointers to Abstract Classes

An abstract class is a class that contains at least one **pure virtual function**, which is a virtual function declared with `= 0`. Pure virtual functions must be overridden in derived classes, making the class abstract and uninstantiable. Pointers to abstract classes can point to objects of derived classes, enabling runtime polymorphism.

Important

Key Concepts:

- **Abstract Class**: A class with at least one pure virtual function. Cannot be instantiated.
- **Pure Virtual Function**: A virtual function declared with `= 0`. Must be overridden in derived classes.
- **Concrete Class**: A class that overrides all pure virtual functions and can be instantiated.
- **Interface**: An abstract class with only pure virtual functions (no data members).

Example: Basic Abstract Class Usage

```
#include <iostream>
```

```cpp
#include <memory>

// Abstract class
class Shape {
public:
    virtual void draw() const = 0;       // Pure virtual function
    virtual double area() const = 0;     // Pure virtual function
    virtual ~Shape() = default;          // Virtual destructor (essential!)
};

// Concrete derived class
class Circle : public Shape {
    double radius;

public:
    explicit Circle(double r) : radius(r) {}

    void draw() const override {
        std::cout << "Drawing a circle with radius " << radius << '\n';
    }

    double area() const override {
        return 3.14159 * radius * radius;
    }
};

// Concrete derived class
class Square : public Shape {
    double side;

public:
    explicit Square(double s) : side(s) {}

    void draw() const override {
        std::cout << "Drawing a square with side " << side << '\n';
    }

    double area() const override {
        return side * side;
    }
};
```

```
void abstractClassPointersExample() {
    // Shape shape;  // Error: cannot instantiate abstract class

    Shape* shapePtr;  // Pointer to abstract class - OK

    Circle circle(5.0);
    Square square(4.0);

    shapePtr = &circle;
    shapePtr->draw();      // Calls Circle::draw()
    std::cout << "Area: " << shapePtr->area() << '\n';

    shapePtr = &square;
    shapePtr->draw();      // Calls Square::draw()
    std::cout << "Area: " << shapePtr->area() << '\n';
}
```

5.3.2 Implementing Interfaces with Abstract Classes

Abstract classes are often used to define interfaces that specify a set of methods that derived classes must implement. This is the foundation of interface-based programming in C++.

```
#include <iostream>
#include <memory>
#include <vector>

// Interface: defines what objects can do, not how they do it
class Printable {
public:
    virtual void print() const = 0;
    virtual ~Printable() = default;
};

class Serializable {
public:
    virtual void serialize(std::ostream& os) const = 0;
    virtual ~Serializable() = default;
};

// A class implementing multiple interfaces
class Document : public Printable, public Serializable {
    std::string title;
```

```
    std::string content;

public:
    Document(std::string t, std::string c)
        : title(std::move(t)), content(std::move(c)) {}

    void print() const override {
        std::cout << "Document: " << title << '\n';
        std::cout << content << '\n';
    }

    void serialize(std::ostream& os) const override {
        os << "TITLE:" << title << '\n';
        os << "CONTENT:" << content << '\n';
        os << "---\n";
    }
};

class Image : public Printable {
    std::string filename;

public:
    explicit Image(std::string f) : filename(std::move(f)) {}

    void print() const override {
        std::cout << "[Image: " << filename << "]\n";
    }
};

void interfaceImplementationExample() {
    std::vector<std::unique_ptr<Printable>> printables;

    printables.push_back(std::make_unique<Document>("Report", "Annual report
    ↪  content..."));
    printables.push_back(std::make_unique<Image>("photo.jpg"));

    for (const auto& printable : printables) {
        printable->print();
        std::cout << "---\n";
    }
}
```

5.3.3 Modern C++: Smart Pointers with Abstract Classes

Smart pointers are the recommended way to manage dynamically allocated polymorphic objects.

```
#include <memory>
#include <vector>
#include <iostream>

class Logger {
public:
    virtual void log(const std::string& message, int level) = 0;
    virtual ~Logger() = default;
};

class ConsoleLogger : public Logger {
public:
    void log(const std::string& message, int level) override {
        std::cout << "[CONSOLE] Level " << level << ": " << message << '\n';
    }
};

class FileLogger : public Logger {
    std::string filename;

public:
    explicit FileLogger(std::string f) : filename(std::move(f)) {}

    void log(const std::string& message, int level) override {
        std::cout << "[FILE:" << filename << "] Level " << level << ": " <<
        ↪  message << '\n';
    }
};

class LoggerManager {
    std::vector<std::unique_ptr<Logger>> loggers;

public:
    void addLogger(std::unique_ptr<Logger> logger) {
        loggers.push_back(std::move(logger));
    }

    void logAll(const std::string& message, int level) {
```

```
        for (const auto& logger : loggers) {
            logger->log(message, level);
        }
    }
};

void smartPointersAbstractClassExample() {
    LoggerManager manager;
    manager.addLogger(std::make_unique<ConsoleLogger>());
    manager.addLogger(std::make_unique<FileLogger>("app.log"));

    manager.logAll("Application started", 1);
    manager.logAll("Processing data...", 2);
}
```

5.3.4 Factory Pattern with Abstract Classes

The factory pattern is a creational design pattern that uses abstract classes to create objects without specifying the exact class.

```
#include <memory>
#include <iostream>
#include <unordered_map>
#include <functional>

class Shape {
public:
    virtual void draw() const = 0;
    virtual double area() const = 0;
    virtual ~Shape() = default;
};

class Circle : public Shape {
    double radius;
public:
    explicit Circle(double r) : radius(r) {}

    void draw() const override {
        std::cout << " Circle (r=" << radius << ")\n";
    }

    double area() const override {
```

```
        return 3.14159 * radius * radius;
    }
};

class Rectangle : public Shape {
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}

    void draw() const override {
        std::cout << " Rectangle (" << width << "x" << height << ")\n";
    }

    double area() const override {
        return width * height;
    }
};

// Simple factory function
std::unique_ptr<Shape> createShape(const std::string& type,
                                   const std::vector<double>& params) {
    if (type == "circle" && params.size() >= 1) {
        return std::make_unique<Circle>(params[0]);
    } else if (type == "rectangle" && params.size() >= 2) {
        return std::make_unique<Rectangle>(params[0], params[1]);
    }
    return nullptr;
}

// Factory with registration (extensible)
class ShapeFactory {
    using Creator = std::function<std::unique_ptr<Shape>(const
    ↪  std::vector<double>&)>;
    std::unordered_map<std::string, Creator> creators;

public:
    void registerShape(const std::string& name, Creator creator) {
        creators[name] = std::move(creator);
    }

    std::unique_ptr<Shape> create(const std::string& name,
                                  const std::vector<double>& params) {
```

```
        auto it = creators.find(name);
        if (it != creators.end()) {
            return it->second(params);
        }
        return nullptr;
    }
};

void factoryPatternExample() {
    // Simple factory
    auto shape1 = createShape("circle", {5.0});
    auto shape2 = createShape("rectangle", {4.0, 6.0});

    if (shape1) shape1->draw();
    if (shape2) shape2->draw();

    // Extensible factory
    ShapeFactory factory;
    factory.registerShape("circle", [](const auto& p) {
        return std::make_unique<Circle>(p[0]);
    });
    factory.registerShape("rectangle", [](const auto& p) {
        return std::make_unique<Rectangle>(p[0], p[1]);
    });

    auto shape3 = factory.create("circle", {3.0});
    if (shape3) shape3->draw();
}
```

5.3.5 Abstract Classes and Multiple Inheritance

Abstract classes can be used in multiple inheritance to define multiple interfaces that derived classes must implement.

```
#include <iostream>
#include <memory>

// Interface 1
class Drawable {
public:
    virtual void draw() const = 0;
    virtual ~Drawable() = default;
```

```cpp
};

// Interface 2
class Scalable {
public:
    virtual void scale(double factor) = 0;
    virtual ~Scalable() = default;
};

// Interface 3
class Rotatable {
public:
    virtual void rotate(double degrees) = 0;
    virtual ~Rotatable() = default;
};

// Concrete class implementing multiple interfaces
class GraphicObject : public Drawable, public Scalable, public Rotatable {
    std::string name;
    double size;
    double angle;

public:
    explicit GraphicObject(std::string n) : name(std::move(n)), size(1.0),
    ↪  angle(0.0) {}

    void draw() const override {
        std::cout << "Drawing " << name << " (size=" << size << ", angle=" <<
        ↪  angle << "°)\n";
    }

    void scale(double factor) override {
        size *= factor;
        std::cout << name << " scaled to " << size << '\n';
    }

    void rotate(double degrees) override {
        angle += degrees;
        std::cout << name << " rotated to " << angle << "°\n";
    }
};
```

```
void multipleInheritanceAbstractClassExample() {
    GraphicObject obj("Cube");

    // Using different interface pointers
    Drawable* drawPtr = &obj;
    Scalable* scalePtr = &obj;
    Rotatable* rotatePtr = &obj;

    drawPtr->draw();
    scalePtr->scale(2.0);
    rotatePtr->rotate(45.0);
    drawPtr->draw();
}
```

5.3.6 Virtual Inheritance with Abstract Classes

Virtual inheritance resolves the ”diamond problem” when multiple derived classes inherit from a common abstract base class.

```
#include <iostream>
#include <memory>

// Common abstract base
class Animal {
public:
    virtual void speak() const = 0;
    virtual ~Animal() = default;
};

// Virtual inheritance to avoid duplication
class Mammal : virtual public Animal {
public:
    virtual void nurse() const {
        std::cout << "Nursing young\n";
    }
};

class Winged : virtual public Animal {
public:
    virtual void fly() const {
        std::cout << "Flying\n";
    }
```

```
};

// Bat inherits from both, but only one Animal subobject
class Bat : public Mammal, public Winged {
public:
    void speak() const override {
        std::cout << "Bat screeches!\n";
    }

    void echolocate() const {
        std::cout << "Echolocating...\n";
    }
};

void virtualInheritanceAbstractClassExample() {
    Bat bat;

    // All pointers point to the same object
    Animal* animalPtr = &bat;
    Mammal* mammalPtr = &bat;
    Winged* wingedPtr = &bat;

    animalPtr->speak();     // Bat::speak()
    mammalPtr->nurse();     // Mammal::nurse()
    wingedPtr->fly();       // Winged::fly()

    // With virtual inheritance, there's only one Animal subobject
    std::cout << "Addresses:\n";
    std::cout << "  Bat:     " << &bat << '\n';
    std::cout << "  Animal:  " << animalPtr << '\n';
    std::cout << "  Mammal:  " << mammalPtr << '\n';
    std::cout << "  Winged:  " << wingedPtr << '\n';
}
```

5.3.7 CRTP (Curiously Recurring Template Pattern) with Abstract Classes

CRTP provides compile-time polymorphism as an alternative to virtual functions.

```
#include <iostream>
#include <memory>

// Template base class for compile-time polymorphism
```

```cpp
template<typename Derived>
class ShapeBase {
public:
    void draw() const {
        static_cast<const Derived*>(this)->drawImpl();
    }

    double area() const {
        return static_cast<const Derived*>(this)->areaImpl();
    }
};

class Circle : public ShapeBase<Circle> {
    double radius;

public:
    explicit Circle(double r) : radius(r) {}

    void drawImpl() const {
        std::cout << "Circle (r=" << radius << ")\n";
    }

    double areaImpl() const {
        return 3.14159 * radius * radius;
    }
};

class Rectangle : public ShapeBase<Rectangle> {
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    void drawImpl() const {
        std::cout << "Rectangle (" << width << "x" << height << ")\n";
    }

    double areaImpl() const {
        return width * height;
    }
};
```

```
void crtpExample() {
    Circle circle(5.0);
    Rectangle rect(4.0, 6.0);

    circle.draw();     // Compile-time resolution, no vtable overhead
    rect.draw();

    std::cout << "Circle area: " << circle.area() << '\n';
    std::cout << "Rectangle area: " << rect.area() << '\n';
}
```

5.3.8 Best Practices for Abstract Classes

1. **Always Declare Virtual Destructor**:
    - Abstract classes must have a virtual destructor to ensure proper cleanup of derived objects.
    - `virtual ~Base() = default;` is the modern way.
2. **Use `override` Keyword**:
    - Mark overriding functions with `override` to catch errors at compile time.
3. **Prefer Smart Pointers for Ownership**:
    - Use `std::unique_ptr` and `std::shared_ptr` instead of raw pointers.
4. **Use `final` When Appropriate**:
    - Mark classes as `final` if they should not be further derived.
    - Enables compiler optimizations (devirtualization).
5. **Consider Pure Abstract Classes (Interfaces)**:
    - An interface has only pure virtual functions and no data members.
    - Use `struct` for interfaces for clarity.
6. **Avoid Multiple Inheritance of Implementation**:
    - Multiple inheritance of implementation can lead to complexity.
    - Prefer multiple inheritance of interfaces.

5.3.9 Common Pitfalls

1. **Forgetting Virtual Destructor**:

```
class Base {  // No virtual destructor
    virtual void func() = 0;
    ~Base() {}  // Non-virtual
};
Base* ptr = new Derived();
delete ptr;  // Undefined behavior!
```

2. **Instantiating Abstract Class**:

```
Shape shape;  // Error: cannot instantiate abstract class
```

3. **Missing `override` Keyword**:

```
class Derived : public Base {
    void draw() { }  // Forgot override - creates new virtual function!
};
```

4. **Calling Pure Virtual Function in Constructor/Destructor**:

```
class Base {
public:
    Base() { pureVirtual(); }  // Undefined behavior!
    virtual void pureVirtual() = 0;
};
```

5.3.10 C++23 and C++26 Enhancements

C++23: `std::out_ptr` with Abstract Classes

```
// Safely interface with C APIs that create polymorphic objects
extern "C" {
    struct Widget;
    void create_widget(Widget** out);
    void destroy_widget(Widget*);
}

std::unique_ptr<Widget, decltype(&destroy_widget)>
    createWidget() {
    std::unique_ptr<Widget, decltype(&destroy_widget)> ptr(nullptr,
    ↪  destroy_widget);
```

```
    create_widget(std::out_ptr(ptr));  // C++23
    return ptr;
}
```

C++26 Preview: `std::observer_ptr` with Abstract Classes

```
// Proposed C++26
#include <observer_ptr>

void processShape(std::observer_ptr<Shape> shape) {
    if (shape) {
        shape->draw();  // Non-owning observation of abstract type
    }
}
```

5.3.11 Summary

- Abstract classes define interfaces through pure virtual functions.
- Pointers to abstract classes enable runtime polymorphism and interface-based programming.
- Smart pointers (`unique_ptr`, `shared_ptr`) are the modern way to manage polymorphic objects.
- Factory patterns using abstract classes provide flexible object creation.
- Multiple inheritance and virtual inheritance allow combining multiple interfaces.
- CRTP offers compile-time polymorphism as an alternative to virtual functions.
- Best practices: virtual destructor, `override`, smart pointers, `final`.
- C++23 adds `std::out_ptr` for C API interoperability; C++26 will introduce `std::observer_ptr`.

Exercise 16. ***Practice Exercises:***

1. *Create an abstract class `Vehicle` with pure virtual functions `start()`, `stop()`, and `getFuelLevel()`. Implement derived classes `Car`, `Motorcycle`, and `Truck`.*
2. *Implement a plugin system where plugins are loaded via abstract base class pointers. Use a factory to create plugins based on configuration.*

3. *Create a hierarchy of abstract classes for a graphics system: `Renderable`, `Transformable`, and `Animatable`. Implement a `Sprite` class that inherits from all three.*

4. *Design a logging system with abstract `Logger` base class. Implement `ConsoleLogger`, `FileLogger`, and `NetworkLogger`. Use a factory to create loggers.*

5. *Implement the CRTP pattern for a `Comparable` interface that provides `operator<` and `operator>` based on a derived class's `compare` method.*

6. *Using C++23's `std::out_ptr`, create a wrapper for a C library that creates polymorphic objects implementing an abstract interface.*

Chapter 6

Pointers and Data Structures

6.1 Pointers and Linked Lists

Linked lists are one of the most fundamental data structures in computer science. They consist of a sequence of nodes, where each node contains data and a pointer to the next node in the sequence. Linked lists are dynamic data structures, meaning their size can grow or shrink during program execution. This section explores how to implement singly and doubly linked lists using pointers in C++. We will cover adding, deleting, and traversing nodes, and provide practical examples to illustrate these concepts. Additionally, we will delve into advanced topics such as circular linked lists, smart pointers, and optimizing linked list operations.

Important

Key Insight: Linked lists demonstrate the power of pointers for creating dynamic, non-contiguous data structures. They trade off random access (O(1) in arrays) for efficient insertion and deletion (O(1) at known positions).

6.1.1 Implementing a Singly Linked List with Raw Pointers

A **singly linked list** is a linear data structure where each node contains data and a pointer to the next node. The last node points to `nullptr`, indicating the end of the list.

Important

Key Points:

- **Node Structure**: Contains data and a pointer to the next node.
- **Head Pointer**: Points to the first node; `nullptr` indicates empty list.
- **Operations**: Append, prepend, insert after, delete, find, traverse.
- **Memory Management**: Must explicitly delete nodes in destructor to avoid leaks.

Complete Singly Linked List Implementation

```
#include <iostream>
#include <stdexcept>

template<typename T>
class SinglyLinkedList {
private:
    struct Node {
        T data;
        Node* next;

        Node(const T& value) : data(value), next(nullptr) {}
        Node(T&& value) : data(std::move(value)), next(nullptr) {}
    };

    Node* head;
    size_t listSize;

public:
    SinglyLinkedList() : head(nullptr), listSize(0) {}

    // Copy constructor (deep copy)
    SinglyLinkedList(const SinglyLinkedList& other) : head(nullptr), listSize(0) {
        Node* current = other.head;
        while (current) {
            append(current->data);
            current = current->next;
        }
    }
```

```
// Move constructor
SinglyLinkedList(SinglyLinkedList&& other) noexcept
    : head(other.head), listSize(other.listSize) {
    other.head = nullptr;
    other.listSize = 0;
}

// Copy assignment
SinglyLinkedList& operator=(const SinglyLinkedList& other) {
    if (this != &other) {
        SinglyLinkedList temp(other);
        swap(temp);
    }
    return *this;
}

// Move assignment
SinglyLinkedList& operator=(SinglyLinkedList&& other) noexcept {
    if (this != &other) {
        clear();
        head = other.head;
        listSize = other.listSize;
        other.head = nullptr;
        other.listSize = 0;
    }
    return *this;
}

// Add node at the end
void append(const T& value) {
    Node* newNode = new Node(value);
    if (!head) {
        head = newNode;
    } else {
        Node* current = head;
        while (current->next) {
            current = current->next;
        }
        current->next = newNode;
    }
    ++listSize;
```

```
    }

    // Add node at the beginning
    void prepend(const T& value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
        ++listSize;
    }

    // Insert after a specific value
    bool insertAfter(const T& afterValue, const T& newValue) {
        Node* current = head;
        while (current && current->data != afterValue) {
            current = current->next;
        }

        if (!current) return false;  // Value not found

        Node* newNode = new Node(newValue);
        newNode->next = current->next;
        current->next = newNode;
        ++listSize;
        return true;
    }

    // Delete first occurrence of value
    bool deleteNode(const T& value) {
        if (!head) return false;

        // If head contains the value
        if (head->data == value) {
            Node* temp = head;
            head = head->next;
            delete temp;
            --listSize;
            return true;
        }

        // Search for the node
        Node* current = head;
        while (current->next && current->next->data != value) {
```

```
            current = current->next;
        }

        if (current->next) {
            Node* temp = current->next;
            current->next = current->next->next;
            delete temp;
            --listSize;
            return true;
        }

        return false;  // Value not found
    }

    // Find node containing value
    bool contains(const T& value) const {
        Node* current = head;
        while (current) {
            if (current->data == value) return true;
            current = current->next;
        }
        return false;
    }

    // Get size
    size_t size() const { return listSize; }

    // Check if empty
    bool empty() const { return head == nullptr; }

    // Traverse and print
    void print() const {
        Node* current = head;
        while (current) {
            std::cout << current->data << " -> ";
            current = current->next;
        }
        std::cout << "nullptr\n";
    }

    // Clear all nodes
    void clear() {
```

```
        Node* current = head;
        while (current) {
            Node* next = current->next;
            delete current;
            current = next;
        }
        head = nullptr;
        listSize = 0;
    }

    // Destructor
    ~SinglyLinkedList() {
        clear();
    }

private:
    void swap(SinglyLinkedList& other) noexcept {
        std::swap(head, other.head);
        std::swap(listSize, other.listSize);
    }
};

void singlyLinkedListExample() {
    SinglyLinkedList<int> list;

    list.append(10);
    list.append(20);
    list.append(30);
    std::cout << "After appending 10, 20, 30: ";
    list.print();

    list.prepend(5);
    std::cout << "After prepending 5: ";
    list.print();

    list.insertAfter(20, 25);
    std::cout << "After inserting 25 after 20: ";
    list.print();

    list.deleteNode(20);
    std::cout << "After deleting 20: ";
    list.print();
```

```
    std::cout << "Size: " << list.size() << '\n';
    std::cout << "Contains 25: " << (list.contains(25) ? "Yes" : "No") << '\n';
}
```

6.1.2 Doubly Linked Lists with Raw Pointers

A **doubly linked list** allows traversal in both directions, with each node containing pointers to both next and previous nodes.

Important

Advantages:

- Bidirectional traversal (forward and backward).
- O(1) deletion of a given node (with pointer to it).
- O(1) insertion before or after a given node.
- Trade-off: extra memory for the previous pointer.

Complete Doubly Linked List Implementation

```
#include <iostream>
#include <memory>

template<typename T>
class DoublyLinkedList {
private:
    struct Node {
        T data;
        Node* prev;
        Node* next;

        Node(const T& value) : data(value), prev(nullptr), next(nullptr) {}
        Node(T&& value) : data(std::move(value)), prev(nullptr), next(nullptr) {}
    };

    Node* head;
    Node* tail;
    size_t listSize;
```

```
public:
    DoublyLinkedList() : head(nullptr), tail(nullptr), listSize(0) {}

    // Add node at the end
    void append(const T& value) {
        Node* newNode = new Node(value);
        if (!head) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            newNode->prev = tail;
            tail = newNode;
        }
        ++listSize;
    }

    // Add node at the beginning
    void prepend(const T& value) {
        Node* newNode = new Node(value);
        if (!head) {
            head = tail = newNode;
        } else {
            newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
        ++listSize;
    }

    // Insert after a given node (by value)
    bool insertAfter(const T& afterValue, const T& newValue) {
        Node* current = head;
        while (current && current->data != afterValue) {
            current = current->next;
        }

        if (!current) return false;

        Node* newNode = new Node(newValue);
        newNode->prev = current;
        newNode->next = current->next;
```

```
    if (current->next) {
        current->next->prev = newNode;
    } else {
        tail = newNode;  // Inserting at the end
    }
    current->next = newNode;
    ++listSize;
    return true;
}

// Insert before a given node
bool insertBefore(const T& beforeValue, const T& newValue) {
    Node* current = head;
    while (current && current->data != beforeValue) {
        current = current->next;
    }

    if (!current) return false;

    Node* newNode = new Node(newValue);
    newNode->next = current;
    newNode->prev = current->prev;

    if (current->prev) {
        current->prev->next = newNode;
    } else {
        head = newNode;  // Inserting at the beginning
    }
    current->prev = newNode;
    ++listSize;
    return true;
}

// Delete first occurrence of value
bool deleteNode(const T& value) {
    Node* current = head;
    while (current && current->data != value) {
        current = current->next;
    }

    if (!current) return false;
```

```
        if (current->prev) {
            current->prev->next = current->next;
        } else {
            head = current->next;  // Deleting head
        }

        if (current->next) {
            current->next->prev = current->prev;
        } else {
            tail = current->prev;  // Deleting tail
        }

        delete current;
        --listSize;
        return true;
    }

    // Forward traversal
    void printForward() const {
        Node* current = head;
        while (current) {
            std::cout << current->data << " <-> ";
            current = current->next;
        }
        std::cout << "nullptr\n";
    }

    // Backward traversal
    void printBackward() const {
        Node* current = tail;
        while (current) {
            std::cout << current->data << " <-> ";
            current = current->prev;
        }
        std::cout << "nullptr\n";
    }

    size_t size() const { return listSize; }
    bool empty() const { return head == nullptr; }

    void clear() {
```

```
        Node* current = head;
        while (current) {
            Node* next = current->next;
            delete current;
            current = next;
        }
        head = tail = nullptr;
        listSize = 0;
    }

    ~DoublyLinkedList() {
        clear();
    }

    // Disable copying (or implement deep copy)
    DoublyLinkedList(const DoublyLinkedList&) = delete;
    DoublyLinkedList& operator=(const DoublyLinkedList&) = delete;

    // Move operations
    DoublyLinkedList(DoublyLinkedList&& other) noexcept
        : head(other.head), tail(other.tail), listSize(other.listSize) {
        other.head = other.tail = nullptr;
        other.listSize = 0;
    }

    DoublyLinkedList& operator=(DoublyLinkedList&& other) noexcept {
        if (this != &other) {
            clear();
            head = other.head;
            tail = other.tail;
            listSize = other.listSize;
            other.head = other.tail = nullptr;
            other.listSize = 0;
        }
        return *this;
    }
};

void doublyLinkedListExample() {
    DoublyLinkedList<int> list;

    list.append(10);
```

```
    list.append(20);
    list.append(30);
    std::cout << "Forward: ";
    list.printForward();
    std::cout << "Backward: ";
    list.printBackward();

    list.prepend(5);
    std::cout << "After prepending 5: ";
    list.printForward();

    list.insertAfter(20, 25);
    std::cout << "After inserting 25 after 20: ";
    list.printForward();

    list.insertBefore(20, 15);
    std::cout << "After inserting 15 before 20: ";
    list.printForward();

    list.deleteNode(20);
    std::cout << "After deleting 20: ";
    list.printForward();
}
```

6.1.3 Modern C++: Smart Pointers with Linked Lists

Using smart pointers eliminates manual memory management and reduces the risk of memory leaks. However, they require careful handling to avoid cycles.

Important

Smart Pointer Considerations:

- `std::unique_ptr` is ideal for singly linked lists (exclusive ownership).
- For doubly linked lists, `std::shared_ptr` combined with `std::weak_ptr` prevents cycles.
- Smart pointers simplify cleanup but add slight overhead.

Singly Linked List with `std::unique_ptr`

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

template<typename T>
class UniqueLinkedList {
private:
    struct Node {
        T data;
        std::unique_ptr<Node> next;

        explicit Node(const T& value) : data(value), next(nullptr) {}
        explicit Node(T&& value) : data(std::move(value)), next(nullptr) {}
    };

    std::unique_ptr<Node> head;
    size_t listSize{0};

public:
    void append(const T& value) {
        auto newNode = std::make_unique<Node>(value);

        if (!head) {
            head = std::move(newNode);
        } else {
            Node* current = head.get();
            while (current->next) {
                current = current->next.get();
            }
            current->next = std::move(newNode);
        }
        ++listSize;
    }

    void prepend(const T& value) {
        auto newNode = std::make_unique<Node>(value);
        newNode->next = std::move(head);
        head = std::move(newNode);
        ++listSize;
    }

    bool deleteNode(const T& value) {
        if (!head) return false;
```

```
        if (head->data == value) {
            head = std::move(head->next);
            --listSize;
            return true;
        }

        Node* current = head.get();
        while (current->next && current->next->data != value) {
            current = current->next.get();
        }

        if (current->next) {
            current->next = std::move(current->next->next);
            --listSize;
            return true;
        }

        return false;
    }

    void print() const {
        Node* current = head.get();
        while (current) {
            std::cout << current->data << " -> ";
            current = current->next.get();
        }
        std::cout << "nullptr\n";
    }

    size_t size() const { return listSize; }

    // No destructor needed - unique_ptr handles cleanup
};

void uniquePtrLinkedListExample() {
    UniqueLinkedList<int> list;
    list.append(10);
    list.append(20);
    list.append(30);
    list.prepend(5);
    list.print();
```

```
    list.deleteNode(20);
    list.print();
}
```

6.1.4 Circular Linked Lists

A **circular linked list** has the last node pointing back to the first, enabling continuous traversal.

```
#include <iostream>
#include <memory>

template<typename T>
class CircularLinkedList {
private:
    struct Node {
        T data;
        Node* next;

        explicit Node(const T& value) : data(value), next(nullptr) {}
    };

    Node* head;
    size_t listSize;

public:
    CircularLinkedList() : head(nullptr), listSize(0) {}

    void append(const T& value) {
        Node* newNode = new Node(value);

        if (!head) {
            head = newNode;
            head->next = head;  // Point to itself
        } else {
            Node* current = head;
            while (current->next != head) {
                current = current->next;
            }
            current->next = newNode;
            newNode->next = head;
        }
        ++listSize;
```

```
    }

    void prepend(const T& value) {
        Node* newNode = new Node(value);

        if (!head) {
            head = newNode;
            head->next = head;
        } else {
            // Find the last node
            Node* last = head;
            while (last->next != head) {
                last = last->next;
            }
            newNode->next = head;
            last->next = newNode;
            head = newNode;
        }
        ++listSize;
    }

    void print() const {
        if (!head) {
            std::cout << "empty\n";
            return;
        }

        Node* current = head;
        do {
            std::cout << current->data << " -> ";
            current = current->next;
        } while (current != head);
        std::cout << "(head)\n";
    }

    // Josephus problem simulation: eliminate every k-th node
    T josephus(int k) {
        if (!head || listSize == 0) throw std::runtime_error("Empty list");

        Node* current = head;
        Node* prev = nullptr;
```

```
        // Find the last node
        while (current->next != head) {
            current = current->next;
        }
        prev = current;
        current = head;

        while (listSize > 1) {
            // Move k-1 steps
            for (int i = 0; i < k - 1; ++i) {
                prev = current;
                current = current->next;
            }

            // Remove current node
            prev->next = current->next;
            T eliminated = current->data;
            delete current;
            --listSize;

            // Move to next node
            current = prev->next;
            std::cout << "Eliminated: " << eliminated << '\n';
        }

        T survivor = current->data;
        delete current;
        head = nullptr;
        listSize = 0;
        return survivor;
    }

    ~CircularLinkedList() {
        if (!head) return;

        Node* current = head;
        Node* next;
        do {
            next = current->next;
            delete current;
            current = next;
        } while (current != head);
```

```
    }
};

void circularLinkedListExample() {
    CircularLinkedList<int> list;
    for (int i = 1; i <= 7; ++i) {
        list.append(i);
    }
    list.print();

    std::cout << "\nJosephus problem (k=3):\n";
    int survivor = list.josephus(3);
    std::cout << "Survivor: " << survivor << '\n';
}
```

6.1.5 Performance Optimization Techniques

1. **Tail Pointer Optimization**:
    - Maintaining a tail pointer makes appending O(1) instead of O(n).
    - Already implemented in the doubly linked list example.
2. **Size Tracking**:
    - Track size to make `size()` O(1) instead of O(n).
    - Implemented in all examples above.
3. **Memory Pool Allocation**:
    - Pre-allocate a pool of nodes to reduce allocation overhead.
    - Useful in real-time or embedded systems.
4. **Sentinel Nodes**:
    - Use dummy head/tail nodes to simplify edge cases.
    - Eliminates special handling for empty lists and head/tail operations.

6.1.6 Common Pitfalls

1. **Memory Leaks**:

```
// Forgetting to delete nodes in destructor
~SinglyLinkedList() {
    // Missing deletion - memory leak!
}
```

2. **Dangling Pointers**:

```
Node* current = head;
delete current;
current->next;  // Dangling pointer access!
```

3. **Lost Nodes**:

```
head = head->next;  // Lost the original head node if not saved
```

4. **Circular References in Doubly Linked Lists**:

```
// With shared_ptr, this creates a cycle
struct Node {
    std::shared_ptr<Node> next;
    std::shared_ptr<Node> prev;  // Cycle!
};
```

   Solution: Use `std::weak_ptr` for prev.

5. **Iterator Invalidation**:

   - Pointers to nodes become invalid when nodes are deleted.
   - Always update pointers after modifications.

6.1.7 Summary

- Singly linked lists provide O(1) insertion/deletion at head, O(n) traversal.
- Doubly linked lists enable bidirectional traversal with O(1) insertion/deletion at both ends.
- Circular linked lists are useful for round-robin scheduling and Josephus problem.
- Modern C++: `std::unique_ptr` for singly linked lists, `shared_ptr` + `weak_ptr` for doubly linked lists.

- Performance optimizations: tail pointer, size tracking, memory pools, sentinel nodes.
- Common pitfalls: memory leaks, dangling pointers, lost nodes, circular references.
- Linked lists trade random access for efficient insertion/deletion compared to arrays.

Exercise 17. ***Practice Exercises:***

1. *Implement a singly linked list with a tail pointer for O(1) append. Add a reverse() method that reverses the list in place.*
2. *Create a doubly linked list with an iterator class that supports forward and backward traversal. Implement begin(), end(), rbegin(), and rend().*
3. *Implement a merge sort algorithm for a singly linked list. Compare its performance with array-based merge sort.*
4. *Create a circular buffer using a circular linked list with a fixed maximum size. Implement push and pop operations.*
5. *Using* `std::unique_ptr`*, implement a singly linked list with a custom allocator that uses a memory pool.*
6. *Solve the Josephus problem for different k values and list sizes. Analyze the time complexity of your implementation.*

6.2 Pointers and Trees

Trees are hierarchical data structures that consist of nodes connected by edges. Each node contains data and pointers to its child nodes. Trees are widely used in computer science for representing hierarchical relationships, such as file systems, organizational charts, and more. In this section, we will explore binary trees, implement a binary search tree (BST), and demonstrate how to traverse trees using pointers. We will cover in-order, pre-order, and post-order traversals with practical examples. Additionally, we will delve into advanced topics such as balanced trees, iterative traversal, and memory management with smart pointers.

Important

Key Insight: Trees enable efficient searching (O(log n) for balanced trees), hierarchical data representation, and recursive algorithms. Pointers are essential for linking nodes in tree structures.

6.2.1 Binary Trees with Raw Pointers

A **binary tree** is a tree data structure where each node has at most two children, referred to as the left child and the right child. Pointers are used to link nodes together, forming the tree structure.

Important

Key Concepts:

- **Root**: The topmost node of the tree.
- **Leaf**: A node with no children.
- **Height**: The number of edges on the longest path from root to leaf.
- **Depth**: The number of edges from root to a specific node.

Binary Tree Node Structure

```
#include <iostream>
#include <memory>
#include <queue>
#include <stack>
```

```cpp
#include <algorithm>

template<typename T>
struct TreeNode {
    T data;
    TreeNode* left;
    TreeNode* right;

    explicit TreeNode(const T& value) : data(value), left(nullptr), right(nullptr)
    ↪ {}
    explicit TreeNode(T&& value) : data(std::move(value)), left(nullptr),
    ↪ right(nullptr) {}

    // Check if leaf
    bool isLeaf() const {
        return left == nullptr && right == nullptr;
    }
};
```

6.2.2 Binary Search Tree Implementation

A **binary search tree (BST)** maintains the property that for any node, all values in the left subtree are less than the node's value, and all values in the right subtree are greater. This property enables efficient search, insertion, and deletion operations.

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

template<typename T>
class BinarySearchTree {
private:
    struct Node {
        T data;
        Node* left;
        Node* right;

        explicit Node(const T& value) : data(value), left(nullptr), right(nullptr)
        ↪ {}
        explicit Node(T&& value) : data(std::move(value)), left(nullptr),
        ↪ right(nullptr) {}
```

```
    };

    Node* root;
    size_t treeSize;

    // Recursive insertion
    Node* insert(Node* node, const T& value) {
        if (!node) {
            ++treeSize;
            return new Node(value);
        }

        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }
        // Duplicate: do nothing
        return node;
    }

    // Find minimum value node
    Node* findMin(Node* node) const {
        while (node && node->left) {
            node = node->left;
        }
        return node;
    }

    // Find maximum value node
    Node* findMax(Node* node) const {
        while (node && node->right) {
            node = node->right;
        }
        return node;
    }

    // Recursive deletion
    Node* deleteNode(Node* node, const T& value) {
        if (!node) return nullptr;

        if (value < node->data) {
```

```
            node->left = deleteNode(node->left, value);
        } else if (value > node->data) {
            node->right = deleteNode(node->right, value);
        } else {
            // Node found
            if (!node->left) {
                Node* temp = node->right;
                delete node;
                --treeSize;
                return temp;
            } else if (!node->right) {
                Node* temp = node->left;
                delete node;
                --treeSize;
                return temp;
            }

            // Node with two children: find inorder successor
            Node* successor = findMin(node->right);
            node->data = successor->data;
            node->right = deleteNode(node->right, successor->data);
        }
        return node;
    }

    // Recursive search
    bool search(Node* node, const T& value) const {
        if (!node) return false;
        if (value == node->data) return true;
        if (value < node->data) return search(node->left, value);
        return search(node->right, value);
    }

    // Recursive tree deletion
    void clear(Node* node) {
        if (!node) return;
        clear(node->left);
        clear(node->right);
        delete node;
    }

public:
```

```
    BinarySearchTree() : root(nullptr), treeSize(0) {}

    // Copy constructor (deep copy)
    BinarySearchTree(const BinarySearchTree& other) : root(nullptr), treeSize(0) {
        copy(other.root);
    }

    // Move constructor
    BinarySearchTree(BinarySearchTree&& other) noexcept
        : root(other.root), treeSize(other.treeSize) {
        other.root = nullptr;
        other.treeSize = 0;
    }

    // Copy assignment
    BinarySearchTree& operator=(const BinarySearchTree& other) {
        if (this != &other) {
            clear();
            copy(other.root);
        }
        return *this;
    }

    // Move assignment
    BinarySearchTree& operator=(BinarySearchTree&& other) noexcept {
        if (this != &other) {
            clear();
            root = other.root;
            treeSize = other.treeSize;
            other.root = nullptr;
            other.treeSize = 0;
        }
        return *this;
    }

    // Public operations
    void insert(const T& value) {
        root = insert(root, value);
    }

    bool remove(const T& value) {
        size_t oldSize = treeSize;
```

```
        root = deleteNode(root, value);
        return treeSize < oldSize;
    }

    bool contains(const T& value) const {
        return search(root, value);
    }

    size_t size() const { return treeSize; }
    bool empty() const { return root == nullptr; }

    // Get minimum value
    T min() const {
        if (!root) throw std::runtime_error("Tree is empty");
        Node* minNode = findMin(root);
        return minNode->data;
    }

    // Get maximum value
    T max() const {
        if (!root) throw std::runtime_error("Tree is empty");
        Node* maxNode = findMax(root);
        return maxNode->data;
    }

    // Destructor
    ~BinarySearchTree() {
        clear();
    }

private:
    void clear() {
        clear(root);
        root = nullptr;
        treeSize = 0;
    }

    void copy(Node* node) {
        if (!node) return;
        insert(node->data);
        copy(node->left);
        copy(node->right);
```

```
    }
};

void binarySearchTreeExample() {
    BinarySearchTree<int> bst;

    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
    bst.insert(60);
    bst.insert(80);

    std::cout << "Size: " << bst.size() << '\n';
    std::cout << "Min: " << bst.min() << '\n';
    std::cout << "Max: " << bst.max() << '\n';
    std::cout << "Contains 40: " << (bst.contains(40) ? "Yes" : "No") << '\n';
    std::cout << "Contains 90: " << (bst.contains(90) ? "Yes" : "No") << '\n';

    bst.remove(40);
    std::cout << "After removing 40, contains 40: " << (bst.contains(40) ? "Yes" :
    ↪  "No") << '\n';
    std::cout << "Size: " << bst.size() << '\n';
}
```

6.2.3 Tree Traversals: Recursive and Iterative

Tree traversal is the process of visiting all nodes in a tree in a specific order. The three most common traversal methods are in-order, pre-order, and post-order.

Important

Traversal Types:

- **In-Order (LNR)**: Left, Node, Right. For BSTs, this produces sorted order.
- **Pre-Order (NLR)**: Node, Left, Right. Useful for tree copying.
- **Post-Order (LRN)**: Left, Right, Node. Useful for tree deletion.
- **Level-Order (BFS)**: Level by level from root.

Recursive Traversals

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

template<typename T>
class TreeTraversals {
public:
    // Recursive in-order (LNR)
    static void inOrder(TreeNode<T>* node, std::vector<T>& result) {
        if (!node) return;
        inOrder(node->left, result);
        result.push_back(node->data);
        inOrder(node->right, result);
    }

    // Recursive pre-order (NLR)
    static void preOrder(TreeNode<T>* node, std::vector<T>& result) {
        if (!node) return;
        result.push_back(node->data);
        preOrder(node->left, result);
        preOrder(node->right, result);
    }

    // Recursive post-order (LRN)
    static void postOrder(TreeNode<T>* node, std::vector<T>& result) {
        if (!node) return;
        postOrder(node->left, result);
        postOrder(node->right, result);
        result.push_back(node->data);
```

```
    }
};

void recursiveTraversalExample() {
    TreeNode<int>* root = new TreeNode<int>(50);
    root->left = new TreeNode<int>(30);
    root->right = new TreeNode<int>(70);
    root->left->left = new TreeNode<int>(20);
    root->left->right = new TreeNode<int>(40);
    root->right->left = new TreeNode<int>(60);
    root->right->right = new TreeNode<int>(80);

    std::vector<int> result;

    TreeTraversals<int>::inOrder(root, result);
    std::cout << "In-order: ";
    for (int v : result) std::cout << v << ' ';
    std::cout << '\n';

    result.clear();
    TreeTraversals<int>::preOrder(root, result);
    std::cout << "Pre-order: ";
    for (int v : result) std::cout << v << ' ';
    std::cout << '\n';

    result.clear();
    TreeTraversals<int>::postOrder(root, result);
    std::cout << "Post-order: ";
    for (int v : result) std::cout << v << ' ';
    std::cout << '\n';

    // Cleanup (simplified)
    delete root->left->left; delete root->left->right; delete root->left;
    delete root->right->left; delete root->right->right; delete root->right;
    delete root;
}
```

Iterative Traversals

Iterative traversals use explicit stacks instead of recursion, avoiding stack overflow for deep trees.

```
#include <stack>
#include <queue>
```

```
template<typename T>
class IterativeTraversals {
public:
    // Iterative in-order using stack
    static void inOrder(TreeNode<T>* root, std::vector<T>& result) {
        std::stack<TreeNode<T>*> stack;
        TreeNode<T>* current = root;

        while (current || !stack.empty()) {
            while (current) {
                stack.push(current);
                current = current->left;
            }
            current = stack.top();
            stack.pop();
            result.push_back(current->data);
            current = current->right;
        }
    }

    // Iterative pre-order using stack
    static void preOrder(TreeNode<T>* root, std::vector<T>& result) {
        if (!root) return;

        std::stack<TreeNode<T>*> stack;
        stack.push(root);

        while (!stack.empty()) {
            TreeNode<T>* current = stack.top();
            stack.pop();
            result.push_back(current->data);

            // Push right first so left is processed first (stack LIFO)
            if (current->right) stack.push(current->right);
            if (current->left) stack.push(current->left);
        }
    }

    // Iterative post-order using two stacks
    static void postOrder(TreeNode<T>* root, std::vector<T>& result) {
        if (!root) return;
```

```
    std::stack<TreeNode<T>*> s1, s2;
    s1.push(root);

    while (!s1.empty()) {
        TreeNode<T>* current = s1.top();
        s1.pop();
        s2.push(current);

        if (current->left) s1.push(current->left);
        if (current->right) s1.push(current->right);
    }

    while (!s2.empty()) {
        result.push_back(s2.top()->data);
        s2.pop();
    }
}

// Level-order (BFS) using queue
static void levelOrder(TreeNode<T>* root, std::vector<std::vector<T>>& result)
↪  {
    if (!root) return;

    std::queue<TreeNode<T>*> queue;
    queue.push(root);

    while (!queue.empty()) {
        size_t levelSize = queue.size();
        std::vector<T> level;

        for (size_t i = 0; i < levelSize; ++i) {
            TreeNode<T>* current = queue.front();
            queue.pop();
            level.push_back(current->data);

            if (current->left) queue.push(current->left);
            if (current->right) queue.push(current->right);
        }
        result.push_back(std::move(level));
    }
}
```

```
};

void iterativeTraversalExample() {
    TreeNode<int>* root = new TreeNode<int>(50);
    root->left = new TreeNode<int>(30);
    root->right = new TreeNode<int>(70);
    root->left->left = new TreeNode<int>(20);
    root->left->right = new TreeNode<int>(40);
    root->right->left = new TreeNode<int>(60);
    root->right->right = new TreeNode<int>(80);

    std::vector<int> result;

    IterativeTraversals<int>::inOrder(root, result);
    std::cout << "Iterative in-order: ";
    for (int v : result) std::cout << v << ' ';
    std::cout << '\n';

    result.clear();
    IterativeTraversals<int>::preOrder(root, result);
    std::cout << "Iterative pre-order: ";
    for (int v : result) std::cout << v << ' ';
    std::cout << '\n';

    result.clear();
    IterativeTraversals<int>::postOrder(root, result);
    std::cout << "Iterative post-order: ";
    for (int v : result) std::cout << v << ' ';
    std::cout << '\n';

    std::vector<std::vector<int>> levels;
    IterativeTraversals<int>::levelOrder(root, levels);
    std::cout << "Level-order:\n";
    for (size_t i = 0; i < levels.size(); ++i) {
        std::cout << "  Level " << i << ": ";
        for (int v : levels[i]) std::cout << v << ' ';
        std::cout << '\n';
    }

    // Cleanup (simplified)
    delete root->left->left; delete root->left->right; delete root->left;
    delete root->right->left; delete root->right->right; delete root->right;
```

```
    delete root;
}
```

6.2.4 Modern C++: Smart Pointers for Trees

Using smart pointers eliminates manual memory management and prevents leaks. For trees, `std::unique_ptr` is ideal as each node exclusively owns its children.

```
#include <memory>
#include <iostream>
#include <vector>

template<typename T>
class SmartBST {
private:
    struct Node {
        T data;
        std::unique_ptr<Node> left;
        std::unique_ptr<Node> right;

        explicit Node(const T& value) : data(value), left(nullptr), right(nullptr)
        ↪ {}
        explicit Node(T&& value) : data(std::move(value)), left(nullptr),
        ↪ right(nullptr) {}
    };

    std::unique_ptr<Node> root;
    size_t treeSize{0};

    std::unique_ptr<Node> insert(std::unique_ptr<Node> node, const T& value) {
        if (!node) {
            ++treeSize;
            return std::make_unique<Node>(value);
        }

        if (value < node->data) {
            node->left = insert(std::move(node->left), value);
        } else if (value > node->data) {
            node->right = insert(std::move(node->right), value);
        }
        return node;
    }
```

```
    bool contains(const Node* node, const T& value) const {
        if (!node) return false;
        if (value == node->data) return true;
        if (value < node->data) return contains(node->left.get(), value);
        return contains(node->right.get(), value);
    }

    void inOrder(const Node* node, std::vector<T>& result) const {
        if (!node) return;
        inOrder(node->left.get(), result);
        result.push_back(node->data);
        inOrder(node->right.get(), result);
    }

public:
    void insert(const T& value) {
        root = insert(std::move(root), value);
    }

    bool contains(const T& value) const {
        return contains(root.get(), value);
    }

    size_t size() const { return treeSize; }

    std::vector<T> toSortedVector() const {
        std::vector<T> result;
        inOrder(root.get(), result);
        return result;
    }

    // No destructor needed - unique_ptr handles cleanup
};

void smartTreeExample() {
    SmartBST<int> bst;
    bst.insert(50);
    bst.insert(30);
    bst.insert(70);
    bst.insert(20);
    bst.insert(40);
```

```
    bst.insert(60);
    bst.insert(80);

    std::cout << "Size: " << bst.size() << '\n';
    std::cout << "Contains 40: " << (bst.contains(40) ? "Yes" : "No") << '\n';

    auto sorted = bst.toSortedVector();
    std::cout << "Sorted: ";
    for (int v : sorted) std::cout << v << ' ';
    std::cout << '\n';
}
```

6.2.5 Advanced Topic: AVL Tree (Self-Balancing)

AVL trees maintain balance by ensuring the height difference between left and right subtrees is at most 1.

```
#include <memory>
#include <algorithm>
#include <iostream>

template<typename T>
class AVLTree {
private:
    struct Node {
        T data;
        std::unique_ptr<Node> left;
        std::unique_ptr<Node> right;
        int height;

        explicit Node(const T& value)
            : data(value), left(nullptr), right(nullptr), height(1) {}
    };

    std::unique_ptr<Node> root;
    size_t treeSize{0};

    int getHeight(const Node* node) const {
        return node ? node->height : 0;
    }

    int getBalance(const Node* node) const {
```

```
    return node ? getHeight(node->left.get()) - getHeight(node->right.get()) :
    ↪  0;
}

void updateHeight(Node* node) {
    if (node) {
        node->height = 1 + std::max(getHeight(node->left.get()),
                                    getHeight(node->right.get()));
    }
}

std::unique_ptr<Node> rotateRight(std::unique_ptr<Node> y) {
    std::unique_ptr<Node> x = std::move(y->left);
    std::unique_ptr<Node> T2 = std::move(x->right);

    x->right = std::move(y);
    x->right->left = std::move(T2);

    updateHeight(x->right.get());
    updateHeight(x.get());

    return x;
}

std::unique_ptr<Node> rotateLeft(std::unique_ptr<Node> x) {
    std::unique_ptr<Node> y = std::move(x->right);
    std::unique_ptr<Node> T2 = std::move(y->left);

    y->left = std::move(x);
    y->left->right = std::move(T2);

    updateHeight(y->left.get());
    updateHeight(y.get());

    return y;
}

std::unique_ptr<Node> insert(std::unique_ptr<Node> node, const T& value) {
    if (!node) {
        ++treeSize;
        return std::make_unique<Node>(value);
    }
```

```
        if (value < node->data) {
            node->left = insert(std::move(node->left), value);
        } else if (value > node->data) {
            node->right = insert(std::move(node->right), value);
        } else {
            return node;  // Duplicate
        }

        updateHeight(node.get());

        int balance = getBalance(node.get());

        // Left Left Case
        if (balance > 1 && value < node->left->data) {
            return rotateRight(std::move(node));
        }

        // Right Right Case
        if (balance < -1 && value > node->right->data) {
            return rotateLeft(std::move(node));
        }

        // Left Right Case
        if (balance > 1 && value > node->left->data) {
            node->left = rotateLeft(std::move(node->left));
            return rotateRight(std::move(node));
        }

        // Right Left Case
        if (balance < -1 && value < node->right->data) {
            node->right = rotateRight(std::move(node->right));
            return rotateLeft(std::move(node));
        }

        return node;
    }

    void inOrder(const Node* node, std::vector<T>& result) const {
        if (!node) return;
        inOrder(node->left.get(), result);
        result.push_back(node->data);
```

```
        inOrder(node->right.get(), result);
    }

public:
    void insert(const T& value) {
        root = insert(std::move(root), value);
    }

    size_t size() const { return treeSize; }

    std::vector<T> toSortedVector() const {
        std::vector<T> result;
        inOrder(root.get(), result);
        return result;
    }

    bool isBalanced() const {
        return std::abs(getBalance(root.get())) <= 1;
    }
};

void avlTreeExample() {
    AVLTree<int> avl;

    // Insert values that would create an unbalanced BST
    for (int i = 1; i <= 10; ++i) {
        avl.insert(i);
    }

    std::cout << "AVL tree size: " << avl.size() << '\n';
    std::cout << "Is balanced: " << (avl.isBalanced() ? "Yes" : "No") << '\n';

    auto sorted = avl.toSortedVector();
    std::cout << "Sorted: ";
    for (int v : sorted) std::cout << v << ' ';
    std::cout << '\n';
}
```

6.2.6 Common Pitfalls

1. **Stack Overflow from Deep Recursion**:

```
// Very deep tree may cause stack overflow with recursion
void recursiveTraversal(TreeNode* node) {
    if (!node) return;
    recursiveTraversal(node->left);  // May overflow for deep trees
}
```

Solution: Use iterative traversal for deep trees.

2. **Memory Leaks from Improper Deletion**:

```
// Missing child deletion
~BinarySearchTree() {
    delete root;  // Only deletes root, children lost!
}
```

Solution: Recursively delete children or use smart pointers.

3. **Lost Nodes After Deletion**:

```
// Deleting node without reattaching children
delete node;  // Children become orphaned!
```

4. **Not Handling Duplicate Values**:

   - Define policy: ignore, count, or store in subtree.

5. **Incorrect BST Property Violation**:

```
// Inserting without maintaining BST property
if (value < node->data) node->right = insert(...);  // Wrong direction!
```

6.2.7 Summary

- Binary trees use pointers to link nodes hierarchically.
- Binary search trees (BSTs) maintain ordering for O(log n) search, insert, delete (average case).
- Tree traversals: in-order (sorted), pre-order (copy), post-order (delete), level-order (BFS).
- Iterative traversals avoid recursion depth issues using explicit stacks/queues.
- Smart pointers (`unique_ptr`) eliminate manual memory management for trees.
- Balanced trees (AVL, Red-Black) maintain logarithmic performance guarantees.

- Common pitfalls: recursion depth, memory leaks, lost nodes, BST property violations.

Exercise 18. ***Practice Exercises:***

1. *Implement a function to compute the height of a binary tree (both recursive and iterative).*
2. *Create a function that checks if a binary tree is a valid BST (using in-order traversal or min/max bounds).*
3. *Implement a level-order traversal that returns a vector of vectors, each containing nodes at the same level.*
4. *Write a function to find the lowest common ancestor (LCA) of two nodes in a binary tree.*
5. *Implement a binary tree using* `std::unique_ptr` *with a method to serialize/deserialize to a string.*
6. *Implement a Red-Black tree (simplified) or research and implement tree rotations for self-balancing.*

6.3 Pointers and Graphs

Graphs are a fundamental data structure used to represent relationships between objects. A graph consists of a set of vertices (or nodes) and a set of edges that connect these vertices. Graphs can be directed or undirected, and they are widely used in applications such as social networks, routing algorithms, and recommendation systems. In this section, we will explore the adjacency list representation of graphs and demonstrate how to implement a graph using pointers in C++. We will also cover graph traversal algorithms, advanced topics such as weighted graphs and directed graphs, and memory management with smart pointers.

Important

Key Insight: Graphs model complex relationships. The adjacency list representation is memory-efficient for sparse graphs and enables efficient traversal algorithms.

6.3.1 Adjacency List Representation with Pointers

The **adjacency list** representation stores each vertex with a list of its adjacent vertices. For weighted graphs, the list can also store edge weights. This representation is efficient for sparse graphs.

Important

Key Concepts:

- **Vertex**: A node in the graph with a unique identifier.
- **Edge**: A connection between two vertices (directed or undirected).
- **Adjacency List**: For each vertex, a list of neighbors (and optionally weights).
- **Sparse vs Dense**: Adjacency list is optimal for sparse graphs (—E— ¡¡ —V—²).

Basic Graph Implementation with Raw Pointers

```
#include <iostream>
#include <vector>
#include <memory>
#include <queue>
#include <stack>
```

```cpp
#include <algorithm>
#include <limits>
#include <unordered_map>

template<typename T>
class Graph {
private:
    struct Vertex {
        T data;
        std::vector<Vertex*> neighbors;

        explicit Vertex(const T& value) : data(value) {}
        explicit Vertex(T&& value) : data(std::move(value)) {}
    };

    std::vector<Vertex*> vertices;
    bool isDirected;

    Vertex* findVertex(const T& value) {
        for (auto v : vertices) {
            if (v->data == value) return v;
        }
        return nullptr;
    }

    const Vertex* findVertex(const T& value) const {
        for (auto v : vertices) {
            if (v->data == value) return v;
        }
        return nullptr;
    }

public:
    explicit Graph(bool directed = false) : isDirected(directed) {}

    // Add vertex
    bool addVertex(const T& value) {
        if (findVertex(value)) return false;
        vertices.push_back(new Vertex(value));
        return true;
    }
```

```
    // Add edge (undirected by default, directed if graph is directed)
    bool addEdge(const T& from, const T& to) {
        Vertex* fromVertex = findVertex(from);
        Vertex* toVertex = findVertex(to);

        if (!fromVertex || !toVertex) return false;

        fromVertex->neighbors.push_back(toVertex);
        if (!isDirected) {
            toVertex->neighbors.push_back(fromVertex);
        }
        return true;
    }

    // Remove edge
    bool removeEdge(const T& from, const T& to) {
        Vertex* fromVertex = findVertex(from);
        Vertex* toVertex = findVertex(to);

        if (!fromVertex || !toVertex) return false;

        auto removeFrom = [](Vertex* v, Vertex* target) {
            auto it = std::find(v->neighbors.begin(), v->neighbors.end(), target);
            if (it != v->neighbors.end()) {
                v->neighbors.erase(it);
                return true;
            }
            return false;
        };

        bool removed = removeFrom(fromVertex, toVertex);
        if (!isDirected) {
            removed = removeFrom(toVertex, fromVertex) || removed;
        }
        return removed;
    }

    // Remove vertex and all incident edges
    bool removeVertex(const T& value) {
        Vertex* target = findVertex(value);
        if (!target) return false;
```

```
        // Remove all edges to/from this vertex
        for (auto v : vertices) {
            auto it = std::find(v->neighbors.begin(), v->neighbors.end(), target);
            if (it != v->neighbors.end()) {
                v->neighbors.erase(it);
            }
        }

        // Remove vertex from list
        auto it = std::find(vertices.begin(), vertices.end(), target);
        if (it != vertices.end()) {
            delete target;
            vertices.erase(it);
            return true;
        }
        return false;
    }

    // Check if edge exists
    bool hasEdge(const T& from, const T& to) const {
        const Vertex* fromVertex = findVertex(from);
        const Vertex* toVertex = findVertex(to);

        if (!fromVertex || !toVertex) return false;

        return std::find(fromVertex->neighbors.begin(),
                         fromVertex->neighbors.end(),
                         toVertex) != fromVertex->neighbors.end();
    }

    // Get neighbors of a vertex
    std::vector<T> getNeighbors(const T& value) const {
        const Vertex* vertex = findVertex(value);
        std::vector<T> result;

        if (vertex) {
            for (auto neighbor : vertex->neighbors) {
                result.push_back(neighbor->data);
            }
        }
        return result;
    }
```

```
// Print graph
void print() const {
    for (auto vertex : vertices) {
        std::cout << vertex->data << " -> ";
        for (auto neighbor : vertex->neighbors) {
            std::cout << neighbor->data << " ";
        }
        std::cout << '\n';
    }
}

// Get number of vertices
size_t vertexCount() const { return vertices.size(); }

// Destructor
~Graph() {
    for (auto vertex : vertices) {
        delete vertex;
    }
}

// Disable copying
Graph(const Graph&) = delete;
Graph& operator=(const Graph&) = delete;

// Enable moving
Graph(Graph&& other) noexcept
    : vertices(std::move(other.vertices)), isDirected(other.isDirected) {
    other.vertices.clear();
}

Graph& operator=(Graph&& other) noexcept {
    if (this != &other) {
        clear();
        vertices = std::move(other.vertices);
        isDirected = other.isDirected;
        other.vertices.clear();
    }
    return *this;
}
```

```
private:
    void clear() {
        for (auto vertex : vertices) delete vertex;
        vertices.clear();
    }
};

void adjacencyListExample() {
    Graph<int> graph(false);  // Undirected graph

    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);
    graph.addVertex(4);

    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);

    std::cout << "Graph adjacency list:\n";
    graph.print();

    std::cout << "Neighbors of 1: ";
    for (int n : graph.getNeighbors(1)) std::cout << n << ' ';
    std::cout << '\n';

    std::cout << "Has edge 1-3: " << (graph.hasEdge(1, 3) ? "Yes" : "No") << '\n';
    std::cout << "Has edge 1-4: " << (graph.hasEdge(1, 4) ? "Yes" : "No") << '\n';
}
```

6.3.2 Graph Traversal Algorithms

Graph traversal algorithms visit all vertices in a graph. The two most common are Depth-First Search (DFS) and Breadth-First Search (BFS).

Important

Traversal Types:

- **DFS (Depth-First Search)**: Explores as far as possible along each branch before backtracking. Uses stack (recursion or explicit).
- **BFS (Breadth-First Search)**: Explores all neighbors at current depth before moving deeper. Uses queue.
- Both can be used for connectivity, cycle detection, and path finding.

Depth-First Search (DFS) Implementation

```
#include <stack>
#include <unordered_set>

template<typename T>
class GraphTraversal {
public:
    // Recursive DFS
    static void dfsRecursive(typename Graph<T>::Vertex* vertex,
                             std::unordered_set<typename Graph<T>::Vertex*>&
                             ↪  visited,
                             std::vector<T>& result) {
        if (!vertex || visited.count(vertex)) return;

        visited.insert(vertex);
        result.push_back(vertex->data);

        for (auto neighbor : vertex->neighbors) {
            dfsRecursive(neighbor, visited, result);
        }
    }

    // Iterative DFS using stack
    static std::vector<T> dfsIterative(Graph<T>& graph, const T& start) {
        std::vector<T> result;
        std::unordered_set<typename Graph<T>::Vertex*> visited;
        std::stack<typename Graph<T>::Vertex*> stack;

        auto* startVertex = graph.findVertex(start);
```

```
        if (!startVertex) return result;

        stack.push(startVertex);

        while (!stack.empty()) {
            auto* current = stack.top();
            stack.pop();

            if (visited.count(current)) continue;

            visited.insert(current);
            result.push_back(current->data);

            // Push neighbors in reverse order for natural DFS order
            for (auto it = current->neighbors.rbegin();
                 it != current->neighbors.rend(); ++it) {
                if (!visited.count(*it)) {
                    stack.push(*it);
                }
            }
        }

        return result;
    }

    // Breadth-First Search (BFS) using queue
    static std::vector<T> bfs(Graph<T>& graph, const T& start) {
        std::vector<T> result;
        std::unordered_set<typename Graph<T>::Vertex*> visited;
        std::queue<typename Graph<T>::Vertex*> queue;

        auto* startVertex = graph.findVertex(start);
        if (!startVertex) return result;

        queue.push(startVertex);
        visited.insert(startVertex);

        while (!queue.empty()) {
            auto* current = queue.front();
            queue.pop();
            result.push_back(current->data);
```

```
            for (auto neighbor : current->neighbors) {
                if (!visited.count(neighbor)) {
                    visited.insert(neighbor);
                    queue.push(neighbor);
                }
            }
        }

        return result;
    }

    // Find connected components
    static std::vector<std::vector<T>> connectedComponents(Graph<T>& graph) {
        std::vector<std::vector<T>> components;
        std::unordered_set<typename Graph<T>::Vertex*> visited;

        for (auto* vertex : graph.getVertices()) {
            if (!visited.count(vertex)) {
                std::vector<T> component;
                std::queue<typename Graph<T>::Vertex*> queue;

                queue.push(vertex);
                visited.insert(vertex);

                while (!queue.empty()) {
                    auto* current = queue.front();
                    queue.pop();
                    component.push_back(current->data);

                    for (auto neighbor : current->neighbors) {
                        if (!visited.count(neighbor)) {
                            visited.insert(neighbor);
                            queue.push(neighbor);
                        }
                    }
                }
                components.push_back(component);
            }
        }

        return components;
    }
```

```
};

void traversalExample() {
    Graph<int> graph(false);

    for (int i = 1; i <= 7; ++i) graph.addVertex(i);

    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(2, 5);
    graph.addEdge(3, 6);
    graph.addEdge(3, 7);

    std::cout << "DFS from 1 (iterative): ";
    auto dfsResult = GraphTraversal<int>::dfsIterative(graph, 1);
    for (int v : dfsResult) std::cout << v << ' ';
    std::cout << '\n';

    std::cout << "BFS from 1: ";
    auto bfsResult = GraphTraversal<int>::bfs(graph, 1);
    for (int v : bfsResult) std::cout << v << ' ';
    std::cout << '\n';
}
```

6.3.3 Weighted Graphs

Weighted graphs associate a weight (cost, distance, capacity) with each edge, enabling algorithms like Dijkstra's shortest path.

```
#include <queue>
#include <vector>
#include <limits>

template<typename T>
class WeightedGraph {
private:
    struct Edge {
        int to;
        int weight;

        Edge(int t, int w) : to(t), weight(w) {}
```

```
    };

    struct Vertex {
        T data;
        std::vector<Edge> neighbors;

        explicit Vertex(const T& value) : data(value) {}
    };

    std::vector<Vertex*> vertices;
    std::unordered_map<T, int> indexMap;  // Value to index mapping

public:
    bool addVertex(const T& value) {
        if (indexMap.count(value)) return false;
        indexMap[value] = vertices.size();
        vertices.push_back(new Vertex(value));
        return true;
    }

    bool addEdge(const T& from, const T& to, int weight) {
        if (!indexMap.count(from) || !indexMap.count(to)) return false;

        int fromIdx = indexMap[from];
        int toIdx = indexMap[to];

        vertices[fromIdx]->neighbors.emplace_back(toIdx, weight);
        return true;
    }

    // Dijkstra's algorithm for shortest path
    std::pair<std::vector<int>, std::vector<T>> dijkstra(const T& start, const T&
    ↪  end) {
        if (!indexMap.count(start) || !indexMap.count(end)) {
            return {{}, {}};
        }

        int startIdx = indexMap[start];
        int endIdx = indexMap[end];
        size_t n = vertices.size();

        std::vector<int> dist(n, std::numeric_limits<int>::max());
```

```
        std::vector<int> prev(n, -1);
        using P = std::pair<int, int>;  // (distance, vertex)
        std::priority_queue<P, std::vector<P>, std::greater<P>> pq;

        dist[startIdx] = 0;
        pq.push({0, startIdx});

        while (!pq.empty()) {
            auto [currentDist, u] = pq.top();
            pq.pop();

            if (currentDist > dist[u]) continue;
            if (u == endIdx) break;

            for (const auto& edge : vertices[u]->neighbors) {
                int v = edge.to;
                int newDist = dist[u] + edge.weight;

                if (newDist < dist[v]) {
                    dist[v] = newDist;
                    prev[v] = u;
                    pq.push({newDist, v});
                }
            }
        }

        // Reconstruct path
        std::vector<T> path;
        if (dist[endIdx] == std::numeric_limits<int>::max()) {
            return {dist, path};  // No path found
        }

        for (int v = endIdx; v != -1; v = prev[v]) {
            path.push_back(vertices[v]->data);
        }
        std::reverse(path.begin(), path.end());

        return {dist, path};
    }

    void print() const {
        for (size_t i = 0; i < vertices.size(); ++i) {
```

```
            std::cout << vertices[i]->data << " -> ";
            for (const auto& edge : vertices[i]->neighbors) {
                std::cout << "(" << vertices[edge.to]->data << "," << edge.weight
                ↪  << ") ";
            }
            std::cout << '\n';
        }
    }

    ~WeightedGraph() {
        for (auto vertex : vertices) delete vertex;
    }
};

void weightedGraphExample() {
    WeightedGraph<char> graph;

    graph.addVertex('A');
    graph.addVertex('B');
    graph.addVertex('C');
    graph.addVertex('D');
    graph.addVertex('E');

    graph.addEdge('A', 'B', 4);
    graph.addEdge('A', 'C', 2);
    graph.addEdge('B', 'C', 1);
    graph.addEdge('B', 'D', 5);
    graph.addEdge('C', 'D', 8);
    graph.addEdge('C', 'E', 10);
    graph.addEdge('D', 'E', 2);

    std::cout << "Weighted Graph:\n";
    graph.print();

    auto [distances, path] = graph.dijkstra('A', 'E');
    std::cout << "Shortest path from A to E: ";
    for (char c : path) std::cout << c << ' ';
    std::cout << "\nDistance: " << distances[graph.getIndex('E')] << '\n';
}
```

6.3.4 Directed Graphs

Directed graphs have edges with direction, modeling one-way relationships.

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

template<typename T>
class DirectedGraph {
private:
    struct Vertex {
        T data;
        std::vector<Vertex*> outNeighbors;  // Outgoing edges
        std::vector<Vertex*> inNeighbors;   // Incoming edges

        explicit Vertex(const T& value) : data(value) {}
    };

    std::vector<Vertex*> vertices;
    std::unordered_map<T, Vertex*> vertexMap;

public:
    bool addVertex(const T& value) {
        if (vertexMap.count(value)) return false;
        auto* v = new Vertex(value);
        vertices.push_back(v);
        vertexMap[value] = v;
        return true;
    }

    bool addEdge(const T& from, const T& to) {
        if (!vertexMap.count(from) || !vertexMap.count(to)) return false;

        Vertex* fromVertex = vertexMap[from];
        Vertex* toVertex = vertexMap[to];

        fromVertex->outNeighbors.push_back(toVertex);
        toVertex->inNeighbors.push_back(fromVertex);
        return true;
    }

    // Detect cycles using DFS
```

```
    bool hasCycle() const {
        std::unordered_set<const Vertex*> visited;
        std::unordered_set<const Vertex*> recursionStack;

        std::function<bool(const Vertex*)> dfs = [&](const Vertex* v) {
            if (recursionStack.count(v)) return true;
            if (visited.count(v)) return false;

            visited.insert(v);
            recursionStack.insert(v);

            for (auto neighbor : v->outNeighbors) {
                if (dfs(neighbor)) return true;
            }

            recursionStack.erase(v);
            return false;
        };

        for (auto v : vertices) {
            if (!visited.count(v) && dfs(v)) {
                return true;
            }
        }
        return false;
    }

    // Topological sort (Kahn's algorithm)
    std::vector<T> topologicalSort() {
        std::unordered_map<const Vertex*, int> inDegree;
        std::queue<const Vertex*> queue;
        std::vector<T> result;

        // Calculate in-degrees
        for (auto v : vertices) {
            inDegree[v] = v->inNeighbors.size();
            if (inDegree[v] == 0) {
                queue.push(v);
            }
        }

        while (!queue.empty()) {
```

```
            const Vertex* current = queue.front();
            queue.pop();
            result.push_back(current->data);

            for (auto neighbor : current->outNeighbors) {
                if (--inDegree[neighbor] == 0) {
                    queue.push(neighbor);
                }
            }
        }

        if (result.size() != vertices.size()) {
            return {};  // Cycle detected, no topological order
        }
        return result;
    }

    void print() const {
        for (auto v : vertices) {
            std::cout << v->data << " -> ";
            for (auto n : v->outNeighbors) {
                std::cout << n->data << ' ';
            }
            std::cout << "(in: ";
            for (auto n : v->inNeighbors) {
                std::cout << n->data << ' ';
            }
            std::cout << ")\n";
        }
    }

    ~DirectedGraph() {
        for (auto v : vertices) delete v;
    }
};

void directedGraphExample() {
    DirectedGraph<int> graph;

    for (int i = 1; i <= 6; ++i) graph.addVertex(i);

    graph.addEdge(1, 2);
```

```
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);
    graph.addEdge(4, 5);
    graph.addEdge(5, 6);

    std::cout << "Directed Graph:\n";
    graph.print();

    std::cout << "Has cycle: " << (graph.hasCycle() ? "Yes" : "No") << '\n';

    auto order = graph.topologicalSort();
    std::cout << "Topological order: ";
    for (int v : order) std::cout << v << ' ';
    std::cout << '\n';
}
```

6.3.5 Modern C++: Smart Pointers for Graphs

Using smart pointers eliminates manual memory management and prevents leaks.

```
#include <memory>
#include <vector>
#include <unordered_map>

template<typename T>
class SmartGraph {
private:
    struct Vertex {
        T data;
        std::vector<std::shared_ptr<Vertex>> neighbors;
        bool visited{false};

        explicit Vertex(const T& value) : data(value) {}
    };

    std::vector<std::shared_ptr<Vertex>> vertices;
    std::unordered_map<T, std::shared_ptr<Vertex>> vertexMap;

public:
    bool addVertex(const T& value) {
        if (vertexMap.count(value)) return false;
```

```
        auto vertex = std::make_shared<Vertex>(value);
        vertices.push_back(vertex);
        vertexMap[value] = vertex;
        return true;
    }

    bool addEdge(const T& from, const T& to) {
        if (!vertexMap.count(from) || !vertexMap.count(to)) return false;

        vertexMap[from]->neighbors.push_back(vertexMap[to]);
        return true;
    }

    void bfs(const T& start) {
        auto it = vertexMap.find(start);
        if (it == vertexMap.end()) return;

        std::queue<std::shared_ptr<Vertex>> queue;
        queue.push(it->second);
        it->second->visited = true;

        while (!queue.empty()) {
            auto current = queue.front();
            queue.pop();
            std::cout << current->data << ' ';

            for (auto neighbor : current->neighbors) {
                if (!neighbor->visited) {
                    neighbor->visited = true;
                    queue.push(neighbor);
                }
            }
        }

        // Reset visited flags
        for (auto& v : vertices) v->visited = false;
        std::cout << '\n';
    }

    void print() const {
        for (const auto& vertex : vertices) {
            std::cout << vertex->data << " -> ";
```

```
            for (const auto& neighbor : vertex->neighbors) {
                std::cout << neighbor->data << ' ';
            }
            std::cout << '\n';
        }
    }

    size_t vertexCount() const { return vertices.size(); }
};

void smartGraphExample() {
    SmartGraph<int> graph;

    graph.addVertex(1);
    graph.addVertex(2);
    graph.addVertex(3);
    graph.addVertex(4);

    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    graph.addEdge(2, 4);
    graph.addEdge(3, 4);

    graph.print();
    std::cout << "BFS from 1: ";
    graph.bfs(1);
}
```

6.3.6 Common Graph Algorithms

1. **Cycle Detection**:
    - Undirected graphs: use DFS with parent tracking.
    - Directed graphs: use DFS with recursion stack (as shown above).
2. **Topological Sort**:
    - Kahn's algorithm (in-degree) or DFS-based.
    - Only possible for directed acyclic graphs (DAGs).
3. **Shortest Path**:

- Dijkstra's algorithm (non-negative weights).
- Bellman-Ford (negative weights allowed, detects negative cycles).
- BFS for unweighted graphs.

4. **Minimum Spanning Tree**:
   - Prim's algorithm (dense graphs).
   - Kruskal's algorithm (sparse graphs) using union-find.

6.3.7 Common Pitfalls

1. **Memory Leaks**:

```
// Forgetting to delete vertices in destructor
~Graph() {
    // Missing deletion - memory leak!
}
```

2. **Cyclic Dependencies with Shared Pointers**:

```
// Using shared_ptr in undirected graph creates cycles
struct Vertex {
    std::shared_ptr<Vertex> neighbor;  // Cycle!
};
```

   Solution: Use `std::weak_ptr` or raw pointers for back references.

3. **Duplicate Edges**:

```
// Adding the same edge multiple times
addEdge(1, 2);
addEdge(1, 2);  // Duplicate!
```

   Solution: Check for existing edges before adding.

4. **Infinite Loops in Traversal**:

```
// Without visited tracking, DFS can loop forever in cyclic graphs
void dfs(Vertex* v) {
    dfs(v->neighbor);  // Infinite loop!
}
```

   Solution: Always track visited vertices.

5. **Stack Overflow with Deep Recursion**:

```
// Deep recursive DFS on large graphs
void dfs(Vertex* v) {  // May overflow stack
    for (auto n : v->neighbors) dfs(n);
}
```

Solution: Use iterative DFS with explicit stack.

6.3.8 Summary

- Graphs model relationships between objects using vertices and edges.
- Adjacency list representation is memory-efficient for sparse graphs.
- DFS and BFS are fundamental graph traversal algorithms.
- Weighted graphs support edge weights for shortest path algorithms (Dijkstra).
- Directed graphs model one-way relationships with cycle detection and topological sort.
- Smart pointers can manage graph memory but require care with cycles.
- Common algorithms: cycle detection, topological sort, shortest path, MST.
- Common pitfalls: memory leaks, duplicate edges, infinite loops, stack overflow.

Exercise 19. ***Practice Exercises:***

1. *Implement a function to detect cycles in an undirected graph using DFS with parent tracking.*
2. *Implement Kruskal's algorithm for minimum spanning tree using union-find data structure.*
3. *Create a function that finds all paths between two vertices in a directed graph.*
4. *Implement a topological sort using DFS (post-order traversal) instead of Kahn's algorithm.*
5. *Implement a graph that supports both directed and undirected edges with a single flag.*
6. *Create a function to check if a graph is bipartite using BFS coloring.*
7. *Implement a social network graph with friend recommendations based on mutual friends.*

Chapter 7

Pointers and Advanced C++ Features

7.1 Pointers and Templates

Templates are a powerful feature in C++ that allow you to write generic and reusable code. They enable you to define functions and classes that operate with any data type. When combined with pointers, templates become even more versatile, allowing you to create dynamic and type-safe data structures. This section explores how to use pointers with template classes and functions, and provides a practical example of implementing a generic linked list. Additionally, we will delve into advanced topics such as template specialization, smart pointers with templates, template metaprogramming, variadic templates, type traits, concepts (C++20), and more.

Important

Core Principle: Templates enable compile-time polymorphism and generic programming. When combined with pointers, they create powerful, type-safe abstractions for dynamic data structures.

7.1.1 Template Functions with Pointers

Template functions can operate on any data type, including pointer types. This enables generic algorithms that work with both values and pointers.

Important

Key Concepts:

- **Template Parameters**: Placeholder types (e.g., `typename T`) that are substituted at compile time.
- **Type Deduction**: The compiler automatically deduces template arguments from function arguments.
- **Pointers in Templates**: Can be used as parameters, return types, and within the function body.

Example: Generic Swap Function with Pointers

```
#include <iostream>
#include <memory>
#include <vector>
#include <type_traits>

// Generic swap function using pointers
template<typename T>
void swapValues(T* a, T* b) {
    T temp = *a;
    *a = *b;
    *b = temp;
}

// Generic swap function using references (more idiomatic)
template<typename T>
void swapRefs(T& a, T& b) {
    T temp = std::move(a);
    a = std::move(b);
    b = std::move(temp);
}

// Function template with pointer return type
template<typename T>
T* findMax(T* arr, size_t size) {
    if (size == 0) return nullptr;
    T* max = arr;
```

```
    for (size_t i = 1; i < size; ++i) {
        if (arr[i] > *max) {
            max = &arr[i];
        }
    }
    return max;
}

void templateFunctionExample() {
    int x = 5, y = 10;
    std::cout << "Before swap: x = " << x << ", y = " << y << '\n';
    swapValues(&x, &y);
    std::cout << "After swap: x = " << x << ", y = " << y << '\n';

    double a = 3.14, b = 2.71;
    swapRefs(a, b);
    std::cout << "After reference swap: a = " << a << ", b = " << b << '\n';

    int arr[] = {1, 5, 3, 9, 2, 7};
    int* max = findMax(arr, 6);
    if (max) {
        std::cout << "Maximum value: " << *max << '\n';
    }
}
```

7.1.2 Template Classes with Pointers

Template classes can manage dynamic memory and create generic data structures. A generic linked list demonstrates this power.

Important

Template Class Design:

- Define the class template with `template<typename T>`.
- Use `T` as the data type for stored values.
- Implement member functions that work with any type.
- Consider providing specializations for specific types when needed.

Complete Generic Linked List with Smart Pointers

```
#include <iostream>
#include <memory>
#include <optional>
#include <initializer_list>

template<typename T>
class GenericLinkedList {
private:
    struct Node {
        T data;
        std::unique_ptr<Node> next;

        explicit Node(const T& value) : data(value), next(nullptr) {}
        explicit Node(T&& value) : data(std::move(value)), next(nullptr) {}
    };

    std::unique_ptr<Node> head;
    size_t listSize{0};

public:
    GenericLinkedList() = default;

    // Constructor with initializer list
    GenericLinkedList(std::initializer_list<T> init) {
        for (const auto& value : init) {
            append(value);
        }
    }

    // Add node at the end
    void append(const T& value) {
        auto newNode = std::make_unique<Node>(value);

        if (!head) {
            head = std::move(newNode);
        } else {
            Node* current = head.get();
            while (current->next) {
                current = current->next.get();
            }
```

```
            current->next = std::move(newNode);
        }
        ++listSize;
    }

    // Add node at the beginning
    void prepend(const T& value) {
        auto newNode = std::make_unique<Node>(value);
        newNode->next = std::move(head);
        head = std::move(newNode);
        ++listSize;
    }

    // Delete first occurrence of value
    bool remove(const T& value) {
        if (!head) return false;

        if (head->data == value) {
            head = std::move(head->next);
            --listSize;
            return true;
        }

        Node* current = head.get();
        while (current->next && current->next->data != value) {
            current = current->next.get();
        }

        if (current->next) {
            current->next = std::move(current->next->next);
            --listSize;
            return true;
        }
        return false;
    }

    // Find value and return pointer to it (non-owning)
    T* find(const T& value) {
        Node* current = head.get();
        while (current) {
            if (current->data == value) {
                return &current->data;
```

```
            }
            current = current->next.get();
        }
        return nullptr;
    }

    // Check if value exists
    bool contains(const T& value) const {
        Node* current = head.get();
        while (current) {
            if (current->data == value) return true;
            current = current->next.get();
        }
        return false;
    }

    // Get size
    size_t size() const { return listSize; }

    // Check if empty
    bool empty() const { return head == nullptr; }

    // Forward iterator (simplified)
    class Iterator {
        Node* current;
    public:
        explicit Iterator(Node* node) : current(node) {}

        T& operator*() { return current->data; }
        T* operator->() { return &current->data; }

        Iterator& operator++() {
            if (current) current = current->next.get();
            return *this;
        }

        bool operator!=(const Iterator& other) const {
            return current != other.current;
        }
    };

    Iterator begin() { return Iterator(head.get()); }
```

```
    Iterator end() { return Iterator(nullptr); }

    // Print the list
    void print() const {
        Node* current = head.get();
        while (current) {
            std::cout << current->data << " -> ";
            current = current->next.get();
        }
        std::cout << "nullptr\n";
    }
};

void genericLinkedListExample() {
    GenericLinkedList<int> intList{10, 20, 30};
    intList.prepend(5);
    intList.append(40);

    std::cout << "Integer list: ";
    intList.print();

    intList.remove(20);
    std::cout << "After removing 20: ";
    intList.print();

    GenericLinkedList<std::string> stringList{"Hello", "World"};
    stringList.prepend("C++");
    stringList.append("!");

    std::cout << "String list: ";
    stringList.print();

    // Range-based for loop with custom iterator
    std::cout << "Iterating: ";
    for (const auto& value : stringList) {
        std::cout << value << ' ';
    }
    std::cout << '\n';
}
```

7.1.3 Template Specialization

Template specialization allows you to define specific implementations for particular types.

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

// Primary template
template<typename T>
class Printer {
public:
    static void print(const T& value) {
        std::cout << value << '\n';
    }
};

// Full specialization for const char*
template<>
class Printer<const char*> {
public:
    static void print(const char* value) {
        std::cout << "String: \"" << value << "\"\n";
    }
};

// Partial specialization for pointer types
template<typename T>
class Printer<T*> {
public:
    static void print(T* value) {
        std::cout << "Pointer to: ";
        if (value) {
            Printer<T>::print(*value);
        } else {
            std::cout << "nullptr\n";
        }
    }
};

void templateSpecializationExample() {
    Printer<int>::print(42);
    Printer<const char*>::print("Hello");
```

```
    int x = 100;
    int* ptr = &x;
    Printer<int*>::print(ptr);

    int* nullPtr = nullptr;
    Printer<int*>::print(nullPtr);
}
```

7.1.4 Smart Pointers with Templates

Smart pointers work seamlessly with templates to create safe, self-managing generic data structures.

```
#include <memory>
#include <vector>
#include <functional>

template<typename T>
using UniquePtr = std::unique_ptr<T>;

template<typename T, typename... Args>
UniquePtr<T> makeUnique(Args&&... args) {
    return std::make_unique<T>(std::forward<Args>(args)...);
}

template<typename T>
class SmartContainer {
private:
    std::vector<std::unique_ptr<T>> items;

public:
    void add(std::unique_ptr<T> item) {
        items.push_back(std::move(item));
    }

    template<typename U, typename... Args>
    void emplace(Args&&... args) {
        items.push_back(std::make_unique<U>(std::forward<Args>(args)...));
    }

    void process(std::function<void(T&)> processor) {
        for (auto& item : items) {
```

```
            if (item) {
                processor(*item);
            }
        }
    }

    size_t size() const { return items.size(); }
};

class Widget {
    int id;
public:
    explicit Widget(int i) : id(i) {}
    void show() const { std::cout << "Widget " << id << '\n'; }
};

void smartPointerTemplatesExample() {
    SmartContainer<Widget> container;
    container.emplace<Widget>(1);
    container.emplace<Widget>(2);
    container.add(std::make_unique<Widget>(3));

    container.process([](Widget& w) { w.show(); });
}
```

7.1.5 Variadic Templates

Variadic templates accept a variable number of arguments, enabling powerful generic functions.

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

// Base case for recursion
void print() {
    std::cout << '\n';
}

// Variadic template function
template<typename T, typename... Args>
void print(T first, Args... args) {
    std::cout << first;
```

```
    if constexpr (sizeof...(args) > 0) {
        std::cout << ", ";
    }
    print(args...);
}

// Build a string from variadic arguments
template<typename... Args>
std::string format(const std::string& fmt, Args... args) {
    std::ostringstream oss;
    std::vector<std::string> argStrs = {toString(args)...};
    // Simplified formatting (not a full implementation)
    size_t pos = 0;
    size_t idx = 0;
    std::string result = fmt;
    while ((pos = result.find("{}", pos)) != std::string::npos && idx <
    ↪  argStrs.size()) {
        result.replace(pos, 2, argStrs[idx++]);
        pos += argStrs[idx - 1].length();
    }
    return result;
}

template<typename T>
std::string toString(const T& value) {
    std::ostringstream oss;
    oss << value;
    return oss.str();
}

// Perfect forwarding with variadic templates
template<typename Func, typename... Args>
auto invoke(Func&& func, Args&&... args) -> decltype(auto) {
    return std::forward<Func>(func)(std::forward<Args>(args)...);
}

void variadicTemplateExample() {
    print(1, 2.5, "Hello", 'A', 42);

    auto result = invoke([](int a, int b) { return a + b; }, 10, 20);
    std::cout << "Invoke result: " << result << '\n';
}
```

7.1.6 Type Traits with Templates

Type traits provide compile-time information about types, enabling conditional compilation and optimizations.

```
#include <iostream>
#include <type_traits>
#include <vector>

// Enable only for integral types
template<typename T>
std::enable_if_t<std::is_integral_v<T>, T>
doubleValue(T value) {
    return value * 2;
}

// Overload for floating point types
template<typename T>
std::enable_if_t<std::is_floating_point_v<T>, T>
doubleValue(T value) {
    return value * 2.0;
}

// Type trait check for pointer types
template<typename T>
void processPointer(T ptr) {
    if constexpr (std::is_pointer_v<T>) {
        if (ptr) {
            std::cout << "Pointer value: " << *ptr << '\n';
        } else {
            std::cout << "Null pointer\n";
        }
    } else {
        std::cout << "Not a pointer: " << ptr << '\n';
    }
}

// Check if type is copy constructible
template<typename T>
class Container {
    static_assert(std::is_copy_constructible_v<T>,
                  "T must be copy constructible");
```

```
    std::vector<T> data;
public:
    void add(const T& value) { data.push_back(value); }
};

void typeTraitsExample() {
    std::cout << doubleValue(5) << '\n';      // 10
    std::cout << doubleValue(3.14) << '\n';   // 6.28

    int x = 42;
    processPointer(&x);   // Pointer value: 42
    processPointer(x);    // Not a pointer: 42

    std::cout << "Is int pointer? " << std::is_pointer<int*>::value << '\n';
    std::cout << "Is int reference? " << std::is_reference<int&>::value << '\n';
}
```

7.1.7 Concepts (C++20)

Concepts constrain template parameters, making templates more expressive and providing better error messages.

```
#include <iostream>
#include <concepts>
#include <vector>
#include <algorithm>

// Define a concept
template<typename T>
concept Numeric = std::is_arithmetic_v<T>;

template<typename T>
concept Container = requires(T t) {
    { t.begin() } -> std::input_iterator;
    { t.end() } -> std::input_iterator;
    { t.size() } -> std::integral;
};

// Function template constrained by concept
template<Numeric T>
T add(T a, T b) {
    return a + b;
```

```
}

// Function template with requires clause
template<typename T>
requires Numeric<T>
T multiply(T a, T b) {
    return a * b;
}

// Concept for types that support pointer arithmetic
template<typename T>
concept PointerType = std::is_pointer_v<T>;

template<PointerType P>
void advancePtr(P& ptr, std::ptrdiff_t n) {
    ptr += n;
}

// Concept for containers with pointer-like access
template<typename C>
concept ContiguousContainer = requires(C c, size_t i) {
    { c.data() } -> std::same_as<typename C::pointer>;
    { c[i] } -> std::convertible_to<typename C::value_type>;
};

template<ContiguousContainer C>
void processContiguous(C& container) {
    auto* ptr = container.data();
    for (size_t i = 0; i < container.size(); ++i) {
        ptr[i] = static_cast<typename C::value_type>(i);
    }
}

void conceptsExample() {
    std::cout << add(10, 20) << '\n';         // OK
    std::cout << multiply(3.14, 2.0) << '\n'; // OK

    std::vector<int> vec(5);
    processContiguous(vec);
    for (int v : vec) std::cout << v << ' ';
    std::cout << '\n';
```

```
    // int* ptr = nullptr;
    // advancePtr(ptr, 5);  // OK - int* is a pointer type
}
```

7.1.8 Template Metaprogramming

Template metaprogramming performs computations at compile time using templates.

```
#include <iostream>
#include <type_traits>

// Compile-time factorial
template<size_t N>
struct Factorial {
    static constexpr size_t value = N * Factorial<N - 1>::value;
};

template<>
struct Factorial<0> {
    static constexpr size_t value = 1;
};

// Compile-time Fibonacci
template<size_t N>
struct Fibonacci {
    static constexpr size_t value = Fibonacci<N - 1>::value + Fibonacci<N -
    ↪  2>::value;
};

template<>
struct Fibonacci<0> {
    static constexpr size_t value = 0;
};

template<>
struct Fibonacci<1> {
    static constexpr size_t value = 1;
};

// Type list operations (compile-time)
template<typename... Types>
struct TypeList {};
```

```
template<typename List>
struct Length;

template<typename... Types>
struct Length<TypeList<Types...>> {
    static constexpr size_t value = sizeof...(Types);
};

// Compile-time selection based on type
template<bool Condition, typename TrueType, typename FalseType>
struct Conditional {
    using type = TrueType;
};

template<typename TrueType, typename FalseType>
struct Conditional<false, TrueType, FalseType> {
    using type = FalseType;
};

// Compile-time integer to type mapping
template<int N>
struct IntToType {
    static constexpr int value = N;
};

void templateMetaprogrammingExample() {
    std::cout << "Factorial<5>::value = " << Factorial<5>::value << '\n';
    std::cout << "Fibonacci<10>::value = " << Fibonacci<10>::value << '\n';

    using MyList = TypeList<int, double, char, std::string>;
    std::cout << "Length of type list: " << Length<MyList>::value << '\n';

    using IntType = Conditional<true, int, double>::type;
    using DoubleType = Conditional<false, int, double>::type;
    std::cout << "Conditional types: " << std::is_same_v<IntType, int> << ' '
              << std::is_same_v<DoubleType, double> << '\n';
}
```

7.1.9 Template Aliases and Variable Templates

Template aliases simplify complex template types, and variable templates provide compile-time values.

```
#include <iostream>
#include <vector>
#include <map>
#include <string>

// Template alias for common container types
template<typename T>
using Vec = std::vector<T>;

template<typename Key, typename Value>
using Map = std::map<Key, Value>;

template<typename T>
using Ptr = T*;

// Variable template for compile-time constants
template<typename T>
constexpr bool is_pointer_v = std::is_pointer<T>::value;

template<typename T>
constexpr T pi = T(3.1415926535897932385);

// Template alias for smart pointer
template<typename T>
using Unique = std::unique_ptr<T>;

// Template alias for function pointer
template<typename R, typename... Args>
using FuncPtr = R(*)(Args...);

void templateAliasExample() {
    Vec<int> intVec = {1, 2, 3, 4, 5};
    Map<std::string, int> strIntMap = {{"one", 1}, {"two", 2}};

    int x = 42;
    Ptr<int> ptr = &x;
    std::cout << "Pointer value: " << *ptr << '\n';
```

```
    std::cout << "is_pointer_v<int*>: " << is_pointer_v<int*> << '\n';
    std::cout << "is_pointer_v<int>: " << is_pointer_v<int> << '\n';

    std::cout << "pi<float>: " << pi<float> << '\n';
    std::cout << "pi<double>: " << pi<double> << '\n';

    Unique<int> uniqueInt = std::make_unique<int>(100);
    std::cout << "Unique ptr value: " << *uniqueInt << '\n';
}
```

7.1.10 Common Pitfalls

1. **Template Code Bloat**:
   - Each instantiation generates separate code.
   - Solution: Factor out non-dependent code into base classes.
2. **Complex Error Messages**:
   - Template errors can be difficult to read.
   - Solution: Use concepts (C++20) to constrain templates.
3. **Mixing Pointers and References Incorrectly**:

   ```
   template<typename T>
   void process(T value) {
       // value might be pointer, reference, or value type
       // Need to handle each case carefully
   }
   ```

   Solution: Use type traits or concepts to detect pointer/reference types.
4. **Unintended Type Deduction**:

   ```
   template<typename T>
   void func(T* ptr) { }  // Only works with pointers

   func(arr);  // OK: array decays to pointer
   func(42);   // Error: int is not a pointer
   ```

   Solution: Use overloads or SFINAE/Concepts to handle different cases.

5. **Incomplete Type Specializations**:

```
// Missing specialization for const char*
template<typename T>
void print(T value) { std::cout << value; }
// Specialization needed for string literals
```

7.1.11 Summary

- Templates enable generic programming with compile-time polymorphism.
- Template functions and classes with pointers create flexible, type-safe data structures.
- Template specialization provides custom implementations for specific types.
- Smart pointers integrate seamlessly with templates for safe memory management.
- Variadic templates handle variable numbers of arguments for flexible APIs.
- Type traits provide compile-time type information for conditional code.
- Concepts (C++20) constrain template parameters for better error messages.
- Template metaprogramming enables compile-time computations.
- Best practices: use concepts for constraints, prefer type traits for type inspection, and leverage variadic templates for generic interfaces.

Exercise 20. ***Practice Exercises:***

1. *Implement a generic binary search tree using templates and smart pointers.*
2. *Create a variadic template function that computes the sum of all arguments.*
3. *Implement a type trait that checks if a type has a member function named* `size()`.
4. *Write a template function that works only with container types (using C++20 concepts).*
5. *Implement a generic* `optional<T>` *class that can hold a value or nothing.*
6. *Create a template metaprogram to compute the greatest common divisor (GCD) at compile time.*
7. *Implement a generic* `apply` *function that invokes a callable with elements from a tuple.*

7.2 Pointers and Multithreading

Multithreading is a powerful feature in modern C++ that allows you to execute multiple threads concurrently, enabling parallel processing and improved performance. However, sharing data between threads can lead to race conditions and undefined behavior if not handled properly. Pointers play a crucial role in multithreading, as they are often used to share data between threads. This section explores how to share data between threads using pointers and demonstrates how to synchronize access to shared resources using mutexes and other synchronization primitives. Additionally, we will delve into advanced topics such as deadlocks, condition variables, atomic operations, thread-safe data structures, and more.

Important

Core Principle: Shared data accessed by multiple threads must be synchronized. Unsynchronized concurrent access leads to data races and undefined behavior.

7.2.1 Sharing Data Between Threads Using Pointers

In multithreaded programs, threads often need to share data. Pointers are commonly used to pass data between threads, as they allow multiple threads to access the same memory location. However, sharing data between threads can lead to race conditions if multiple threads attempt to access or modify the same data simultaneously without proper synchronization.

Important

Key Concepts:

- **Data Race**: Two or more threads accessing the same memory location concurrently, with at least one writing, without synchronization.
- **Race Condition**: The outcome of a program depends on the timing of uncontrolled concurrent events.
- **Synchronization**: Mechanisms to coordinate thread access to shared resources.

Example: Unsynchronized Shared Data (Data Race)

```
#include <iostream>
```

```cpp
#include <thread>
#include <vector>

void increment(int* counter, int iterations) {
    for (int i = 0; i < iterations; ++i) {
        ++(*counter);  // Data race! Multiple threads accessing without
        ↪  synchronization
    }
}

void unsynchronizedExample() {
    int counter = 0;
    const int iterations = 100000;
    const int numThreads = 10;
    std::vector<std::thread> threads;

    for (int i = 0; i < numThreads; ++i) {
        threads.emplace_back(increment, &counter, iterations);
    }

    for (auto& t : threads) {
        t.join();
    }

    // Expected: 1,000,000, but actual may be less due to data races
    std::cout << "Final counter value: " << counter << '\n';
}
```

7.2.2 Synchronizing Access with Mutexes

The most common synchronization primitive is the **mutex** (mutual exclusion). A mutex ensures that only one thread can access a shared resource at a time.

Important

Mutex Best Practices:

- Use `std::lock_guard` or `std::unique_lock` for RAII-based locking.
- Keep critical sections as short as possible.
- Never hold locks across unknown code (e.g., callbacks).
- Acquire locks in a consistent order to prevent deadlocks.

Example: Synchronizing with Mutex

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>

std::mutex mtx;  // Global mutex for synchronizing access

void increment(int* counter, int iterations) {
    for (int i = 0; i < iterations; ++i) {
        std::lock_guard<std::mutex> lock(mtx);  // RAII lock
        ++(*counter);
    }  // Automatically unlocked
}

void synchronizedMutexExample() {
    int counter = 0;
    const int iterations = 100000;
    const int numThreads = 10;
    std::vector<std::thread> threads;

    for (int i = 0; i < numThreads; ++i) {
        threads.emplace_back(increment, &counter, iterations);
    }

    for (auto& t : threads) {
        t.join();
    }
```

```
    std::cout << "Final counter value: " << counter << '\n';  // 1,000,000
}
```

7.2.3 Atomic Operations

For simple operations like increments, `std::atomic` provides lock-free synchronization with better performance.

```
#include <iostream>
#include <thread>
#include <vector>
#include <atomic>

std::atomic<int> counter{0};

void incrementAtomic(int iterations) {
    for (int i = 0; i < iterations; ++i) {
        ++counter;  // Atomic increment (no data race)
    }
}

void atomicOperationsExample() {
    const int iterations = 100000;
    const int numThreads = 10;
    std::vector<std::thread> threads;

    for (int i = 0; i < numThreads; ++i) {
        threads.emplace_back(incrementAtomic, iterations);
    }

    for (auto& t : threads) {
        t.join();
    }

    std::cout << "Final counter value: " << counter << '\n';  // 1,000,000
}
```

7.2.4 C++20: `std::jthread` and `std::stop_token`

C++20 introduces `std::jthread`, a joining thread that automatically joins on destruction, and `std::stop_token` for cooperative cancellation.

```cpp
#include <iostream>
#include <thread>
#include <atomic>
#include <chrono>

void worker(std::stop_token stoken, std::atomic<bool>& running) {
    while (!stoken.stop_requested()) {
        // Do work
        running = true;
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
    running = false;
    std::cout << "Worker thread stopped\n";
}

void jthreadExample() {
    std::atomic<bool> running{false};

    // jthread automatically joins on destruction
    std::jthread t(worker, std::ref(running));

    std::this_thread::sleep_for(std::chrono::seconds(1));

    // Request stop
    t.request_stop();

    // No need to call join() - jthread handles it
    std::cout << "Main thread exiting\n";
}
```

7.2.5 Deadlocks and How to Avoid Them

A deadlock occurs when two or more threads are blocked forever, waiting for each other to release locks.

Important

Deadlock Prevention Strategies:

- Acquire locks in a consistent global order.
- Use `std::lock` to acquire multiple locks atomically.
- Use `std::try_lock` with backoff.
- Avoid holding locks while calling unknown code.
- Use lock-free data structures when possible.

Example: Deadlock Scenario

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx1, mtx2;

void thread1() {
    std::lock_guard<std::mutex> lock1(mtx1);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock2(mtx2);  // Deadlock if thread2 holds mtx2
    std::cout << "Thread 1 finished\n";
}

void thread2() {
    std::lock_guard<std::mutex> lock2(mtx2);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::lock_guard<std::mutex> lock1(mtx1);  // Deadlock if thread1 holds mtx1
    std::cout << "Thread 2 finished\n";
}

void deadlockExample() {
    std::thread t1(thread1);
    std::thread t2(thread2);

    t1.join();
    t2.join();
}
```

Solution: Use `std::lock` to acquire multiple locks atomically

```
void safeThread1() {
    std::unique_lock<std::mutex> lock1(mtx1, std::defer_lock);
    std::unique_lock<std::mutex> lock2(mtx2, std::defer_lock);
    std::lock(lock1, lock2);  // Acquire both locks atomically
    std::cout << "Thread 1 finished\n";
}

void safeThread2() {
    std::unique_lock<std::mutex> lock1(mtx1, std::defer_lock);
    std::unique_lock<std::mutex> lock2(mtx2, std::defer_lock);
    std::lock(lock1, lock2);  // Same order guarantees no deadlock
    std::cout << "Thread 2 finished\n";
}
```

7.2.6 Condition Variables

Condition variables allow threads to wait for a condition to become true, avoiding busy-waiting.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <condition_variable>
#include <queue>

class ThreadSafeQueue {
    std::queue<int> queue;
    std::mutex mtx;
    std::condition_variable cv;

public:
    void push(int value) {
        {
            std::lock_guard<std::mutex> lock(mtx);
            queue.push(value);
        }
        cv.notify_one();  // Notify waiting thread
    }

    bool pop(int& value, std::chrono::milliseconds timeout =
    ↪  std::chrono::milliseconds::max()) {
```

```
        std::unique_lock<std::mutex> lock(mtx);

        if (timeout == std::chrono::milliseconds::max()) {
            cv.wait(lock, [this] { return !queue.empty(); });
        } else {
            if (!cv.wait_for(lock, timeout, [this] { return !queue.empty(); })) {
                return false;  // Timeout
            }
        }

        value = queue.front();
        queue.pop();
        return true;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(mtx);
        return queue.empty();
    }
};

void conditionVariableExample() {
    ThreadSafeQueue queue;
    std::atomic<bool> done{false};

    // Producer
    std::thread producer([&queue, &done] {
        for (int i = 0; i < 10; ++i) {
            queue.push(i);
            std::cout << "Produced: " << i << '\n';
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
        }
        done = true;
    });

    // Consumer
    std::thread consumer([&queue, &done] {
        int value;
        while (!done || !queue.empty()) {
            if (queue.pop(value, std::chrono::milliseconds(500))) {
                std::cout << "Consumed: " << value << '\n';
            }
```

```
        }
    });

    producer.join();
    consumer.join();
}
```

7.2.7 Thread-Safe Data Structures with Smart Pointers

Combining smart pointers with synchronization creates safe, self-managing concurrent data structures.

```
#include <memory>
#include <mutex>
#include <unordered_map>
#include <optional>

template<typename Key, typename Value>
class ThreadSafeCache {
private:
    struct Entry {
        Value value;
        std::chrono::steady_clock::time_point expiry;
    };

    std::unordered_map<Key, std::shared_ptr<Entry>> cache;
    mutable std::shared_mutex rwMutex;  // C++17: shared_mutex for readers/writers
    std::chrono::seconds defaultTTL;

public:
    explicit ThreadSafeCache(std::chrono::seconds ttl = std::chrono::seconds(60))
        : defaultTTL(ttl) {}

    void put(const Key& key, const Value& value) {
        auto entry = std::make_shared<Entry>();
        entry->value = value;
        entry->expiry = std::chrono::steady_clock::now() + defaultTTL;

        std::unique_lock lock(rwMutex);
        cache[key] = entry;
    }
```

```
    std::optional<Value> get(const Key& key) {
        std::shared_lock lock(rwMutex);  // Shared lock for reading

        auto it = cache.find(key);
        if (it == cache.end()) {
            return std::nullopt;
        }

        auto entry = it->second;
        if (std::chrono::steady_clock::now() > entry->expiry) {
            lock.unlock();  // Upgrade to unique lock for deletion
            std::unique_lock uniqueLock(rwMutex);

            // Check again after acquiring unique lock
            if (cache.find(key) != cache.end() &&
                std::chrono::steady_clock::now() > cache[key]->expiry) {
                cache.erase(key);
                return std::nullopt;
            }
            return entry->value;
        }

        return entry->value;
    }

    void cleanExpired() {
        std::unique_lock lock(rwMutex);
        auto now = std::chrono::steady_clock::now();

        for (auto it = cache.begin(); it != cache.end();) {
            if (now > it->second->expiry) {
                it = cache.erase(it);
            } else {
                ++it;
            }
        }
    }
};

void threadSafeCacheExample() {
    ThreadSafeCache<int, std::string> cache(std::chrono::seconds(2));
```

```
    std::jthread writer([&cache] {
        cache.put(1, "Hello");
        cache.put(2, "World");
    });

    std::jthread reader([&cache] {
        for (int i = 0; i < 5; ++i) {
            if (auto val = cache.get(1)) {
                std::cout << "Got: " << *val << '\n';
            } else {
                std::cout << "Not found or expired\n";
            }
            std::this_thread::sleep_for(std::chrono::seconds(1));
        }
    });

    writer.join();
    reader.join();
}
```

7.2.8 Thread Pools with Modern C++

A thread pool manages a collection of worker threads for executing tasks concurrently.

```
#include <iostream>
#include <thread>
#include <vector>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <functional>
#include <future>
#include <type_traits>

class ThreadPool {
private:
    std::vector<std::jthread> workers;
    std::queue<std::function<void()>> tasks;
    std::mutex mtx;
    std::condition_variable cv;
    bool stop;
```

```
public:
    explicit ThreadPool(size_t numThreads) : stop(false) {
        for (size_t i = 0; i < numThreads; ++i) {
            workers.emplace_back([this](std::stop_token stoken) {
                while (!stoken.stop_requested()) {
                    std::function<void()> task;
                    {
                        std::unique_lock<std::mutex> lock(mtx);
                        cv.wait(lock, [this] { return stop || !tasks.empty(); });

                        if (stop && tasks.empty()) {
                            return;
                        }

                        task = std::move(tasks.front());
                        tasks.pop();
                    }
                    task();
                }
            });
        }
    }

    template<typename F, typename... Args>
    auto enqueue(F&& f, Args&&... args) -> std::future<typename
    ↪  std::invoke_result_t<F, Args...>> {
        using return_type = typename std::invoke_result_t<F, Args...>;

        auto task = std::make_shared<std::packaged_task<return_type()>>(
            std::bind(std::forward<F>(f), std::forward<Args>(args)...)
        );

        std::future<return_type> result = task->get_future();

        {
            std::lock_guard<std::mutex> lock(mtx);
            if (stop) {
                throw std::runtime_error("Enqueue on stopped ThreadPool");
            }
            tasks.emplace([task]() { (*task)(); });
        }
```

```
        cv.notify_one();
        return result;
    }

    ~ThreadPool() {
        {
            std::lock_guard<std::mutex> lock(mtx);
            stop = true;
        }
        cv.notify_all();
        // jthread automatically joins on destruction
    }

    size_t pendingTasks() const {
        std::lock_guard<std::mutex> lock(mtx);
        return tasks.size();
    }
};

void threadPoolExample() {
    ThreadPool pool(4);
    std::vector<std::future<int>> results;

    // Submit tasks
    for (int i = 0; i < 10; ++i) {
        results.emplace_back(pool.enqueue([i] {
            std::this_thread::sleep_for(std::chrono::milliseconds(100));
            return i * i;
        }));
    }

    // Collect results
    for (auto& result : results) {
        std::cout << "Result: " << result.get() << '\n';
    }

    std::cout << "All tasks completed\n";
}
```

7.2.9 Thread Local Storage

Thread local storage (TLS) provides each thread with its own instance of a variable.

```
#include <iostream>
#include <thread>
#include <vector>
#include <random>

thread_local int threadLocalCounter = 0;
thread_local std::mt19937 rng{std::random_device{}()};

void incrementThreadLocal(int iterations) {
    for (int i = 0; i < iterations; ++i) {
        ++threadLocalCounter;
    }
    std::cout << "Thread " << std::this_thread::get_id()
              << ": " << threadLocalCounter << '\n';
}

void threadLocalStorageExample() {
    const int iterations = 1000;
    const int numThreads = 5;
    std::vector<std::jthread> threads;

    for (int i = 0; i < numThreads; ++i) {
        threads.emplace_back(incrementThreadLocal, iterations);
    }

    // Each thread has its own counter, no synchronization needed
    std::cout << "Main thread counter: " << threadLocalCounter << '\n';
}
```

7.2.10 C++23: `std::mdspan` for Multithreaded Array Access

C++23's `std::mdspan` provides safe multi-dimensional array views for parallel algorithms.

```
#include <mdspan>
#include <vector>
#include <algorithm>
#include <execution>

void mdspanParallelExample() {
    const size_t rows = 1000, cols = 1000;
    std::vector<double> data(rows * cols);
```

```
    // Create a 2D view
    std::mdspan<double, std::extents<size_t, std::dynamic_extent,
    ↪  std::dynamic_extent>>
        matrix(data.data(), rows, cols);

    // Parallel initialization
    std::for_each(std::execution::par, data.begin(), data.end(),
                  [i = 0](double& val) mutable { val = i++; });

    // Parallel transformation
    std::for_each(std::execution::par, data.begin(), data.end(),
                  [](double& val) { val = std::sqrt(val); });
}
```

7.2.11 Common Pitfalls and Best Practices

1. **Data Races**:

   ```
   // Wrong: unsynchronized access
   int counter;
   void increment() { ++counter; }  // Data race!
   ```

   Solution: Use mutex, atomic, or thread-safe data structures.

2. **Deadlocks**:

   ```
   // Wrong: inconsistent lock order
   void f1() { lock(m1); lock(m2); }
   void f2() { lock(m2); lock(m1); }  // Potential deadlock
   ```

   Solution: Acquire locks in consistent order or use `std::lock`.

3. **Spurious Wakeups**:

   ```
   // Wrong: not checking condition
   cv.wait(lock);  // May wake up spuriously
   ```

   Solution: Always use predicate with wait:

   ```
   cv.wait(lock, []{ return condition; });
   ```

4. **Holding Locks Too Long**:

```
// Wrong: slow operation while holding lock
lock();
slow_operation();  // Blocks other threads unnecessarily
unlock();
```

Solution: Release lock before slow operations.

5. **Using Raw Pointers for Shared Data**:

```
// Wrong: lifetime management issues
int* data = new int(42);
std::thread t([data] { process(data); });
delete data;  // Use after delete!
```

Solution: Use `std::shared_ptr` with proper lifetime management.

7.2.12 Summary

- Pointers are used to share data between threads but require synchronization to avoid data races.
- Mutexes (`std::mutex`) with RAII locks (`std::lock_guard`, `std::unique_lock`) provide mutual exclusion.
- Atomic operations (`std::atomic`) offer lock-free synchronization for simple operations.
- C++20's `std::jthread` simplifies thread management with automatic joining and cancellation.
- Condition variables enable efficient waiting for state changes.
- Thread pools manage worker threads for task-based parallelism.
- Thread local storage provides per-thread isolation.
- Best practices: use RAII for locks, consistent lock ordering, avoid deadlocks, and prefer atomic operations when possible.

Exercise 21. ***Practice Exercises:***

1. *Implement a thread-safe counter using `std::atomic` and compare its performance with mutex-based version.*

2. *Create a producer-consumer queue using* `std::condition_variable` *with timeout support.*

3. *Implement a thread pool that can return* `std::future` *results and handle exceptions.*

4. *Design a thread-safe LRU cache with read-write locks (*`std::shared_mutex`*).*

5. *Use* `std::jthread` *with stop tokens to implement a graceful shutdown mechanism.*

6. *Implement a parallel matrix multiplication using* `std::async` *or thread pool.*

7. *Create a thread-safe singleton using* `std::call_once` *and* `std::once_flag`*.*

7.3 Pointers and Move Semantics

Move semantics is a cornerstone of modern C++ programming, introduced in C++11 to optimize resource management. It allows for the efficient transfer of resources, such as dynamically allocated memory, file handles, or network connections, from one object to another. This eliminates unnecessary copying and significantly improves performance, especially in resource-intensive applications. Pointers are central to implementing move semantics, as they enable the transfer of ownership of resources without the overhead of deep copying. This section delves into the intricacies of using pointers with move constructors and move assignment operators, provides practical examples of move-aware classes, and explores advanced topics like the Rule of Five, smart pointers, perfect forwarding, and move semantics in STL containers.

Important

Core Principle: Move semantics transfers ownership of resources instead of copying them. Pointers are the mechanism for transferring this ownership efficiently.

7.3.1 Move Semantics Fundamentals

Move semantics is implemented through two special member functions: the **move constructor** and the **move assignment operator**. These functions allow an object to ”steal” resources from another object, leaving the source object in a valid but unspecified state (typically empty).

Important

Key Concepts:

- **rvalue references** (`&&`): Bind to temporary objects and indicate that resources can be moved.
- `std::move`: Casts an object to an rvalue reference, enabling move semantics.
- **Move Constructor**: Transfers resources from a source object to a newly created object.
- **Move Assignment Operator**: Transfers resources from a source object to an existing object.
- **Valid but Unspecified State**: After a move, the source object must be destructible and assignable, but its contents are unspecified.

Basic Move-Aware Class with Raw Pointers

```
#include <iostream>
#include <utility>
#include <algorithm>

class Buffer {
private:
    int* data;
    size_t size;

public:
    // Constructor
    explicit Buffer(size_t sz) : size(sz), data(new int[sz]) {
        std::cout << "Buffer allocated with size " << size << '\n';
    }

    // Destructor
    ~Buffer() {
        delete[] data;
        std::cout << "Buffer deallocated\n";
    }

    // Move Constructor (noexcept is essential for optimization)
    Buffer(Buffer&& other) noexcept
```

```
        : data(other.data), size(other.size) {
        other.data = nullptr;
        other.size = 0;
        std::cout << "Buffer moved (constructor)\n";
    }

    // Move Assignment Operator
    Buffer& operator=(Buffer&& other) noexcept {
        if (this != &other) {
            delete[] data;                    // Free existing resource
            data = other.data;                // Transfer ownership
            size = other.size;
            other.data = nullptr;             // Leave source in valid state
            other.size = 0;
            std::cout << "Buffer moved (assignment)\n";
        }
        return *this;
    }

    // Copy operations are deleted (move-only type)
    Buffer(const Buffer&) = delete;
    Buffer& operator=(const Buffer&) = delete;

    // Accessors
    int* getData() const { return data; }
    size_t getSize() const { return size; }

    // Fill with value
    void fill(int value) {
        for (size_t i = 0; i < size; ++i) {
            data[i] = value;
        }
    }

    // Check if valid
    bool isValid() const { return data != nullptr; }
};

void moveSemanticsExample() {
    Buffer buf1(10);
    buf1.fill(42);
```

```
    Buffer buf2(std::move(buf1));  // Move constructor
    std::cout << "buf1 valid: " << buf1.isValid() << '\n';
    std::cout << "buf2 size: " << buf2.getSize() << '\n';

    Buffer buf3(5);
    buf3 = std::move(buf2);        // Move assignment
    std::cout << "buf2 valid: " << buf2.isValid() << '\n';
    std::cout << "buf3 size: " << buf3.getSize() << '\n';
}
```

7.3.2 The Rule of Five

The **Rule of Five** states that if a class defines any of the following special member functions, it should explicitly define all of them:

1. Destructor
2. Copy Constructor
3. Copy Assignment Operator
4. Move Constructor
5. Move Assignment Operator

This ensures proper resource management and avoids issues like memory leaks, double deletions, or undefined behavior.

Important

Rule of Five Guidelines:

- Define all five if you manage a resource (e.g., raw pointers, file handles).
- Use `= default` for default implementations when possible.
- Mark move operations as `noexcept` for optimal performance.
- For move-only types, delete copy operations.

Complete Rule of Five Example

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

class StringBuffer {
private:
    char* data;
    size_t length;

    // Helper for copy operations
    void copyFrom(const char* src, size_t len) {
        data = new char[len + 1];
        std::copy(src, src + len, data);
        data[len] = '\0';
        length = len;
    }

public:
    // Default constructor
    StringBuffer() : data(nullptr), length(0) {}

    // Constructor from C-string
    explicit StringBuffer(const char* str) {
        length = std::strlen(str);
        data = new char[length + 1];
        std::copy(str, str + length, data);
        data[length] = '\0';
        std::cout << "Constructed: " << data << '\n';
    }

    // Destructor
    ~StringBuffer() {
        delete[] data;
        std::cout << "Destroyed: " << (data ? data : "empty") << '\n';
    }

    // Copy Constructor (deep copy)
    StringBuffer(const StringBuffer& other) : length(other.length) {
        if (other.data) {
            copyFrom(other.data, other.length);
        } else {
            data = nullptr;
```

```
        }
        std::cout << "Copied: " << (data ? data : "empty") << '\n';
    }

    // Copy Assignment Operator
    StringBuffer& operator=(const StringBuffer& other) {
        if (this != &other) {
            delete[] data;
            if (other.data) {
                copyFrom(other.data, other.length);
            } else {
                data = nullptr;
                length = 0;
            }
            std::cout << "Copy assigned: " << (data ? data : "empty") << '\n';
        }
        return *this;
    }

    // Move Constructor (noexcept for optimization)
    StringBuffer(StringBuffer&& other) noexcept
        : data(other.data), length(other.length) {
        other.data = nullptr;
        other.length = 0;
        std::cout << "Moved (ctor): " << (data ? data : "empty") << '\n';
    }

    // Move Assignment Operator
    StringBuffer& operator=(StringBuffer&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            length = other.length;
            other.data = nullptr;
            other.length = 0;
            std::cout << "Moved (assign): " << (data ? data : "empty") << '\n';
        }
        return *this;
    }

    // Accessors
    const char* c_str() const { return data ? data : ""; }
```

```
    size_t size() const { return length; }
    bool empty() const { return length == 0; }
};

void ruleOfFiveExample() {
    StringBuffer s1("Hello");
    StringBuffer s2(s1);                // Copy constructor
    StringBuffer s3(std::move(s1));     // Move constructor

    StringBuffer s4("World");
    s4 = s2;                            // Copy assignment
    s4 = std::move(s3);                 // Move assignment

    std::cout << "s1: " << s1.c_str() << '\n';
    std::cout << "s2: " << s2.c_str() << '\n';
    std::cout << "s3: " << s3.c_str() << '\n';
    std::cout << "s4: " << s4.c_str() << '\n';
}
```

7.3.3 Smart Pointers and Move Semantics

Smart pointers like `std::unique_ptr` and `std::shared_ptr` inherently support move semantics, making them ideal for managing dynamically allocated resources.

Important

Smart Pointer Move Semantics:

- `std::unique_ptr` is move-only; it cannot be copied.
- `std::shared_ptr` supports both copy and move, but move is more efficient.
- `std::weak_ptr` is also movable.

Example: Smart Pointers and Move Semantics

```
#include <iostream>
#include <memory>
#include <vector>
#include <string>

class Resource {
```

```
    std::string name;
public:
    explicit Resource(std::string n) : name(std::move(n)) {
        std::cout << "Resource " << name << " created\n";
    }
    ~Resource() {
        std::cout << "Resource " << name << " destroyed\n";
    }
    void use() const {
        std::cout << "Using resource " << name << '\n';
    }
};

void smartPointerMoveExample() {
    // unique_ptr is move-only
    std::unique_ptr<Resource> ptr1 = std::make_unique<Resource>("A");
    std::unique_ptr<Resource> ptr2 = std::move(ptr1);  // Move ownership
    // ptr1 is now nullptr

    if (ptr1) {
        ptr1->use();  // Not executed
    }
    ptr2->use();  // OK

    // shared_ptr supports both copy and move
    std::shared_ptr<Resource> sptr1 = std::make_shared<Resource>("B");
    std::shared_ptr<Resource> sptr2 = sptr1;            // Copy (increments ref
    ↪  count)
    std::shared_ptr<Resource> sptr3 = std::move(sptr1); // Move (no ref count
    ↪  change)

    std::cout << "sptr2 use count: " << sptr2.use_count() << '\n';
    std::cout << "sptr3 use count: " << sptr3.use_count() << '\n';

    // Vector of unique_ptr with move semantics
    std::vector<std::unique_ptr<Resource>> vec;
    vec.push_back(std::make_unique<Resource>("C"));
    vec.push_back(std::make_unique<Resource>("D"));

    // Transfer ownership out of vector
    auto moved = std::move(vec[0]);
    moved->use();
```

```
}
```

7.3.4 Perfect Forwarding with Move Semantics

Perfect forwarding allows preserving the value category (lvalue/rvalue) of arguments when passing them through functions.

```
#include <iostream>
#include <utility>
#include <memory>

template<typename T>
void process(T&& arg) {
    std::cout << "Processing: ";
    // Perfect forwarding preserves value category
    handle(std::forward<T>(arg));
}

void handle(int& x) {
    std::cout << "lvalue: " << x << '\n';
}

void handle(int&& x) {
    std::cout << "rvalue: " << x << '\n';
}

// Factory function with perfect forwarding
template<typename T, typename... Args>
std::unique_ptr<T> makeUnique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}

class Widget {
    std::string name;
public:
    explicit Widget(std::string n) : name(std::move(n)) {
        std::cout << "Widget created: " << name << '\n';
    }
    Widget(const Widget&) = delete;
    Widget(Widget&&) = default;
};
```

```
void perfectForwardingExample() {
    int x = 42;
    process(x);           // lvalue
    process(100);         // rvalue

    // Perfect forwarding in factory
    auto w1 = makeUnique<Widget>("Hello");
    std::string str = "World";
    auto w2 = makeUnique<Widget>(str);  // lvalue
    auto w3 = makeUnique<Widget>("!");  // rvalue
}
```

7.3.5 Move Semantics in STL Containers

STL containers are designed to leverage move semantics for efficient operations.

```
#include <iostream>
#include <vector>
#include <string>
#include <chrono>

class HeavyObject {
    std::vector<int> data;
    std::string name;
public:
    HeavyObject(std::string n, size_t size)
        : name(std::move(n)), data(size, 42) {
        std::cout << "Constructed: " << name << '\n';
    }

    HeavyObject(const HeavyObject& other)
        : data(other.data), name(other.name) {
        std::cout << "Copied: " << name << '\n';
    }

    HeavyObject(HeavyObject&& other) noexcept
        : data(std::move(other.data)), name(std::move(other.name)) {
        std::cout << "Moved: " << name << '\n';
    }

    ~HeavyObject() {
        std::cout << "Destroyed: " << name << '\n';
```

```
    }
};

void stlMoveSemanticsExample() {
    std::vector<HeavyObject> vec;

    // Emplace constructs in place (no copy or move)
    vec.emplace_back("A", 1000);

    // push_back with move semantics
    HeavyObject obj("B", 1000);
    vec.push_back(std::move(obj));  // Move into vector

    // Vector reallocation uses move semantics (if noexcept)
    vec.reserve(10);  // May move existing elements

    // Transfer ownership between containers
    std::vector<HeavyObject> vec2 = std::move(vec);
    std::cout << "vec size after move: " << vec.size() << '\n';
    std::cout << "vec2 size: " << vec2.size() << '\n';
}

// Performance comparison: copy vs move
void performanceComparison() {
    const size_t count = 100000;

    // Copy version
    auto start = std::chrono::high_resolution_clock::now();
    std::vector<std::string> copyVec;
    for (size_t i = 0; i < count; ++i) {
        std::string s = "Long string that requires allocation " +
        ↪  std::to_string(i);
        copyVec.push_back(s);  // Copy
    }
    auto copyTime = std::chrono::high_resolution_clock::now() - start;

    // Move version
    start = std::chrono::high_resolution_clock::now();
    std::vector<std::string> moveVec;
    for (size_t i = 0; i < count; ++i) {
        std::string s = "Long string that requires allocation " +
        ↪  std::to_string(i);
```

```
        moveVec.push_back(std::move(s));  // Move (much faster)
    }
    auto moveTime = std::chrono::high_resolution_clock::now() - start;

    std::cout << "Copy time: " << copyTime.count() << " ns\n";
    std::cout << "Move time: " << moveTime.count() << " ns\n";
    std::cout << "Move is " << (copyTime / moveTime) << "x faster\n";
}
```

7.3.6 Move-Only Types

Some types are designed to be move-only (cannot be copied). This is common for types that manage exclusive ownership of resources.

```
#include <iostream>
#include <memory>
#include <utility>

class MoveOnly {
    std::unique_ptr<int> ptr;

public:
    MoveOnly() : ptr(std::make_unique<int>(0)) {}
    explicit MoveOnly(int value) : ptr(std::make_unique<int>(value)) {}

    // Move constructor
    MoveOnly(MoveOnly&& other) noexcept
        : ptr(std::move(other.ptr)) {
        std::cout << "Move constructed\n";
    }

    // Move assignment
    MoveOnly& operator=(MoveOnly&& other) noexcept {
        if (this != &other) {
            ptr = std::move(other.ptr);
            std::cout << "Move assigned\n";
        }
        return *this;
    }

    // Copy operations are deleted
    MoveOnly(const MoveOnly&) = delete;
```

```
    MoveOnly& operator=(const MoveOnly&) = delete;

    int value() const { return ptr ? *ptr : 0; }
    bool valid() const { return ptr != nullptr; }
};

MoveOnly createMoveOnly(int value) {
    return MoveOnly(value);  // RVO ensures no move/copy
}

void moveOnlyExample() {
    MoveOnly m1(42);
    MoveOnly m2 = std::move(m1);  // Move constructor
    // m1 is now in valid but unspecified state

    MoveOnly m3;
    m3 = std::move(m2);  // Move assignment

    std::cout << "m1 valid: " << m1.valid() << '\n';
    std::cout << "m2 valid: " << m2.valid() << '\n';
    std::cout << "m3 value: " << m3.value() << '\n';

    auto m4 = createMoveOnly(100);  // RVO: no move needed
    std::cout << "m4 value: " << m4.value() << '\n';
}
```

7.3.7 The `noexcept` Specification

Marking move operations as `noexcept` is crucial for performance, as many STL containers use this information to optimize.

Important

Why `noexcept` Matters:

- `std::vector` uses move semantics during reallocation only if the move constructor is `noexcept`. Otherwise, it falls back to copying.
- `noexcept` enables compiler optimizations.
- It's part of the contract: move operations should not throw exceptions.

Example: Impact of `noexcept` on `std::vector`

```
#include <iostream>
#include <vector>
#include <type_traits>

class ThrowOnMove {
    std::string name;
public:
    explicit ThrowOnMove(std::string n) : name(std::move(n)) {}

    // Move constructor that might throw
    ThrowOnMove(ThrowOnMove&& other) {
        throw std::runtime_error("Move threw!");
    }

    // Copy constructor
    ThrowOnMove(const ThrowOnMove& other) : name(other.name) {}

    std::string getName() const { return name; }
};

class NoexceptMove {
    std::string name;
public:
    explicit NoexceptMove(std::string n) : name(std::move(n)) {}

    // noexcept move constructor
    NoexceptMove(NoexceptMove&& other) noexcept : name(std::move(other.name)) {}

    std::string getName() const { return name; }
};

void noexceptExample() {
    std::cout << "ThrowOnMove is noexcept move? "
              << std::is_nothrow_move_constructible_v<ThrowOnMove> << '\n';
    std::cout << "NoexceptMove is noexcept move? "
              << std::is_nothrow_move_constructible_v<NoexceptMove> << '\n';

    std::vector<NoexceptMove> vec;
    for (int i = 0; i < 10; ++i) {
        vec.emplace_back("Item " + std::to_string(i));
```

```
    }
    // Vector reallocation uses move (fast)

    std::cout << "Vector size: " << vec.size() << '\n';
}
```

7.3.8 C++23 Enhancements: `std::move_only_function`

C++23 introduces `std::move_only_function`, a move-only version of `std::function`.

```
#include <functional>
#include <iostream>
#include <memory>

void moveOnlyFunctionExample() {
    // C++23: move_only_function
    std::move_only_function<void()> func = [] {
        std::cout << "Hello from move_only_function\n";
    };

    // Move-only, cannot be copied
    auto func2 = std::move(func);
    func2();

    // Can store move-only types
    auto moveOnlyLambda = [ptr = std::make_unique<int>(42)] {
        std::cout << "Value: " << *ptr << '\n';
    };

    std::move_only_function<void()> moveFunc = std::move(moveOnlyLambda);
    moveFunc();
}
```

7.3.9 Common Pitfalls

1. **Using Moved-from Objects**:

   ```
   std::string s = "hello";
   std::string t = std::move(s);
   std::cout << s << '\n';  // Valid but unspecified - don't rely on content!
   ```

   Solution: After move, only assign or destroy the source object.

2. **Missing `noexcept` on Move Operations**:

```
// Without noexcept, vector may copy instead of move
MyClass(MyClass&& other) { /* ... */ }  // Missing noexcept
```

   Solution: Mark move operations as `noexcept` when possible.

3. **Self-Move Assignment**:

```
MyClass obj;
obj = std::move(obj);  // Self-move assignment
```

   Solution: Check for self-assignment in move assignment operator.

4. **Not Resetting Source in Move**:

```
// Bad: source still owns resource
MyClass(MyClass&& other) : data(other.data) {
    // Missing: other.data = nullptr;
}
```

   Solution: Always leave the source in a valid state (e.g., set pointers to null).

5. **Throwing Exceptions from Move Operations**:

```
// Bad: move should not throw
MyClass(MyClass&& other) { throw std::runtime_error(""); }
```

   Solution: Move operations should be `noexcept` and not throw.

7.3.10 Best Practices

1. **Follow the Rule of Five**: Define all five special member functions when managing resources.
2. **Mark Move Operations `noexcept`**: Enables optimizations in STL containers.
3. **Use `std::move` for Transferring Ownership**: Indicates intent clearly.
4. **Prefer Smart Pointers Over Raw Pointers**: They handle move semantics automatically.
5. **Return Local Objects by Value**: RVO and move semantics make this efficient.
6. **Use Perfect Forwarding with `std::forward`**: Preserves value categories.
7. **Don't Use Moved-from Objects**: Except for assignment or destruction.

7.3.11 Summary

- Move semantics enables efficient transfer of resources using move constructors and move assignment operators.
- Pointers are central to move semantics, allowing ownership transfer without copying.
- The Rule of Five ensures proper resource management for classes that manage resources.
- Smart pointers (`unique_ptr`, `shared_ptr`) support move semantics natively.
- Perfect forwarding preserves value categories using `std::forward`.
- STL containers leverage move semantics for efficient operations.
- `noexcept` on move operations enables optimizations.
- C++23 adds `std::move_only_function` for move-only callables.
- Best practices: follow Rule of Five, use `noexcept`, prefer smart pointers, and avoid using moved-from objects.

Exercise 22. ***Practice Exercises:***

1. *Implement a move-only `FileHandle` class that manages a FILE* using move semantics.*
2. *Create a class that follows the Rule of Five and manages a dynamically allocated array of integers.*
3. *Compare the performance of copying vs moving a large `std::vector<std::string>`.*
4. *Implement a factory function that uses perfect forwarding to construct objects.*
5. *Create a move-only `Buffer` class and demonstrate its use with `std::vector`.*
6. *Write a function that accepts both lvalues and rvalues and forwards them to another function.*
7. *Implement a custom `optional<T>` class with move semantics support.*

Chapter 8

Best Practices and Debugging

8.1 Best Practices with Pointers

Pointers are one of the most powerful and versatile features of C++, enabling direct memory manipulation, efficient resource management, and advanced programming techniques. However, their misuse can lead to severe issues such as memory leaks, dangling pointers, and invalid memory access, which can cause crashes, undefined behavior, and security vulnerabilities. This section provides an in-depth exploration of best practices for working with pointers, focusing on avoiding common pitfalls and writing safe, efficient, and maintainable code.

Important

Golden Rule of Modern C++: If you find yourself writing `new` or `delete`, stop and consider using smart pointers or containers instead.

8.1.1 Avoiding Common Pitfalls

1. **Memory Leaks**

   A **memory leak** occurs when dynamically allocated memory is not properly deallocated, causing the program to gradually consume more memory until it exhausts system resources. Memory leaks are particularly problematic in long-running applications, such as servers or daemons, where even small leaks can accumulate over time.

Important

Causes of Memory Leaks:

- Forgetting to call `delete` or `delete[]` after allocation.
- Losing pointer references (overwriting the pointer without deallocation).
- Exception handling issues (exception thrown before deallocation).
- Circular references with `std::shared_ptr`.

Example of a Memory Leak:

```
void memoryLeakExample() {
    int* ptr = new int(10);  // Allocate memory
    // Forget to delete ptr - memory leak!
}
```

How to Avoid Memory Leaks:

(a) **Use Smart Pointers**:

```
void safeExample() {
    auto ptr = std::make_unique<int>(10);  // Automatically deallocated
}
```

(b) **Follow RAII (Resource Acquisition Is Initialization)**:

```
class Resource {
    std::unique_ptr<int[]> data;
public:
    explicit Resource(size_t size) : data(std::make_unique<int[]>(size))
    ↪ {}
    // No destructor needed - data automatically cleaned up
};
```

(c) **Use Standard Containers**:

```
std::vector<int> vec = {1, 2, 3};  // Automatic memory management
```

(d) **Avoid Circular References with `shared_ptr`**:

```
struct Node {
    std::shared_ptr<Node> next;
    std::weak_ptr<Node> prev;  // Use weak_ptr to break cycles
};
```

2. **Dangling Pointers**

   A **dangling pointer** is a pointer that points to memory that has already been deallocated. Accessing or modifying such memory leads to **undefined behavior**.

   Important

   Causes of Dangling Pointers:

   - Deleting a pointer and then using it.
   - Returning pointers to local variables.
   - Multiple pointers to the same memory with one deleting it.
   - Using `std::shared_ptr` with circular references.

   Example of a Dangling Pointer:

```
int* danglingPointerExample() {
    int x = 10;
    return &x;  // Pointer to local variable - dangling after return!
}

void useAfterDelete() {
    int* ptr = new int(42);
    delete ptr;
    *ptr = 100;  // Dangling pointer - undefined behavior!
}
```

   How to Avoid Dangling Pointers:

   (a) **Set Pointers to `nullptr` After Deletion**:

```
delete ptr;
ptr = nullptr;  // Makes accidental dereference detectable
```

   (b) **Use Smart Pointers**:

```
auto ptr = std::make_unique<int>(42);  // Never manually deleted
```

   (c) **Avoid Returning Pointers to Local Variables**:

```
std::unique_ptr<int> safeReturn() {
    return std::make_unique<int>(42);  // Ownership transferred
}
```

(d) **Use `std::optional` for Optional Values**:

```
std::optional<int> findValue(const std::vector<int>& vec, int target) {
    for (int v : vec) {
        if (v == target) return v;
    }
    return std::nullopt;  // Clear alternative to nullptr
}
```

3. **Invalid Memory Access**

Invalid memory access occurs when a program attempts to read or write memory it does not own, leading to crashes, data corruption, or security vulnerabilities.

Important

Causes of Invalid Memory Access:

- Dereferencing null or uninitialized pointers.
- Out-of-bounds array access.
- Accessing memory after deallocation (use-after-free).
- Buffer overflows.

Example of Invalid Memory Access:

```
void invalidAccessExample() {
    int* ptr = nullptr;
    *ptr = 10;  // Dereferencing null pointer - crash!

    int arr[5] = {1, 2, 3, 4, 5};
    arr[10] = 42;  // Out-of-bounds access - undefined behavior!
}
```

How to Avoid Invalid Memory Access:

(a) **Always Initialize Pointers**:

```
int* ptr = nullptr;  // Initialize to null
if (ptr) {
    *ptr = 10;  // Safe check
}
```

(b) **Use `std::span` for Array Views**:

```
#include <span>
void process(std::span<int> data) {
    for (int& val : data) {  // Bounds-safe iteration
        val *= 2;
    }
}
```

(c) **Use `std::array` or `std::vector`**:

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};
arr.at(10);  // Throws std::out_of_range in debug mode
```

(d) **Enable Compiler Sanitizers**:

```
g++ -fsanitize=address -g program.cpp -o program
./program  # Detects out-of-bounds and use-after-free
```

8.1.2 Writing Safe and Efficient Pointer Code

To write safe and efficient pointer code, follow these best practices:

1. **Prefer Smart Pointers Over Raw Pointers**

   Smart pointers express ownership clearly and automate resource management.

   ```
   // Bad: raw pointer ownership
   int* rawPtr = new int(42);
   delete rawPtr;

   // Good: smart pointer ownership
   auto smartPtr = std::make_unique<int>(42);
   ```

   Choice Guide:

   - `std::unique_ptr`: Exclusive ownership, zero overhead.
   - `std::shared_ptr`: Shared ownership, reference counting.
   - `std::weak_ptr`: Non-owning observation, breaks cycles.

2. **Use RAII for All Resources**

   RAII (Resource Acquisition Is Initialization) ensures resources are released even when exceptions occur.

```
class FileHandle {
    std::FILE* file;
public:
    FileHandle(const char* name, const char* mode)
        : file(std::fopen(name, mode)) {
        if (!file) throw std::runtime_error("Cannot open file");
    }

    ~FileHandle() {
        if (file) std::fclose(file);
    }

    // Move-only (or copy with deep copy)
    FileHandle(FileHandle&& other) noexcept
        : file(std::exchange(other.file, nullptr)) {}

    FileHandle& operator=(FileHandle&& other) noexcept {
        if (this != &other) {
            if (file) std::fclose(file);
            file = std::exchange(other.file, nullptr);
        }
        return *this;
    }

    // Non-copyable
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    std::FILE* get() const { return file; }
};
```

3. **Use `std::optional` Instead of Nullable Pointers**

   When a value may or may not exist, use `std::optional` for clarity and safety.

```
#include <optional>

// Bad: raw pointer with null meaning "not found"
int* find(std::vector<int>& vec, int target) {
    for (int& v : vec) {
        if (v == target) return &v;
    }
    return nullptr;
```

```
}

// Good: optional clearly indicates optional return
std::optional<int> find(const std::vector<int>& vec, int target) {
    for (int v : vec) {
        if (v == target) return v;
    }
    return std::nullopt;
}

// Usage
if (auto result = find(vec, 42)) {
    std::cout << "Found: " << *result << '\n';
}
```

4. **Use `std::span` for Array Parameters**

   Replace raw pointer + size parameters with `std::span` for safer array handling.

```
#include <span>

// Bad: raw pointer + size
void process(int* data, size_t size) {
    for (size_t i = 0; i < size; ++i) {
        data[i] *= 2;  // No bounds checking
    }
}

// Good: std::span with bounds awareness
void process(std::span<int> data) {
    for (int& val : data) {
        val *= 2;  // Bounds-safe iteration
    }
}

// Usage
std::vector<int> vec = {1, 2, 3, 4, 5};
process(vec);  // Works with vector
int arr[] = {1, 2, 3, 4, 5};
process(arr);  // Works with array
process(std::span<int>(arr + 1, 3));  // Subrange {2, 3, 4}
```

5. **Use `std::observer_ptr` for Non-Owning Pointers (C++26 Preview)**

The proposed C++26 standard introduces `std::observer_ptr` to clearly express non-owning pointer semantics.

```
// Proposed C++26
#include <observer_ptr>

void process(std::observer_ptr<Widget> widget) {
    if (widget) {  // Clear null check
        widget->doWork();  // Non-owning access
    }
    // observer_ptr never deletes
}

void example() {
    Widget w;
    process(std::observer_ptr(&w));  // Clear: non-owning
}
```

6. **Use `std::variant` for Type-Safe Unions**

   Replace `void*` with `std::variant` for type-safe storage of different pointer types.

```
#include <variant>
#include <iostream>

using PtrVariant = std::variant<int*, double*, std::string*>;

void processPtr(PtrVariant ptr) {
    std::visit([](auto* p) {
        if (p) {
            using T = std::decay_t<decltype(*p)>;
            std::cout << "Value: " << *p << '\n';
        }
    }, ptr);
}

void variantExample() {
    int x = 42;
    double y = 3.14;

    processPtr(&x);   // Works with int*
    processPtr(&y);   // Works with double*
    processPtr(nullptr);  // Works with nullptr
```

```
    }
```

8.1.3 Example: Writing Safe and Efficient Pointer Code

The following example demonstrates safe pointer usage combining smart pointers, RAII, and modern C++ features.

```
#include <iostream>
#include <memory>
#include <vector>
#include <span>
#include <optional>

class ResourceManager {
private:
    std::vector<std::unique_ptr<int[]>> resources;

public:
    // RAII: resources automatically cleaned up
    ~ResourceManager() = default;  // unique_ptr handles cleanup

    void addResource(size_t size) {
        resources.push_back(std::make_unique<int[]>(size));
        auto& data = resources.back();

        // Initialize with RAII
        for (size_t i = 0; i < size; ++i) {
            data[i] = static_cast<int>(i);
        }
    }

    // Safe view using std::span
    std::optional<std::span<int>> getResource(size_t index) {
        if (index < resources.size()) {
            size_t size = getResourceSize(index);
            return std::span<int>(resources[index].get(), size);
        }
        return std::nullopt;
    }

    size_t getResourceSize(size_t index) const {
        // In a real implementation, we'd store sizes separately
```

```
        // For demo, we assume a fixed pattern
        return 10;
    }

    // Process all resources safely
    void processAll() {
        for (size_t i = 0; i < resources.size(); ++i) {
            if (auto span = getResource(i)) {
                for (int& val : *span) {
                    val *= 2;  // Safe mutation
                }
            }
        }
    }

    size_t resourceCount() const { return resources.size(); }
};

void safePointerExample() {
    ResourceManager manager;
    manager.addResource(5);
    manager.addResource(10);

    if (auto span = manager.getResource(0)) {
        std::cout << "Resource 0: ";
        for (int val : *span) std::cout << val << ' ';
        std::cout << '\n';
    }

    manager.processAll();

    if (auto span = manager.getResource(0)) {
        std::cout << "After processing: ";
        for (int val : *span) std::cout << val << ' ';
        std::cout << '\n';
    }

    // No manual cleanup needed - RAII handles it
}
```

8.1.4 Debugging Pointer-Related Issues

Debugging pointer-related issues can be challenging, but several tools and techniques can help:

1. **Address Sanitizer (ASan)**

   Compile with `-fsanitize=address` to detect memory errors at runtime.

   ```
   # Compilation
   g++ -fsanitize=address -g program.cpp -o program

   # Run
   ./program  # Detects use-after-free, buffer overflows, memory leaks
   ```

   Detected Issues:

   - Use-after-free
   - Heap buffer overflow
   - Stack buffer overflow
   - Memory leaks (with LeakSanitizer)
   - Double-free

2. **Undefined Behavior Sanitizer (UBSan)**

   Detects undefined behavior at runtime.

   ```
   g++ -fsanitize=undefined -g program.cpp -o program
   ./program  # Detects integer overflow, null pointer dereference, etc.
   ```

3. **Valgrind (Linux)**

   Comprehensive memory debugging tool.

   ```
   valgrind --leak-check=full --show-leak-kinds=all ./program
   ```

4. **Static Analyzers**

   Catch issues at compile time.

   ```
   # clang-tidy
   clang-tidy program.cpp --checks=*
   ```

```
# cppcheck
cppcheck --enable=all program.cpp

# MSVC /analyze
cl /analyze program.cpp
```

5. **Debugging Techniques**

(a) **Use `std::stacktrace` (C++23)**:

```
#include <stacktrace>
#include <iostream>

void dangerous() {
    int* ptr = nullptr;
    std::cout << std::stacktrace::current() << '\n';  // Print call stack
    *ptr = 42;  // Crash with stack trace
}
```

(b) **Logging and Assertions**:

```
#include <cassert>
#include <iostream>

void safeUse(int* ptr) {
    assert(ptr != nullptr && "Pointer is null!");
    *ptr = 42;
}
```

(c) **Use `std::source_location` (C++20)**:

```
#include <source_location>
#include <iostream>

void log(const std::source_location& loc =
↪  std::source_location::current()) {
    std::cout << loc.file_name() << ':' << loc.line() << '\n';
}
```

8.1.5 Modern C++ Safety Features Summary

Feature	Purpose	Standard
`std::unique_ptr`	Exclusive ownership	C++11
`std::shared_ptr`	Shared ownership	C++11
`std::weak_ptr`	Non-owning observation	C++11
`std::optional`	Optional values	C++17
`std::variant`	Type-safe unions	C++17
`std::span`	Non-owning array views	C++20
`std::source_location`	Debug information	C++20
`std::stacktrace`	Call stack capture	C++23
`std::observer_ptr`	Non-owning pointers (proposed)	C++26

8.1.6 Common Pitfalls Summary

1. **Memory Leaks**: Use smart pointers and RAII.
2. **Dangling Pointers**: Use smart pointers, set to nullptr after delete.
3. **Null Dereference**: Always check pointers before use.
4. **Buffer Overflow**: Use containers, `std::span`, bounds checking.
5. **Use-After-Free**: Smart pointers prevent this.
6. **Double Delete**: Smart pointers prevent this.
7. **Circular References**: Use `std::weak_ptr` to break cycles.
8. **Object Slicing**: Use pointers or references for polymorphism.

8.1.7 Summary

- Memory leaks, dangling pointers, and invalid memory access are the most common pointer-related bugs.
- Smart pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`) automate memory management and prevent leaks.

- RAII ensures resources are properly released, even in the presence of exceptions.
- Modern vocabulary types (`optional`, `span`, `variant`, `observer_ptr`) provide safer alternatives to raw pointers.
- Compiler sanitizers (ASan, UBSan) and tools (Valgrind, static analyzers) help detect pointer-related issues.
- Best practices: prefer smart pointers, use RAII, validate pointers, leverage standard library containers.

Exercise 23. ***Practice Exercises:***

1. *Identify and fix memory leaks, dangling pointers, and invalid memory access in a given piece of code.*
2. *Refactor raw pointer code to use smart pointers and RAII.*
3. *Implement a thread-safe resource manager using* `std::shared_ptr` *and* `std::weak_ptr`*.*
4. *Use* `std::span` *to replace raw pointer + size parameters in existing code.*
5. *Enable AddressSanitizer and UndefinedBehaviorSanitizer on a test program and interpret the output.*
6. *Implement a custom RAII wrapper for a POSIX file descriptor.*
7. *Research* `std::observer_ptr` *(C++26 draft) and write a comparison with raw pointers for non-owning semantics.*

8.2 Debugging Pointer Issues

Debugging pointer-related issues is one of the most challenging aspects of C++ programming. Pointers, while powerful, are prone to errors such as memory leaks, dangling pointers, and invalid memory access. These issues can lead to crashes, undefined behavior, and security vulnerabilities. Fortunately, there are powerful tools and techniques available to help identify and resolve these problems. This section provides an in-depth exploration of debugging techniques, focusing on tools like **Valgrind**, **AddressSanitizer**, **UndefinedBehaviorSanitizer**, and modern C++ debugging features, including practical examples of debugging memory leaks, invalid memory access, and other pointer-related issues.

Important

Core Principle: Use tools early and often. Sanitizers during development catch bugs before they reach production.

8.2.1 Tools for Debugging Pointer-Related Problems

1. **Valgrind**

   Valgrind is a widely used open-source tool for detecting memory leaks, invalid memory access, and other memory-related errors. It runs your program in a virtual machine, tracking all memory allocations and deallocations.

   Important

   Key Features:

   - **Memcheck**: Detects memory leaks, invalid reads/writes, use of uninitialized memory, and double frees.
   - **Helgrind**: Detects thread race conditions.
   - **Cachegrind**: Analyzes cache performance.
   - **Callgrind**: Profiles call graphs.

   Example Command:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes
↪  ./program
```

Example Output:

```
==12345== HEAP SUMMARY:
==12345==     in use at exit: 72 bytes in 3 blocks
==12345==   total heap usage: 5 allocs, 2 frees, 1,024 bytes allocated
==12345==
==12345== 72 bytes in 3 blocks are definitely lost in loss record 1 of 1
==12345==    at 0x4C2BBAF: operator new(unsigned long)
↪  (vg_replace_malloc.c:334)
==12345==    by 0x4005E6: memoryLeakExample() (example.cpp:5)
==12345==    by 0x4005F6: main (example.cpp:10)
==12345==
==12345== LEAK SUMMARY:
==12345==    definitely lost: 72 bytes in 3 blocks
==12345==    indirectly lost: 0 bytes in 0 blocks
==12345==      possibly lost: 0 bytes in 0 blocks
==12345==    still reachable: 0 bytes in 0 blocks
```

2. **AddressSanitizer (ASan)**

AddressSanitizer is a memory error detector built into modern compilers (GCC, Clang). It detects use-after-free, buffer overflows, and memory leaks at runtime with low overhead.

Important

Key Features:

- Use-after-free detection
- Heap buffer overflow detection
- Stack buffer overflow detection
- Global buffer overflow detection
- Memory leak detection (with LeakSanitizer)
- Double-free detection

Example Compilation:

```
# GCC/Clang
g++ -fsanitize=address -g -O1 program.cpp -o program
clang++ -fsanitize=address -g -O1 program.cpp -o program

# With memory leak detection (included by default)
g++ -fsanitize=address -fsanitize=leak -g program.cpp -o program
```

Example Output:

```
=================================================================
==12345==ERROR: AddressSanitizer: heap-use-after-free on address
↪  0x604000000010
READ of size 4 at 0x604000000010 thread T0
    #0 0x4006a6 in useAfterFreeExample() /path/to/program.cpp:12
    #1 0x4006b6 in main /path/to/program.cpp:18

0x604000000010 is located 0 bytes inside of 4-byte region
↪  [0x604000000010,0x604000000014)
freed by thread T0 here:
    #0 0x4f5b8d in operator delete(void*) /path/to/asan
    #1 0x40069a in useAfterFreeExample() /path/to/program.cpp:10

previously allocated by thread T0 here:
    #0 0x4f5b8d in operator new(unsigned long) /path/to/asan
    #1 0x40068e in useAfterFreeExample() /path/to/program.cpp:8
```

3. **UndefinedBehaviorSanitizer (UBSan)**

   UndefinedBehaviorSanitizer detects undefined behavior at runtime, including null pointer dereference, signed integer overflow, and alignment violations.

   Example Compilation:

   ```
   g++ -fsanitize=undefined -g program.cpp -o program
   ```

   Example Output:

   ```
   /program.cpp:15:10: runtime error: null pointer dereference
   SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior /program.cpp:15:10
   ```

4. **Static Analyzers**

   Static analyzers detect potential issues at compile time without executing the code.

Important

Popular Static Analyzers:

- **clang-tidy**: Clang-based static analyzer with extensive checks
- **cppcheck**: Open-source static analyzer for C/C++
- **PVS-Studio**: Commercial static analyzer
- **MSVC /analyze**: Microsoft's built-in analyzer

Example Commands:

```
# clang-tidy
clang-tidy program.cpp --checks=* -- -std=c++23

# cppcheck
cppcheck --enable=all --inconclusive --std=c++23 program.cpp

# MSVC
cl /analyze /W4 program.cpp
```

8.2.2 Example: Debugging a Memory Leak

Let's walk through a complete example of debugging a memory leak using both Valgrind and AddressSanitizer.

1. **Problem Code**

```
#include <iostream>
#include <memory>

void memoryLeakExample() {
    int* ptr = new int(42);  // Allocate memory
    // Forget to delete ptr - memory leak!
    std::cout << "Value: " << *ptr << '\n';
}

int main() {
    for (int i = 0; i < 100; ++i) {
        memoryLeakExample();
```

```
    }
    std::cout << "Memory leak example completed\n";
    return 0;
}
```

2. **Debugging with Valgrind**

```
g++ -g -O0 memory_leak.cpp -o memory_leak
valgrind --leak-check=full --show-leak-kinds=all ./memory_leak
```

Valgrind Output:

```
==12345== HEAP SUMMARY:
==12345==     in use at exit: 400 bytes in 100 blocks
==12345==   total heap usage: 101 allocs, 1 frees, 1,024 bytes allocated
==12345==
==12345== 400 bytes in 100 blocks are definitely lost in loss record 1 of 1
==12345==    at 0x4C2BBAF: operator new(unsigned long)
↪  (vg_replace_malloc.c:334)
==12345==    by 0x4006A6: memoryLeakExample() (memory_leak.cpp:5)
==12345==    by 0x4006E6: main (memory_leak.cpp:11)
==12345==
==12345== LEAK SUMMARY:
==12345==    definitely lost: 400 bytes in 100 blocks
==12345==    indirectly lost: 0 bytes in 0 blocks
==12345==      possibly lost: 0 bytes in 0 blocks
==12345==    still reachable: 0 bytes in 0 blocks
```

3. **Debugging with AddressSanitizer**

```
g++ -fsanitize=address -g -O0 memory_leak.cpp -o memory_leak_asan
./memory_leak_asan
```

AddressSanitizer Output:

```
=================================================================
==12345==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 400 byte(s) in 100 object(s) allocated from:
    #0 0x4f5b8d in operator new(unsigned long)
    ↪  (/path/to/memory_leak_asan+0x4f5b8d)
    #1 0x4006a6 in memoryLeakExample() memory_leak.cpp:5
    #2 0x4006e6 in main memory_leak.cpp:11
```

```
    #3 0x7f8c5b8b8b96 in __libc_start_main
    ↪  (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)

SUMMARY: AddressSanitizer: 400 byte(s) leaked in 100 allocation(s).
```

4. **Fixing the Memory Leak**

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

void memoryLeakExample() {
    // Use smart pointer for automatic cleanup
    auto ptr = std::make_unique<int>(42);
    std::cout << "Value: " << *ptr << '\n';
}  // ptr automatically deallocated

int main() {
    for (int i = 0; i < 100; ++i) {
        memoryLeakExample();
    }
    std::cout << "Memory leak fixed\n";
    return 0;
}
```

8.2.3 Example: Debugging Invalid Memory Access

1. **Problem Code (Use-After-Free)**

```cpp
#include <iostream>

void useAfterFreeExample() {
    int* ptr = new int(42);
    delete ptr;
    *ptr = 100;  // Use after free! Undefined behavior
    std::cout << "Value: " << *ptr << '\n';
}

int main() {
    useAfterFreeExample();
    return 0;
}
```

2. **Debugging with AddressSanitizer**

```
g++ -fsanitize=address -g -O0 use_after_free.cpp -o use_after_free
./use_after_free
```

AddressSanitizer Output:

```
=================================================================
==12345==ERROR: AddressSanitizer: heap-use-after-free on address
↪  0x604000000010
READ of size 4 at 0x604000000010 thread T0
    #0 0x4006a6 in useAfterFreeExample() use_after_free.cpp:8
    #1 0x4006b6 in main use_after_free.cpp:13

0x604000000010 is located 0 bytes inside of 4-byte region
↪  [0x604000000010,0x604000000014)
freed by thread T0 here:
    #0 0x4f5b8d in operator delete(void*) /path/to/asan
    #1 0x40069a in useAfterFreeExample() use_after_free.cpp:6

previously allocated by thread T0 here:
    #0 0x4f5b8d in operator new(unsigned long) /path/to/asan
    #1 0x40068e in useAfterFreeExample() use_after_free.cpp:5

SUMMARY: AddressSanitizer: heap-use-after-free /path/to/use_after_free.cpp:8
```

3. **Problem Code (Buffer Overflow)**

```
#include <iostream>

void bufferOverflowExample() {
    int arr[5] = {1, 2, 3, 4, 5};
    arr[10] = 42;  // Buffer overflow! Out of bounds
    std::cout << "Value: " << arr[10] << '\n';
}

int main() {
    bufferOverflowExample();
    return 0;
}
```

AddressSanitizer Output:

```
=================================================================
==12345==ERROR: AddressSanitizer: stack-buffer-overflow on address
↪  0x7ffc12345678
```

```
WRITE of size 4 at 0x7ffc12345678 thread T0
    #0 0x4006a6 in bufferOverflowExample() buffer_overflow.cpp:6
    #1 0x4006b6 in main buffer_overflow.cpp:11
```

8.2.4 Advanced Debugging Techniques

1. **Custom Debugging Macros**

   Create macros to track memory allocations during development.

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

#ifdef DEBUG_MEMORY
    #define NEW new(__FILE__, __LINE__)
    #define new NEW
#endif

struct AllocationInfo {
    const char* file;
    int line;
    size_t size;
};

std::unordered_map<void*, AllocationInfo> allocations;

void* operator new(size_t size, const char* file, int line) {
    void* ptr = malloc(size);
    allocations[ptr] = {file, line, size};
    std::cout << "Allocated " << size << " bytes at " << file << ":" << line
    ↪  << '\n';
    return ptr;
}

void operator delete(void* ptr) noexcept {
    auto it = allocations.find(ptr);
    if (it != allocations.end()) {
        std::cout << "Deallocated " << it->second.size
                  << " bytes from " << it->second.file << ":" <<
                  ↪  it->second.line << '\n';
        allocations.erase(it);
    }
```

```
    free(ptr);
}

void memoryTrackingExample() {
    int* ptr = new int(42);  // Tracked allocation
    delete ptr;              // Tracked deallocation
}

int main() {
    memoryTrackingExample();

    if (!allocations.empty()) {
        std::cout << "Memory leaks detected:\n";
        for (const auto& [ptr, info] : allocations) {
            std::cout << "  " << info.size << " bytes at " << info.file <<
            ↪ ":" << info.line << '\n';
        }
    }
    return 0;
}
```

2. **Debugging with GDB**

   GDB (GNU Debugger) is essential for debugging pointer issues.

Important

Useful GDB Commands:

- `break main`: Set breakpoint at main
- `watch *ptr`: Watch memory location for changes
- `print ptr`: Print pointer value
- `x/10x ptr`: Examine memory at ptr
- `backtrace`: Show call stack
- `info locals`: Show local variables
- `frame N`: Jump to stack frame N

Example GDB Session:

```
gdb ./program
(gdb) break main
(gdb) run
(gdb) break functionName
(gdb) continue
(gdb) watch *ptr
(gdb) continue
(gdb) print ptr
(gdb) x/10x ptr
(gdb) backtrace
(gdb) info locals
```

3. **C++23: `std::stacktrace` for Debugging**

   C++23 introduces `std::stacktrace` for programmatic stack trace capture.

```cpp
#include <stacktrace>
#include <iostream>
#include <memory>

void dangerousFunction() {
    int* ptr = nullptr;
    // Print stack trace before potential crash
    std::cout << "Stack trace:\n" << std::stacktrace::current() << '\n';
    *ptr = 42;  // Will crash, but we have stack trace
}

void safeWithStackTrace() {
    int* ptr = new int(42);
    std::cout << "Allocated at:\n" << std::stacktrace::current() << '\n';
    delete ptr;
}

int main() {
    safeWithStackTrace();
    // dangerousFunction();  // Uncomment to see crash with stack trace
    return 0;
}
```

4. **C++20: `std::source_location` for Logging**

C++20 provides `std::source_location` for capturing file, line, and function information.

```
#include <source_location>
#include <iostream>

template<typename T>
void debugPointer(T* ptr, const std::source_location& loc =
↪  std::source_location::current()) {
    std::cout << loc.file_name() << ':' << loc.line()
              << " (" << loc.function_name() << ") - ";
    if (ptr) {
        std::cout << "ptr = " << *ptr << '\n';
    } else {
        std::cout << "nullptr\n";
    }
}

void example() {
    int x = 42;
    int* ptr = &x;
    debugPointer(ptr);

    ptr = nullptr;
    debugPointer(ptr);
}
```

5. **Using Sanitizers with CMake**

   For CMake projects, enable sanitizers easily:

```
# CMakeLists.txt
if(CMAKE_CXX_COMPILER_ID MATCHES "GNU|Clang")
    target_compile_options(program PRIVATE
        -fsanitize=address
        -fsanitize=undefined
        -g -O1)
    target_link_options(program PRIVATE
        -fsanitize=address
        -fsanitize=undefined)
endif()
```

8.2.5 Common Pointer Issues and Their Symptoms

Issue	Symptoms	Detection Tools
Memory Leak	Increasing memory usage, crashes	Valgrind, ASan
Use-After-Free	Random crashes, corruption	ASan, Valgrind
Buffer Overflow	Crashes, data corruption	ASan, Valgrind
Null Dereference	Segmentation fault	UBSan, GDB
Double Free	Crashes, heap corruption	ASan, Valgrind
Uninitialized Pointer	Random behavior, crashes	Valgrind, UBSan
Dangling Pointer	Intermittent crashes	ASan, Valgrind

8.2.6 Best Practices for Debugging Pointer Issues

1. **Enable Sanitizers During Development**:

```
# Always compile with sanitizers in debug mode
g++ -fsanitize=address,undefined -g -O1 program.cpp -o program
```

2. **Use Valgrind for Comprehensive Analysis**:

```
valgrind --leak-check=full --track-origins=yes ./program
```

3. **Set Pointers to `nullptr` After Delete**:

```
delete ptr;
ptr = nullptr;  // Makes use-after-delete detectable
```

4. **Use `assert()` for Invariant Checking**:

```
#include <cassert>
void process(int* ptr) {
    assert(ptr != nullptr && "Pointer must not be null");
    *ptr = 42;
}
```

5. **Use Static Analyzers in CI/CD**:

```
# Run clang-tidy on every commit
clang-tidy program.cpp --checks=* -- -std=c++23
```

6. **Log Pointer Operations During Debugging**:

```
#define DEBUG_PTR(ptr) std::cout << __LINE__ << ": " << #ptr << " = " << ptr
↪  << '\n'
```

8.2.7 C++26 Preview: Enhanced Debugging Features

The proposed C++26 standard includes enhanced debugging support:

```
// Proposed C++26: std::debugging
#include <debugging>

void enhancedDebugExample() {
    int* ptr = new int(42);

    // Attach debug metadata to pointer
    std::debugging::annotate(ptr, "Allocated in enhancedDebugExample");

    delete ptr;

    // Detect use after free with metadata
    // *ptr = 100;  // Would trigger debugger breakpoint
}
```

8.2.8 Summary

- Valgrind provides comprehensive memory error detection with detailed reports.
- AddressSanitizer offers fast, low-overhead detection of memory errors.
- UndefinedBehaviorSanitizer detects null dereference, overflow, and other UB.
- Static analyzers catch issues at compile time.
- GDB enables interactive debugging with breakpoints, watches, and memory examination.
- C++23 adds `std::stacktrace` for programmatic stack capture.
- C++20 adds `std::source_location` for context-aware logging.
- Best practices: use sanitizers during development, set pointers to nullptr after delete, use assertions, and run static analyzers regularly.

Exercise 24. ***Practice Exercises:***

1. *Create a program with a memory leak, then use Valgrind to identify and fix it.*
2. *Write code with a use-after-free bug and debug it using AddressSanitizer.*

3. *Create a buffer overflow and use AddressSanitizer to locate the exact line.*
4. *Use GDB to set watchpoints and debug a dangling pointer issue.*
5. *Integrate clang-tidy into your build system and fix all pointer-related warnings.*
6. *Use* `std::stacktrace` *(C++23) to capture stack traces at key points in your program.*
7. *Implement a custom memory tracker using* `operator new/delete` *overrides to detect leaks.*

Conclusion

The Journey Through C++ Pointers

Throughout this book, we have embarked on a comprehensive journey through one of the most powerful and essential features of the C++ programming language: **pointers**. From the fundamental concepts of memory addresses and pointer arithmetic to the sophisticated abstractions of smart pointers and move semantics, we have explored every facet of pointer usage in modern C++. This conclusion synthesizes the key lessons learned and provides guidance for applying this knowledge in your own programming endeavors.

The Evolution of Pointers in C++

The story of pointers in C++ is one of evolution and refinement. What began as a direct inheritance from C—raw pointers with manual memory management—has evolved into a sophisticated ecosystem of smart pointers, vocabulary types, and language features that make memory management safer and more expressive than ever before.

Important

The Modern C++ Philosophy:

- **Express Ownership Clearly**: Use `std::unique_ptr` for exclusive ownership, `std::shared_ptr` for shared ownership, and raw pointers (or `std::observer_ptr` in C++26) for non-owning observation.
- **Prefer RAII**: Let constructors acquire resources and destructors release them automatically.
- **Embrace Move Semantics**: Transfer ownership efficiently without copying.
- **Use Vocabulary Types**: `std::optional`, `std::span`, and `std::variant` express intent clearly.
- **Leverage the Standard Library**: Containers like `std::vector` and `std::string` eliminate manual memory management.

Key Takeaways from Each Chapter

Chapter 1: Introduction to Pointers in C++

We established the foundation by defining pointers as variables that store memory addresses. We explored the critical role of pointers in dynamic memory allocation, efficient data manipulation, and systems programming. We distinguished between pointers and references, clarified common misconceptions, and introduced the evolution toward smart pointers.

Chapter 2: Basics of Pointers

We mastered the fundamentals: declaring and initializing pointers, understanding pointer arithmetic, and exploring the intimate relationship between pointers and arrays. We learned that array names decay to pointers, enabling efficient array traversal and manipulation.

Chapter 3: Advanced Pointers

We delved into advanced techniques: passing pointers to functions, returning pointers (with careful lifetime management), and using function pointers for callbacks. We explored pointers to structures and classes, the `this` pointer, and dynamic memory management with `new` and `delete`.

Chapter 4: Modern C++ and Smart Pointers

We embraced the modern C++ approach to memory management. `std::unique_ptr` provides exclusive ownership with zero overhead. `std::shared_ptr` enables shared ownership through reference counting. `std::weak_ptr` breaks circular references and provides non-owning observation. These smart pointers form the backbone of safe resource management in modern C++.

Chapter 5: Pointers and Object-Oriented Programming

We explored how pointers enable runtime polymorphism through base class pointers and virtual functions. We examined the vtable mechanism that makes dynamic dispatch possible, and we learned to use `dynamic_cast` for safe downcasting in inheritance hierarchies.

Chapter 6: Pointers and Data Structures

We applied pointer knowledge to build fundamental data structures: linked lists (singly and doubly), binary search trees, and graphs. We learned how pointers enable dynamic, non-contiguous data structures that can grow and shrink at runtime.

Chapter 7: Pointers and Advanced C++ Features

We integrated pointers with advanced C++ features: templates for generic programming, move semantics for efficient resource transfer, and multithreading for concurrent execution. We explored type traits, concepts, and the Rule of Five for proper resource management.

Chapter 8: Best Practices and Debugging

We consolidated best practices for safe pointer usage: prefer smart pointers, implement RAII, validate pointers before dereferencing, and use standard library containers. We learned to debug pointer issues using tools like Valgrind, AddressSanitizer, and GDB.

The Modern C++ Landscape

As we look toward the future of C++, the evolution of pointer-related features continues:

1. **C++23 Enhancements**:
    - `std::out_ptr` and `std::inout_ptr` for safe C API interoperability
    - `std::make_unique_for_overwrite` for performance-critical array allocations
    - `std::stacktrace` for debugging pointer-related crashes
    - `std::mdspan` for multidimensional array views
2. **C++26 Preview**:
    - `std::observer_ptr` for clear non-owning pointer semantics
    - `std::ptr_len` for standardized pointer-length pairs
    - Enhanced debugging and reflection capabilities

Guidelines for Continued Learning

Mastering pointers is not a destination but a continuous journey. Here are recommendations for deepening your expertise:

1. **Practice Consistently**
    - Implement data structures from scratch using smart pointers.
    - Refactor existing code to use modern pointer practices.
    - Contribute to open-source C++ projects to see real-world patterns.
2. **Study Advanced Resources**
    - Read ”Effective Modern C++” by Scott Meyers for best practices.
    - Explore ”C++ Concurrency in Action” by Anthony Williams for multithreaded pointer usage.
    - Follow CppCon and C++Now conference talks on YouTube.
3. **Use Tools Rigorously**

- Enable AddressSanitizer and UndefinedBehaviorSanitizer in development builds.
- Run Valgrind for comprehensive memory analysis.
- Use static analyzers like clang-tidy to catch issues early.

4. **Stay Current with the Standard**
   - Follow the ISO C++ committee papers (WG21) for upcoming features.
   - Experiment with new standards using Compiler Explorer (godbolt.org).
   - Subscribe to C++ newsletters and blogs.

Final Reflections

Pointers are often described as one of the most challenging aspects of C++—and indeed, they demand respect and careful attention. But they are also one of the most rewarding. The ability to directly manipulate memory, to build efficient data structures, to interface with hardware, and to achieve performance that higher-level languages cannot match—these are the powers that pointers bestow.

The evolution of C++ has not diminished the importance of pointers; rather, it has provided us with the tools to use them safely and expressively. Smart pointers express ownership, move semantics transfer resources efficiently, and vocabulary types communicate intent clearly. Modern C++ does not abandon the power of pointers—it harnesses it.

> *"C++ is a language where you can do anything—but with that power comes responsibility. Smart pointers and modern idioms help you exercise that responsibility effectively."*

As you continue your programming journey, remember the principles we have explored:

- **Ownership Matters**: Always know who owns a resource.
- **RAII is Your Friend**: Let object lifetimes manage resources.
- **Prefer Smart Pointers**: Use `unique_ptr` and `shared_ptr` for ownership.
- **Use the Standard Library**: Containers eliminate manual memory management.
- **Validate and Debug**: Check pointers before use, and use tools to catch errors.

- **Think in Terms of Memory**: Understanding how memory works makes you a better programmer.

A Final Word

Thank you for joining me on this journey through the world of C++ pointers. Whether you are a student beginning your programming career, a professional seeking to deepen your expertise, or an educator guiding others, I hope this book has provided you with the knowledge and confidence to use pointers effectively and safely.
The world of C++ continues to evolve, and the best way to stay proficient is to keep learning, keep coding, and keep exploring. I invite you to connect with the C++ community, share your knowledge, and continue the journey of mastery.

Happy coding, and may your pointers always point to valid memory!

`Ayman Alheraki`

`April 2026`

What's Next?

To continue your learning journey, explore these resources:

1. **Books**:
    - "Effective Modern C++" by Scott Meyers
    - "The C++ Programming Language" by Bjarne Stroustrup
    - "C++ Concurrency in Action" by Anthony Williams
2. **Online Resources**:
    - `https://en.cppreference.com` — Comprehensive C++ reference
    - `https://isocpp.org` — The home of Standard C++
    - `https://godbolt.org` — Compiler Explorer for experimentation
3. **Communities**:

- https://stackoverflow.com/questions/tagged/c++
- https://www.reddit.com/r/cpp/
- C++ Slack and Discord communities

4. **Conferences**:
    - CppCon (cppcon.org)
    - C++Now (cppnow.org)
    - Meeting C++ (meetingcpp.com)

Stay Connected:

Follow me on LinkedIn: https://linkedin.com/in/aymanalheraki

Visit my website: https://simplifycpp.org

Appendices

Appendix A: Key Terminology

This appendix defines essential terms related to pointers in C++ to reinforce your understanding.

- **Pointer**: A variable that stores the memory address of another variable.
- **Dereferencing**: Accessing the value stored at the memory address held by a pointer.
- **Null Pointer**: A pointer that is not assigned a valid address (`nullptr` in Modern C++).
- **Dangling Pointer**: A pointer that refers to a memory location that has been deallocated.
- **Memory Leak**: Occurs when dynamically allocated memory is not properly deallocated.
- **Smart Pointer**: A C++11 feature that automatically manages memory (`std::unique_ptr`, `std::shared_ptr`, etc.).
- **Pointer Arithmetic**: Performing arithmetic operations on pointers, such as incrementing or decrementing addresses.
- **Function Pointer**: A pointer that stores the address of a function.
- **Void Pointer**: A generic pointer that can hold addresses of any data type (`void*`).
- **Weak Pointer**: A `std::weak_ptr` that prevents circular references in smart pointer usage.

Appendix B: C++ Pointer Cheat Sheet

A quick reference guide for using pointers efficiently in C++.

Basic Pointer Syntax

```
int x = 10;
int* ptr = &x; // Pointer to x
```

Pointer Arithmetic

```
int arr[5] = {1, 2, 3, 4, 5};
int* p = arr;
p++; // Moves to the next array element
```

Smart Pointers

```
#include <memory>
std::unique_ptr<int> uptr = std::make_unique<int>(10);
std::shared_ptr<int> sptr = std::make_shared<int>(20);
```

Common Pointer Mistakes

```
int* ptr = new int(5);
delete ptr;
delete ptr; // Double delete - causes undefined behavior!
```

Appendix C: Memory Management in C++

Understanding memory management is critical when working with pointers.

Stack vs. Heap Memory

- **Stack**: Stores function calls and local variables; managed automatically.
- **Heap**: Stores dynamically allocated memory; must be managed manually.

Manual Memory Management

```
int* ptr = new int(10); // Allocates memory on heap
delete ptr; // Deallocates memory
```

Using Smart Pointers for Safety

```
std::unique_ptr<int> uPtr = std::make_unique<int>(42);
std::shared_ptr<int> sPtr = std::make_shared<int>(55);
```

Appendix D: Common Pointer Errors and Debugging Techniques

Pointers can introduce subtle bugs if not used carefully. This section covers common mistakes and debugging techniques.

Common Errors

1. **Null Pointer Dereferencing**

   ```
   int* ptr = nullptr;
   *ptr = 10; // Undefined behavior!
   ```

2. **Dangling Pointers**

   ```
   int* ptr = new int(10);
   delete ptr;
   *ptr = 20; // Accessing deleted memory!
   ```

3. **Memory Leaks**

   ```
   int* ptr = new int(100);
   // No delete operation - memory leak!
   ```

Debugging Tools

- **Valgrind** (Linux/macOS): Detects memory leaks and invalid memory access.
- **AddressSanitizer**: Available in Clang and GCC, helps detect memory issues.
- **GDB/LLDB**: Debuggers that help track pointer-related errors.

Appendix E: Practical Exercises

Strengthen your understanding with these hands-on exercises.

Exercise 1: Pointer Arithmetic

Write a program that demonstrates pointer arithmetic with an array.

```
#include <iostream>
int main() {
    int arr[] = {10, 20, 30, 40};
    int* ptr = arr;
    for (int i = 0; i < 4; i++) {
        std::cout << "Value: " << *ptr << "\n";
        ptr++;
    }
}
```

Exercise 2: Implementing a Smart Pointer

Write a basic implementation of a `unique_ptr`.

```
#include <iostream>
template <typename T>
class UniquePtr {
private:
    T* ptr;
public:
    explicit UniquePtr(T* p = nullptr) : ptr(p) {}
    ~UniquePtr() { delete ptr; }
    T& operator*() { return *ptr; }
};
int main() {
    UniquePtr<int> p(new int(42));
    std::cout << *p << "\n";
}
```

Exercise 3: Implementing a Linked List Using Pointers

Implement a simple singly linked list using raw pointers.

Appendix F: Additional Resources

Expand your knowledge with these resources.

Books

- *Effective Modern C++* by Scott Meyers
- *The C++ Programming Language* by Bjarne Stroustrup
- *C++ Primer* by Lippman, Lajoie, and Moo

Online Courses and Tutorials

- C++ Reference: https://en.cppreference.com
- Modern C++ Best Practices: cppbestpractices.com
- Online C++ Compiler: https://godbolt.org

Communities and Forums

- C++ Reddit: https://www.reddit.com/r/cpp/
- Stack Overflow: https://stackoverflow.com/questions/tagged/c%2b%2b
- C++ ISO Standard Discussion: https://isocpp.org

References

This section provides an up-to-date collection of the most recent C++ standards papers, proposals, and resources that are directly relevant to pointers, memory management, and the evolving C++ language. All documents listed here are accessible through the official ISO C++ committee (WG21) website or other authoritative sources.

C++23 and C++26 Standards Papers

The following papers have been adopted into C++23 or are under consideration for C++26. They introduce new vocabulary types, safer pointer abstractions, and improved memory management facilities.

1. **P0260R4: `std::out_ptr` and `std::inout_ptr`**
   Jonathan Wakely, 2022.
   https://wg21.link/p0260r4
   Adopted for C++23. Provides safe interoperability between smart pointers and legacy C APIs.

2. **P2273R3: `std::observer_ptr` — A Non-Owning Pointer for C++26**
   T. Veldhuizen, 2024.
   https://wg21.link/p2273r3
   Proposed for C++26. Introduces a vocabulary type for non-owning observation, clarifying pointer semantics.

3. **P2286R2: `std::ptr_len` — A Vocabulary Type for Pointer-Length Pairs**
   B. Stroustrup, 2024.
   https://wg21.link/p2286r2

Proposed for C++26. Standardises the common pattern of passing a pointer and a length.

4. **P2996R4: Compile-Time Reflection**
   M. Clow, D. Vandevoorde, et al., 2025.
   https://wg21.link/p2996r4
   Accepted for C++26. Adds compile-time reflection, enabling safer metaprogramming and automatic resource management.

5. **P2692R0: Contracts for C++**
   G. Dos Reis, J. Lakos, et al., 2024.
   https://wg21.link/p2692r0
   Accepted for C++26. Introduces design-by-contract with `contract_assert`, `pre`, and `post`.

6. **P3471R4: Bounds-Hardened Standard Library**
   A. G. G. S., 2025.
   https://wg21.link/p3471r4
   Accepted for C++26. Adds optional bounds checking to containers and views to prevent buffer overflows.

7. **P1928R0: `<simd>` — Data-Parallel Types**
   M. Kretz, 2023.
   https://wg21.link/p1928r0
   Adopted for C++26. Provides SIMD vector types for explicit data-parallel programming.

8. **P3304R1: Hazard Pointers for Safe Memory Reclamation**
   P. Dimov, 2025.
   https://wg21.link/p3304r1
   Accepted for C++26. Adds hazard pointers to the Concurrency Technical Specification, enabling lock-free data structures.

9. **P2300R10: `std::execution` — A Standard Library of Asynchronous Algorithms**
   E. Niebler, 2025.
   https://wg21.link/p2300r10
   Adopted for C++26. Provides a model for asynchronous execution, simplifying parallel and concurrent programming.

Official ISO C++ Standards

1. **ISO/IEC 14882:2024 — Programming Languages — C++ (C++23)**
   International Organization for Standardization, 2024.

2. **ISO/IEC 14882:2020 — Programming Languages — C++ (C++20)**
   International Organization for Standardization, 2020.

3. **ISO/IEC 14882:2017 — Programming Languages — C++ (C++17)**
   International Organization for Standardization, 2017.

Recent Books on Modern C++

1. **Stroustrup, Bjarne** — *A Tour of C++ (3rd Edition)*
   Addison-Wesley Professional, 2022. ISBN: 978-0136816485.

2. **Josuttis, Nicolai M.** — *C++ Move Semantics: The Complete Guide*
   Leanpub, 2021. ISBN: 979-8742191137.

3. **Williams, Anthony** — *C++ Concurrency in Action (2nd Edition)*
   Manning Publications, 2019. ISBN: 978-1617294693.

4. **Gregoire, Marc** — *Professional C++ (5th Edition)*
   Wrox, 2021. ISBN: 978-1119695400.

5. **Vandevoorde, David; Josuttis, Nicolai M.; Gregor, Douglas** — *C++ Templates: The Complete Guide (2nd Edition)*
   Addison-Wesley Professional, 2017. ISBN: 978-0321714121.

Essential Online Resources

1. **cppreference.com** — *C++ Reference*
   https://en.cppreference.com
   The definitive online reference for C++ language and library features.

2. **ISO C++ Foundation** — *Standard C++*
   https://isocpp.org

Official home of the C++ standards committee and the C++ community.

3. **C++ Core Guidelines**
   https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
   A set of best practices maintained by Bjarne Stroustrup and Herb Sutter.

4. **Compiler Explorer (godbolt.org)**
   https://godbolt.org
   Interactive tool to explore compiler output and test code snippets.

5. **WG21 Papers**
   https://wg21.link
   Official repository of all C++ standards committee papers.

6. **The C++ Standards Committee (github.com/cplusplus)**
   https://github.com/cplusplus
   Public access to committee papers and related repositories.

7. **Reddit r/cpp**
   https://www.reddit.com/r/cpp/
   Active community discussing C++ news, proposals, and best practices.

Tools for Memory Safety and Debugging

1. **AddressSanitizer (ASan)**
   https://clang.llvm.org/docs/AddressSanitizer.html
   Fast memory error detector built into GCC and Clang.

2. **UndefinedBehaviorSanitizer (UBSan)**
   https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
   Detects undefined behavior at runtime.

3. **Valgrind**
   https://valgrind.org
   Comprehensive memory debugging and profiling tool.

4. **clang-tidy**

https://clang.llvm.org/extra/clang-tidy/
Static analyzer for detecting potential bugs and enforcing best practices.

5. **cppcheck**
   http://cppcheck.sourceforge.net
   Open-source static analyzer for C and C++.

Conference Talks and Videos

1. **CppCon**
   https://cppcon.org/
   Annual C++ conference; talks available on YouTube.

2. **C++Now**
   https://cppnow.org/
   Annual C++ conference with a focus on advanced topics.

3. **Meeting C++**
   https://meetingcpp.com/
   European C++ conference with videos and slides.

4. **C++ Weekly with Jason Turner**
   https://www.youtube.com/@cppweekly
   Short, focused videos on modern C++ features and techniques.

Additional Recent Papers on Pointers and Memory Management

1. **P2549R1: `std::atomic<std::shared_ptr>` — Thread-Safety Improvements**
   J. Wakely, 2023.
   https://wg21.link/p2549r1

2. **P3320R2: `std::inplace_vector` — A Fixed-Capacity Vector**
   G. Dimov, 2025.
   https://wg21.link/p3320r2

3. **P2680R0: `std::is_pointer_interconvertible_with_class` — Type Trait for Pointer Interconvertibility**
   A. G. G. S., 2024.
   https://wg21.link/p2680r0

4. **P2738R2: `constexpr` cast from `void*` — Improving `constexpr` Allocation**
   B. Filipek, 2024.
   https://wg21.link/p2738r2

5. **P3131R1: Hazard Pointers and Read-Copy-Update for the Concurrency TS**
   P. Dimov, 2025.
   https://wg21.link/p3131r1

Acknowledgments

The author thanks the ISO C++ committee members, the open-source community, and all contributors whose work has shaped the language and the resources listed above. Special thanks to the many speakers at CppCon, C++Now, and Meeting C++ whose insights have enriched this book.