https://www.simplifycpp.org

Modern C++ Handbooks: Object-Oriented Programming (OOP) in Modern C++

Prepared by: Ayman Alheraki

Target Audience: Intermediate learners

3

Modern C++ Handbooks:
Object-Oriented Programming (OOP) in Modern C++

Prepared by Ayman Alheraki

Target Audience: Intermediate learner

simplifycpp.org

January 2025

Contents

Modern C++ Handbooks

Introduction to the Modern C++ Handbooks Series
I am pleased to present this series of booklets, compiled from various sources, including books, websites, and GenAI (Generative Artificial Intelligence) systems, to serve as a quick reference for simplifying the C++ language for both beginners and professionals. This series consists of 10 booklets, each approximately 150 pages, covering essential topics of this powerful language, ranging from beginner to advanced levels. I hope that this free series will provide significant value to the followers of https://simplifycpp.org and align with its strategy of simplifying C++. Below is the index of these booklets, which will be included at the beginning of each booklet.

Book 1: Getting Started with Modern C++

- **Target Audience:** Absolute beginners.
- **Content:**
  - **Introduction to C++:**
    - What is C++? Why use Modern C++?
    - History of C++ and the evolution of standards (C++11 to C++23).
  - **Setting Up the Environment:**
    - Installing a modern C++ compiler (GCC, Clang, MSVC).

    * Setting up an IDE (Visual Studio, CLion, VS Code).
    * Using CMake for project management.

– **Writing Your First Program:**

    * Hello World in Modern C++.
    * Understanding `main()`, `#include`, and `using namespace std`.

– **Basic Syntax and Structure:**

    * Variables and data types (`int`, `double`, `bool`, `auto`).
    * Input and output (`std::cin`, `std::cout`).
    * Operators (arithmetic, logical, relational).

– **Control Flow:**

    * `if`, `else`, `switch`.
    * Loops (`for`, `while`, `do-while`).

– **Functions:**

    * Defining and calling functions.
    * Function parameters and return values.
    * Inline functions and `constexpr`.

– **Practical Examples:**

    * Simple programs to reinforce concepts (e.g., calculator, number guessing game).

– **Debugging and Version Control:**

    * Debugging basics (using GDB or IDE debuggers).
    * Introduction to version control (Git).

Book 2: Core Modern C++ Features (C++11 to C++23)

- **Target Audience:** Beginners and intermediate learners.
- **Content:**
  - **C++11 Features:**
    - `auto` keyword for type inference.
    - Range-based `for` loops.
    - `nullptr` for null pointers.
    - Uniform initialization ({} syntax).
    - `constexpr` for compile-time evaluation.
    - Lambda expressions.
    - Move semantics and rvalue references (`std::move`, `std::forward`).
  - **C++14 Features:**
    - Generalized lambda captures.
    - Return type deduction for functions.
    - Relaxed `constexpr` restrictions.
  - **C++17 Features:**
    - Structured bindings.
    - `if` and `switch` with initializers.
    - `inline` variables.
    - Fold expressions.
  - **C++20 Features:**
    - Concepts and constraints.

    - Ranges library.
    - Coroutines.
    - Three-way comparison (<=> operator).
  - **C++23 Features:**
    - `std::expected` for error handling.
    - `std::mdspan` for multidimensional arrays.
    - `std::print` for formatted output.
  - **Practical Examples:**
    - Programs demonstrating each feature (e.g., using lambdas, ranges, and coroutines).
  - **Features and Performance :**
    - Best practices for using Modern C++ features.
    - Performance implications of Modern C++.

Book 3: Object-Oriented Programming (OOP) in Modern C++

- **Target Audience:** Intermediate learners.
- **Content:**
  - **Classes and Objects:**
    - Defining classes and creating objects.
    - Access specifiers (`public`, `private`, `protected`).
  - **Constructors and Destructors:**
    - Default, parameterized, and copy constructors.

    - Move constructors and assignment operators.
    - Destructors and RAII (Resource Acquisition Is Initialization).
  - **Inheritance and Polymorphism:**
    - Base and derived classes.
    - Virtual functions and overriding.
    - Abstract classes and interfaces.
  - **Advanced OOP Concepts:**
    - Multiple inheritance and virtual base classes.
    - `override` and `final` keywords.
    - CRTP (Curiously Recurring Template Pattern).
  - **Practical Examples:**
    - Designing a class hierarchy (e.g., shapes, vehicles).
  - **Design patterns:**
    - Design patterns in Modern C++ (e.g., Singleton, Factory, Observer).

Book 4: Modern C++ Standard Library (STL)

- **Target Audience:** Intermediate learners.
- **Content:**
  - **Containers:**
    - Sequence containers (`std::vector`, `std::list`, `std::deque`).
    - Associative containers (`std::map`, `std::set`, `std::unordered_map`).

    - Container adapters (`std::stack`, `std::queue`, `std::priority_queue`).

  - **Algorithms:**
    - Sorting, searching, and modifying algorithms.
    - Parallel algorithms (C++17).
  - **Utilities:**
    - Smart pointers (`std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`).
    - `std::optional`, `std::variant`, `std::any`.
    - `std::function` and `std::bind`.
  - **Iterators and Ranges:**
    - Iterator categories.
    - Ranges library (C++20).
  - **Practical Examples:**
    - Programs using STL containers and algorithms (e.g., sorting, searching).
  - **Allocators and Benchmarks :**
    - Custom allocators.
    - Performance benchmarks.

Book 5: Advanced Modern C++ Techniques

- **Target Audience:** Advanced learners and professionals.
- **Content:**

- **Templates and Metaprogramming:**
  - Function and class templates.
  - Variadic templates.
  - Type traits and `std::enable_if`.
  - Concepts and constraints (C++20).
- **Concurrency and Parallelism:**
  - Threading (`std::thread`, `std::async`).
  - Synchronization (`std::mutex`, `std::atomic`).
  - Coroutines (C++20).
- **Error Handling:**
  - Exceptions and `noexcept`.
  - `std::optional`, `std::expected` (C++23).
- **Advanced Libraries:**
  - Filesystem library (`std::filesystem`).
  - Networking (C++20 and beyond).
- **Practical Examples:**
  - Advanced programs (e.g., multithreaded applications, template metaprogramming).
- **Lock-free and Memory Management:**
  - Lock-free programming.
  - Custom memory management.

Book 6: Modern C++ Best Practices and Principles

- **Target Audience:** Professionals.

- **Content:**
  - **Code Quality:**
    - Writing clean and maintainable code.
    - Naming conventions and coding standards.
  - **Performance Optimization:**
    - Profiling and benchmarking.
    - Avoiding common pitfalls (e.g., unnecessary copies).
  - **Design Principles:**
    - SOLID principles in Modern C++.
    - Dependency injection.
  - **Testing and Debugging:**
    - Unit testing with frameworks (e.g., Google Test).
    - Debugging techniques and tools.
  - **Security:**
    - Secure coding practices.
    - Avoiding vulnerabilities (e.g., buffer overflows).
  - **Practical Examples:**
    - Case studies of well-designed Modern C++ projects.
  - **Deployment (CI/CD):**
    - Continuous integration and deployment (CI/CD).

Book 7: Specialized Topics in Modern C++

- **Target Audience:** Professionals.
- **Content:**
  - **Scientific Computing:**
    - Numerical methods and libraries (e.g., Eigen, Armadillo).
    - Parallel computing (OpenMP, MPI).
  - **Game Development:**
    - Game engines and frameworks.
    - Graphics programming (Vulkan, OpenGL).
  - **Embedded Systems:**
    - Real-time programming.
    - Low-level hardware interaction.
  - **Practical Examples:**
    - Specialized applications (e.g., simulations, games, embedded systems).
  - **Optimizations:**
    - Domain-specific optimizations.

Book 8: The Future of C++ (Beyond C++23)

- **Target Audience:** Professionals and enthusiasts.
- **Content:**

  - Upcoming features in C++26 and beyond.
  - Reflection and metaclasses.
  - Advanced concurrency models.
  - **Experimental and Developments:**
    * Experimental features and proposals.
    * Community trends and developments.

Book 9: Advanced Topics in Modern C++

- **Target Audience:** Experts and professionals.
- **Content:**
  - **Template Metaprogramming:**
    * SFINAE and `std::enable_if`.
    * Variadic templates and parameter packs.
    * Compile-time computations with `constexpr`.
  - **Advanced Concurrency:**
    * Lock-free data structures.
    * Thread pools and executors.
    * Real-time concurrency.
  - **Memory Management:**
    * Custom allocators.
    * Memory pools and arenas.
    * Garbage collection techniques.

  - **Performance Tuning:**
    * Cache optimization.
    * SIMD (Single Instruction, Multiple Data) programming.
    * Profiling and benchmarking tools.
  - **Advanced Libraries:**
    * Boost library overview.
    * GPU programming (CUDA, SYCL).
    * Machine learning libraries (e.g., TensorFlow C++ API).
  - **Practical Examples:**
    * High-performance computing (HPC) applications.
    * Real-time systems and embedded applications.
  - **C++ projects:**
    * Case studies of cutting-edge C++ projects.

Book 10: Modern C++ in the Real World

- **Target Audience:** Professionals.
- **Content:**
  - **Case Studies:**
    * Real-world applications of Modern C++ (e.g., game engines, financial systems, scientific simulations).
  - **Industry Best Practices:**
    * How top companies use Modern C++.

    - Lessons from large-scale C++ projects.

- **Open Source Contributions:**
    - Contributing to open-source C++ projects.
    - Building your own C++ libraries.
- **Career Development:**
    - Building a portfolio with Modern C++.
    - Preparing for C++ interviews.
- **Networking and conferences :**
    - Networking with the C++ community.
    - Attending conferences and workshops.

Chapter 1

Classes and Objects

1.1 Defining Classes and Creating Objects

1.1.1 Introduction to Classes and Objects

In C++, **classes** and **objects** are the foundation of Object-Oriented Programming (OOP). A class is a blueprint or template for creating objects, which are instances of the class. Classes encapsulate data (attributes) and behavior (methods) into a single entity, promoting modularity, reusability, and abstraction.
This section will guide you through the process of defining classes and creating objects in C++. By the end of this section, you will understand how to design classes, declare member variables and functions, and instantiate objects to interact with them.

1.1.2 Defining a Class

A class in C++ is defined using the `class` keyword. The syntax for defining a class is as follows:

```
class ClassName {
    // Access specifier (private, public, or protected)
    private:
        // Private member variables and functions
    public:
        // Public member variables and functions
    protected:
        // Protected member variables and functions
};
```

- **Access Specifiers:**
  - `private`: Members are accessible only within the class. This is the default access level for class members.
  - `public`: Members are accessible from outside the class.
  - `protected`: Members are accessible within the class and its derived classes (used in inheritance).
- **Member Variables:** These are the attributes or data members of the class.
- **Member Functions:** These are the methods or functions that operate on the member variables.

Example: Defining a Simple Class

```
class Car {
    private:
        std::string brand;  // Private member variable
        int year;           // Private member variable
```

```
    public:
        // Public member functions
        void setBrand(std::string b) {
            brand = b;
        }

        void setYear(int y) {
            year = y;
        }

        void displayInfo() {
            std::cout << "Brand: " << brand << ", Year: " << year <<
            ↪  std::endl;
        }
};
```

In this example:

- The `Car` class has two private member variables: `brand` and `year`.
- It also has three public member functions: `setBrand`, `setYear`, and `displayInfo`.

1.1.3 Creating Objects

Once a class is defined, you can create objects (instances) of that class. Objects are created using the class name followed by the object name.

Syntax for Creating Objects:

```
ClassName objectName;
```

Example: Creating Objects of the `Car` Class

```
int main() {
    Car myCar;  // Creating an object of the Car class

    // Accessing public member functions
    myCar.setBrand("Toyota");
    myCar.setYear(2020);

    // Displaying car information
    myCar.displayInfo();  // Output: Brand: Toyota, Year: 2020

    return 0;
}
```

In this example:

- `myCar` is an object of the `Car` class.
- The `setBrand` and `setYear` methods are used to set the values of the private member variables.
- The `displayInfo` method is used to display the car's information.

1.1.4 Accessing Class Members

- **Public Members:** Public members can be accessed directly using the dot operator (`.`) on the object.
- **Private Members:** Private members cannot be accessed directly from outside the class. They can only be accessed through public member functions (getters and setters).

Example: Accessing Private Members

```
class Car {
    private:
        std::string brand;
        int year;

    public:
        void setBrand(std::string b) {
            brand = b;
        }

        std::string getBrand() {
            return brand;
        }
};

int main() {
    Car myCar;
    myCar.setBrand("Honda");

    // Accessing private member through a public function
    std::cout << "Brand: " << myCar.getBrand() << std::endl;  // Output:
    ↪  Brand: Honda

    return 0;
}
```

In this example:

- The `getBrand` function is used to access the private `brand` member variable.

1.1.5 Constructors and Destructors

- **Constructors:** Special member functions that are automatically called when an object is created. They are used to initialize member variables.
- **Destructors:** Special member functions that are automatically called when an object is destroyed. They are used to release resources.

Example: Using Constructors and Destructors

```
class Car {
    private:
        std::string brand;
        int year;

    public:
        // Constructor
        Car(std::string b, int y) {
            brand = b;
            year = y;
            std::cout << "Car object created: " << brand << std::endl;
        }

        // Destructor
        ~Car() {
            std::cout << "Car object destroyed: " << brand << std::endl;
        }

        void displayInfo() {
            std::cout << "Brand: " << brand << ", Year: " << year <<
            ↪  std::endl;
        }
```

```
};

int main() {
    Car myCar("Ford", 2018);  // Constructor is called
    myCar.displayInfo();       // Output: Brand: Ford, Year: 2018
    return 0;
}  // Destructor is called when myCar goes out of scope
```

In this example:

- The constructor initializes the `brand` and `year` member variables.
- The destructor is called automatically when the object goes out of scope.

1.1.6 Summary

- A **class** is a blueprint for creating objects, encapsulating data and behavior.
- **Objects** are instances of a class, created using the class name.
- **Access specifiers** (`private`, `public`, `protected`) control the visibility of class members.
- **Constructors** initialize objects, while **destructors** clean up resources when objects are destroyed.

By mastering the concepts of defining classes and creating objects, you lay the groundwork for building more complex and reusable C++ programs.

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

1.2.1 Introduction to Access Specifiers

Access specifiers in C++ are keywords that define the visibility and accessibility of class members (variables and functions). They are a fundamental aspect of encapsulation, one of the core principles of Object-Oriented Programming (OOP). By controlling access to class members, you can enforce data hiding, improve code maintainability, and design robust and secure systems.
C++ provides three access specifiers:

1. **`public`**: Members are accessible from anywhere.
2. **`private`**: Members are accessible only within the class.
3. **`protected`**: Members are accessible within the class and its derived classes (used in inheritance).

This section will explain each access specifier in detail, along with examples and best practices for their use.

1.2.2 `public` Access Specifier

- **Visibility:** Members declared as `public` are accessible from anywhere, both inside and outside the class.
- **Use Case:** Typically used for member functions that provide an interface to the class, allowing external code to interact with the class.

Example: `public` Members

```
class Circle {
    public:
        double radius;  // Public member variable

        // Public member function
        double calculateArea() {
            return 3.14159 * radius * radius;
        }
};

int main() {
    Circle myCircle;
    myCircle.radius = 5.0;  // Accessing public member variable
    std::cout << "Area: " << myCircle.calculateArea() << std::endl;  //
    ↪  Accessing public member function
    return 0;
}
```

In this example:

- The `radius` member variable and the `calculateArea` member function are declared as `public`.
- They can be accessed directly from the `main` function.

1.2.3 `private` Access Specifier

- **Visibility:** Members declared as `private` are accessible only within the class. They cannot be accessed directly from outside the class.
- **Use Case:** Used to hide implementation details and enforce data encapsulation. Private members are typically accessed through public member functions (getters and setters).

Example: `private` Members

```
class Circle {
    private:
        double radius;  // Private member variable

    public:
        // Public member function to set radius
        void setRadius(double r) {
            if (r > 0) {
                radius = r;
            } else {
                std::cout << "Invalid radius!" << std::endl;
            }
        }

        // Public member function to get radius
        double getRadius() {
            return radius;
        }

        // Public member function to calculate area
        double calculateArea() {
            return 3.14159 * radius * radius;
        }
};

int main() {
    Circle myCircle;
    myCircle.setRadius(5.0);  // Accessing private member through a public
    ↪ function
    std::cout << "Radius: " << myCircle.getRadius() << std::endl;  //
    ↪ Accessing private member through a public function
```

```
    std::cout << "Area: " << myCircle.calculateArea() << std::endl;
    return 0;
}
```

In this example:

- The `radius` member variable is declared as `private`.
- It can only be accessed or modified through the public member functions `setRadius` and `getRadius`.

1.2.4 `protected` Access Specifier

- **Visibility:** Members declared as `protected` are accessible within the class and its derived classes (used in inheritance). They are not accessible from outside the class hierarchy.
- **Use Case:** Used when you want to allow derived classes to access certain members while still hiding them from external code.

Example: `protected` Members

```
class Shape {
    protected:
        double width;
        double height;

    public:
        void setDimensions(double w, double h) {
            width = w;
            height = h;
```

```
        }
};

class Rectangle : public Shape {
    public:
        double calculateArea() {
            return width * height;  // Accessing protected members from
            ↪  the derived class
        }
};

int main() {
    Rectangle myRect;
    myRect.setDimensions(5.0, 3.0);  // Accessing public member function
    ↪  of the base class
    std::cout << "Area: " << myRect.calculateArea() << std::endl;  //
    ↪  Accessing public member function of the derived class
    return 0;
}
```

In this example:

- The `width` and `height` member variables are declared as `protected` in the `Shape` class.
- They are accessible in the derived class `Rectangle` but not from outside the class hierarchy.

1.2.5 Access Specifiers and Encapsulation

Access specifiers play a crucial role in encapsulation, which is the bundling of data and methods that operate on that data within a single unit (the class). By controlling access to class members,

you can:

- **Prevent unauthorized access:** Ensure that sensitive data is not modified directly.
- **Improve code maintainability:** Hide implementation details, allowing you to change the internal workings of a class without affecting external code.
- **Enforce validation:** Use public member functions to validate data before modifying private members.

Example: Encapsulation in Action

```
class BankAccount {
    private:
        double balance;  // Private member variable

    public:
        // Public member function to deposit money
        void deposit(double amount) {
            if (amount > 0) {
                balance += amount;
            } else {
                std::cout << "Invalid deposit amount!" << std::endl;
            }
        }

        // Public member function to withdraw money
        void withdraw(double amount) {
            if (amount > 0 && amount <= balance) {
                balance -= amount;
            } else {
                std::cout << "Invalid withdrawal amount!" << std::endl;
```

```
        }
    }

    // Public member function to get balance
    double getBalance() {
        return balance;
    }
};

int main() {
    BankAccount account;
    account.deposit(1000.0);  // Accessing public member function
    account.withdraw(500.0);  // Accessing public member function
    std::cout << "Balance: " << account.getBalance() << std::endl;  //
    ↪  Accessing public member function
    return 0;
}
```

In this example:

- The `balance` member variable is private, ensuring it cannot be modified directly.
- Public member functions (`deposit`, `withdraw`, and `getBalance`) provide controlled access to the `balance`.

1.2.6 Best Practices for Using Access Specifiers

1. **Use `private` for Data Hiding:**
    - Declare member variables as `private` to prevent direct access and modification from outside the class.
    - Use public member functions (getters and setters) to provide controlled access.

2. **Use `public` for Interfaces:**
    - Declare member functions that define the class's interface as `public`.
    - Ensure that public functions are well-documented and easy to use.
3. **Use `protected` for Inheritance:**
    - Use `protected` for members that need to be accessible in derived classes but hidden from external code.
    - Be cautious when using `protected`, as it can lead to tightly coupled class hierarchies.
4. **Minimize the Use of `public` Member Variables:**
    - Avoid declaring member variables as `public` unless absolutely necessary.
    - Public member variables break encapsulation and make the class harder to maintain.

1.2.7 Summary

- **`public`:** Members are accessible from anywhere.
- **`private`:** Members are accessible only within the class.
- **`protected`:** Members are accessible within the class and its derived classes.
- Access specifiers are essential for encapsulation, data hiding, and designing robust class hierarchies.

By understanding and effectively using access specifiers, you can create well-structured, maintainable, and secure C++ programs.

Chapter 2

Constructors and Destructors:

2.1 Default, Parameterized, and Copy Constructors

2.1.1 Introduction to Constructors

Constructors are special member functions in C++ that are automatically called when an object of a class is created. They are used to initialize the object's member variables and set up any necessary resources. Constructors have the same name as the class and do not have a return type, not even `void`.
C++ supports several types of constructors:

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**

This section will explain each type of constructor in detail, along with examples and best practices for their use.

2.1.2 Default Constructor

- **Definition:** A default constructor is a constructor that takes no arguments. It is automatically called when an object is created without any initial values.
- **Use Case:** Used to initialize objects with default values or perform setup tasks.

Example: Default Constructor

```
class Rectangle {
    private:
        double width;
        double height;

    public:
        // Default constructor
        Rectangle() {
            width = 1.0;
            height = 1.0;
            std::cout << "Default constructor called!" << std::endl;
        }

        // Member function to calculate area
        double calculateArea() {
            return width * height;
        }
};

int main() {
    Rectangle rect1;  // Default constructor is called
    std::cout << "Area: " << rect1.calculateArea() << std::endl;  //
    ↪  Output: Area: 1
    return 0;
```

```
}
```

In this example:

- The `Rectangle` class has a default constructor that initializes `width` and `height` to `1.0`.
- When `rect1` is created, the default constructor is called automatically.

2.1.3 Parameterized Constructor

- **Definition:** A parameterized constructor is a constructor that takes one or more arguments. It is used to initialize objects with specific values provided at the time of creation.
- **Use Case:** Used to initialize objects with user-defined values.

Example: Parameterized Constructor

```
class Rectangle {
    private:
        double width;
        double height;

    public:
        // Parameterized constructor
        Rectangle(double w, double h) {
            width = w;
            height = h;
            std::cout << "Parameterized constructor called!" << std::endl;
        }
```

```
        // Member function to calculate area
        double calculateArea() {
            return width * height;
        }
};

int main() {
    Rectangle rect2(5.0, 3.0);  // Parameterized constructor is called
    std::cout << "Area: " << rect2.calculateArea() << std::endl;  //
    ↪  Output: Area: 15
    return 0;
}
```

In this example:

- The `Rectangle` class has a parameterized constructor that takes two arguments (`w` and `h`) to initialize `width` and `height`.
- When `rect2` is created with arguments `5.0` and `3.0`, the parameterized constructor is called.

2.1.4 Copy Constructor

- **Definition:** A copy constructor is a constructor that initializes an object using another object of the same class. It is used to create a copy of an existing object.
- **Use Case:** Used when you need to create a new object that is a copy of an existing object.

Example: Copy Constructor

```cpp
class Rectangle {
    private:
        double width;
        double height;

    public:
        // Parameterized constructor
        Rectangle(double w, double h) {
            width = w;
            height = h;
            std::cout << "Parameterized constructor called!" << std::endl;
        }

        // Copy constructor
        Rectangle(const Rectangle &other) {
            width = other.width;
            height = other.height;
            std::cout << "Copy constructor called!" << std::endl;
        }

        // Member function to calculate area
        double calculateArea() {
            return width * height;
        }
};

int main() {
    Rectangle rect3(4.0, 6.0);  // Parameterized constructor is called
    Rectangle rect4 = rect3;    // Copy constructor is called
    std::cout << "Area: " << rect4.calculateArea() << std::endl;  //
    ↪ Output: Area: 24
    return 0;
```

```
}
```

In this example:

- The `Rectangle` class has a copy constructor that takes a `const` reference to another `Rectangle` object.
- When `rect4` is created as a copy of `rect3`, the copy constructor is called.

2.1.5 When Are Constructors Called?

1. **Default Constructor:**
    - Called when an object is created without any arguments.
    - Example: `Rectangle rect1;`
2. **Parameterized Constructor:**
    - Called when an object is created with arguments.
    - Example: `Rectangle rect2(5.0, 3.0);`
3. **Copy Constructor:**
    - Called when an object is initialized using another object of the same class.
    - Example: `Rectangle rect4 = rect3;`

2.1.6 Best Practices for Using Constructors

1. **Always Define a Default Constructor:**

- If your class does not have a default constructor, the compiler will generate one for you. However, it is good practice to define your own default constructor to ensure proper initialization.

2. **Use Parameterized Constructors for Flexibility:**

   - Parameterized constructors allow you to initialize objects with specific values, making your class more flexible and reusable.

3. **Implement a Copy Constructor for Deep Copying:**

   - If your class manages dynamic memory or other resources, implement a copy constructor to ensure deep copying and avoid issues like shallow copying.

4. **Use Member Initialization Lists:**

   - Use member initialization lists to initialize member variables efficiently, especially for parameterized constructors.

Example: Member Initialization List

```
class Rectangle {
    private:
        double width;
        double height;

    public:
        // Parameterized constructor with member initialization list
        Rectangle(double w, double h) : width(w), height(h) {
            std::cout << "Parameterized constructor called!" << std::endl;
        }
```

```
        // Copy constructor with member initialization list
        Rectangle(const Rectangle &other) : width(other.width),
        ↪  height(other.height) {
            std::cout << "Copy constructor called!" << std::endl;
        }

        // Member function to calculate area
        double calculateArea() {
            return width * height;
        }
};
```

In this example:

- The parameterized and copy constructors use member initialization lists to initialize `width` and `height`.

2.1.7 Summary

- **Default Constructor:** Initializes objects with default values.
- **Parameterized Constructor:** Initializes objects with user-defined values.
- **Copy Constructor:** Initializes objects as copies of existing objects.
- Constructors are essential for proper object initialization and resource management.

By mastering the use of constructors, you can create well-initialized and robust C++ classes.

2.2 Move Constructors and Assignment Operators

2.2.1 Introduction to Move Semantics

Move semantics is a feature introduced in C++11 that allows the efficient transfer of resources (such as dynamically allocated memory) from one object to another. This is particularly useful for optimizing performance when working with large objects or resource-heavy classes. Move semantics is implemented using **move constructors** and **move assignment operators**.
This section will explain move constructors and move assignment operators in detail, along with examples and best practices for their use.

2.2.2 Move Constructor

- **Definition:** A move constructor is a special constructor that takes an rvalue reference to an object of the same class. It ”moves” the resources from the source object to the newly created object, leaving the source object in a valid but unspecified state.
- **Use Case:** Used to efficiently transfer resources when an object is initialized using an rvalue (e.g., a temporary object or the result of `std::move`).

Example: Move Constructor

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

class String {
    private:
        char* data;
        size_t length;
```

```
    public:
        // Parameterized constructor
        String(const char* str) {
            length = std::strlen(str);
            data = new char[length + 1];
            std::strcpy(data, str);
            std::cout << "Parameterized constructor called!" << std::endl;
        }

        // Move constructor
        String(String&& other) noexcept {
            data = other.data;       // Transfer ownership of the resource
            length = other.length;   // Copy the length
            other.data = nullptr;    // Invalidate the source object
            other.length = 0;        // Reset the source object's length
            std::cout << "Move constructor called!" << std::endl;
        }

        // Destructor
        ~String() {
            delete[] data;
            std::cout << "Destructor called!" << std::endl;
        }

        // Member function to display the string
        void display() const {
            std::cout << "String: " << (data ? data : "nullptr") <<
            ↪  std::endl;
        }
};

int main() {
```

```
    String str1("Hello, World!");  // Parameterized constructor called
    String str2 = std::move(str1); // Move constructor called

    str1.display();  // Output: String: nullptr
    str2.display();  // Output: String: Hello, World!

    return 0;
}
```

In this example:

- The `String` class has a move constructor that transfers ownership of the `data` pointer from the source object (`other`) to the newly created object.
- After the move, the source object (`str1`) is left in a valid but unspecified state (its `data` pointer is set to `nullptr`).

2.2.3 Move Assignment Operator

- **Definition:** A move assignment operator is a special assignment operator that takes an rvalue reference to an object of the same class. It ”moves” the resources from the source object to the destination object, leaving the source object in a valid but unspecified state.
- **Use Case:** Used to efficiently transfer resources when an object is assigned an rvalue (e.g., a temporary object or the result of `std::move`).

Example: Move Assignment Operator

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

```
class String {
    private:
        char* data;
        size_t length;

    public:
        // Parameterized constructor
        String(const char* str) {
            length = std::strlen(str);
            data = new char[length + 1];
            std::strcpy(data, str);
            std::cout << "Parameterized constructor called!" << std::endl;
        }

        // Move constructor
        String(String&& other) noexcept {
            data = other.data;
            length = other.length;
            other.data = nullptr;
            other.length = 0;
            std::cout << "Move constructor called!" << std::endl;
        }

        // Move assignment operator
        String& operator=(String&& other) noexcept {
            if (this != &other) {  // Check for self-assignment
                delete[] data;     // Release existing resources

                data = other.data;       // Transfer ownership of the
                ↪  resource
                length = other.length;   // Copy the length
                other.data = nullptr;    // Invalidate the source object
```

```
            other.length = 0;        // Reset the source object's
            ↪ length
            std::cout << "Move assignment operator called!" <<
            ↪ std::endl;
        }
        return *this;
    }

    // Destructor
    ~String() {
        delete[] data;
        std::cout << "Destructor called!" << std::endl;
    }

    // Member function to display the string
    void display() const {
        std::cout << "String: " << (data ? data : "nullptr") <<
        ↪ std::endl;
    }
};

int main() {
    String str1("Hello, World!");  // Parameterized constructor called
    String str2("Goodbye!");       // Parameterized constructor called

    str2 = std::move(str1);  // Move assignment operator called

    str1.display();  // Output: String: nullptr
    str2.display();  // Output: String: Hello, World!

    return 0;
}
```

In this example:

- The `String` class has a move assignment operator that transfers ownership of the `data` pointer from the source object (`other`) to the destination object (`*this`).
- After the move, the source object (`str1`) is left in a valid but unspecified state (its `data` pointer is set to `nullptr`).

2.2.4 Key Points About Move Semantics

1. **Rvalue References:**
    - Move constructors and move assignment operators use rvalue references (`&&`) to identify temporary objects or objects explicitly marked with `std::move`.
2. **Noexcept:**
    - Move constructors and move assignment operators should be marked as `noexcept` to indicate that they do not throw exceptions. This allows certain optimizations in the standard library.
3. **Invalidating the Source Object:**
    - After a move operation, the source object should be left in a valid but unspecified state. Typically, this means setting its resource pointers to `nullptr`.
4. **Self-Assignment Check:**
    - In the move assignment operator, always check for self-assignment (`if (this != &other)`) to avoid releasing resources prematurely.

2.2.5 Best Practices for Move Semantics

1. **Implement Move Semantics for Resource-Managing Classes:**
    - If your class manages dynamic memory, file handles, or other resources, implement move constructors and move assignment operators to optimize performance.
2. **Use `std::move` to Enable Move Semantics:**
    - Use `std::move` to explicitly indicate that an object can be moved from. This is particularly useful when passing objects to functions or returning them from functions.
3. **Mark Move Operations as `noexcept`:**
    - Mark move constructors and move assignment operators as `noexcept` to enable optimizations and ensure compatibility with standard library containers.
4. **Follow the Rule of Five:**
    - If you define a move constructor or move assignment operator, you should also define the copy constructor, copy assignment operator, and destructor to ensure proper resource management.

Example: Rule of Five

```
class String {
    private:
        char* data;
        size_t length;
```

```
public:
    // Parameterized constructor
    String(const char* str) {
        length = std::strlen(str);
        data = new char[length + 1];
        std::strcpy(data, str);
        std::cout << "Parameterized constructor called!" << std::endl;
    }

    // Copy constructor
    String(const String& other) {
        length = other.length;
        data = new char[length + 1];
        std::strcpy(data, other.data);
        std::cout << "Copy constructor called!" << std::endl;
    }

    // Move constructor
    String(String&& other) noexcept {
        data = other.data;
        length = other.length;
        other.data = nullptr;
        other.length = 0;
        std::cout << "Move constructor called!" << std::endl;
    }

    // Copy assignment operator
    String& operator=(const String& other) {
        if (this != &other) {
            delete[] data;
            length = other.length;
            data = new char[length + 1];
```

```
            std::strcpy(data, other.data);
            std::cout << "Copy assignment operator called!" <<
            ↪  std::endl;
        }
        return *this;
    }

    // Move assignment operator
    String& operator=(String&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            length = other.length;
            other.data = nullptr;
            other.length = 0;
            std::cout << "Move assignment operator called!" <<
            ↪  std::endl;
        }
        return *this;
    }

    // Destructor
    ~String() {
        delete[] data;
        std::cout << "Destructor called!" << std::endl;
    }

    // Member function to display the string
    void display() const {
        std::cout << "String: " << (data ? data : "nullptr") <<
        ↪  std::endl;
    }
```

```
};
```

In this example:

- The `String` class follows the Rule of Five by implementing the copy constructor, copy assignment operator, move constructor, move assignment operator, and destructor.

2.2.6 Summary

- **Move Constructor:** Efficiently transfers resources from a source object to a newly created object.
- **Move Assignment Operator:** Efficiently transfers resources from a source object to an existing object.
- Move semantics is a powerful feature for optimizing performance when working with resource-heavy classes.

By mastering move constructors and move assignment operators, you can write efficient and modern C++ code.

2.3 Destructors and RAII (Resource Acquisition Is Initialization)

2.3.1 Introduction to Destructors and RAII

Destructors are special member functions in C++ that are automatically called when an object goes out of scope or is explicitly deleted. They are used to release resources (such as dynamically allocated memory, file handles, or network connections) that the object may have acquired during its lifetime.
RAII (Resource Acquisition Is Initialization) is a programming idiom that ties resource management to object lifetime. It ensures that resources are properly released when an object is destroyed, preventing resource leaks and ensuring exception safety.
This section will explain destructors and the RAII idiom in detail, along with examples and best practices for their use.

2.3.2 Destructors

- **Definition:** A destructor is a special member function with the same name as the class, prefixed with a tilde (~). It has no parameters, no return type, and cannot be overloaded.
- **Use Case:** Used to release resources acquired by the object, such as freeing dynamically allocated memory, closing files, or releasing locks.

Example: Destructor

```
#include <iostream>

class FileHandler {
    private:
```

```
        FILE* file;

    public:
        // Constructor
        FileHandler(const char* filename, const char* mode) {
            file = std::fopen(filename, mode);
            if (file) {
                std::cout << "File opened successfully!" << std::endl;
            } else {
                std::cout << "Failed to open file!" << std::endl;
            }
        }

        // Destructor
        ~FileHandler() {
            if (file) {
                std::fclose(file);
                std::cout << "File closed successfully!" << std::endl;
            }
        }

        // Member function to write to the file
        void write(const char* text) {
            if (file) {
                std::fprintf(file, "%s", text);
            }
        }
};

int main() {
    FileHandler fh("example.txt", "w");
    fh.write("Hello, World!");
```

```
    // Destructor is called automatically when fh goes out of scope
    return 0;
}
```

In this example:

- The `FileHandler` class manages a file resource using a `FILE*` pointer.
- The constructor opens the file, and the destructor closes it.
- When the `FileHandler` object (`fh`) goes out of scope, the destructor is called automatically, ensuring the file is closed.

2.3.3 RAII (Resource Acquisition Is Initialization)

- **Definition:** RAII is a programming idiom where resource acquisition is tied to object initialization. Resources are acquired in the constructor and released in the destructor, ensuring proper cleanup even in the presence of exceptions.
- **Use Case:** Used to manage resources such as memory, file handles, network connections, and locks in a safe and exception-safe manner.

Example: RAII with Dynamic Memory

```
#include <iostream>

class Buffer {
    private:
        int* data;
        size_t size;
```

```
    public:
        // Constructor
        Buffer(size_t sz) : size(sz) {
            data = new int[size];
            std::cout << "Buffer allocated!" << std::endl;
        }

        // Destructor
        ~Buffer() {
            delete[] data;
            std::cout << "Buffer deallocated!" << std::endl;
        }

        // Member function to access the buffer
        int& operator[](size_t index) {
            return data[index];
        }
};

int main() {
    Buffer buf(10);  // Buffer allocated
    buf[0] = 42;     // Access the buffer
    // Destructor is called automatically when buf goes out of scope
    return 0;
}
```

In this example:

- The `Buffer` class manages a dynamically allocated array of integers.
- The constructor allocates memory, and the destructor deallocates it.
- When the `Buffer` object (`buf`) goes out of scope, the destructor is called automatically,

ensuring the memory is freed.

2.3.4 Exception Safety and RAII

RAII ensures exception safety by guaranteeing that resources are released even if an exception is thrown. This is because destructors are called automatically when an object goes out of scope, regardless of how the scope is exited (e.g., normal execution or exception).

Example: Exception Safety with RAII

```
#include <iostream>
#include <stdexcept>

class Resource {
    public:
        Resource() {
            std::cout << "Resource acquired!" << std::endl;
        }

        ~Resource() {
            std::cout << "Resource released!" << std::endl;
        }

        void use() {
            std::cout << "Resource in use!" << std::endl;
            throw std::runtime_error("Error occurred!");
        }
};

int main() {
    try {
        Resource res;  // Resource acquired
```

```
        res.use();      // Throws an exception
    } catch (const std::exception& e) {
        std::cout << "Exception: " << e.what() << std::endl;
    }
    // Destructor is called automatically when res goes out of scope
    return 0;
}
```

In this example:

- The `Resource` class simulates a resource that throws an exception when used.
- Even though an exception is thrown, the destructor is called automatically, ensuring the resource is released.

2.3.5 Best Practices for Destructors and RAII

1. **Release Resources in Destructors:**
    - Always release resources (e.g., memory, file handles, locks) in the destructor to prevent resource leaks.
2. **Use RAII for Resource Management:**
    - Use RAII to manage resources by tying their acquisition to object initialization and their release to object destruction.
3. **Mark Destructors as `noexcept`:**
    - Destructors should not throw exceptions. Mark them as `noexcept` to ensure they terminate the program if an exception is thrown.

4. **Follow the Rule of Five:**

    - If your class manages resources, implement the Rule of Five (destructor, copy constructor, copy assignment operator, move constructor, and move assignment operator) to ensure proper resource management.

Example: Rule of Five with RAII

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

class String {
    private:
        char* data;
        size_t length;

    public:
        // Parameterized constructor
        String(const char* str) {
            length = std::strlen(str);
            data = new char[length + 1];
            std::strcpy(data, str);
            std::cout << "Parameterized constructor called!" << std::endl;
        }

        // Copy constructor
        String(const String& other) {
            length = other.length;
            data = new char[length + 1];
            std::strcpy(data, other.data);
            std::cout << "Copy constructor called!" << std::endl;
        }
```

```
    // Move constructor
    String(String&& other) noexcept {
        data = other.data;
        length = other.length;
        other.data = nullptr;
        other.length = 0;
        std::cout << "Move constructor called!" << std::endl;
    }

    // Copy assignment operator
    String& operator=(const String& other) {
        if (this != &other) {
            delete[] data;
            length = other.length;
            data = new char[length + 1];
            std::strcpy(data, other.data);
            std::cout << "Copy assignment operator called!" <<
            ↪  std::endl;
        }
        return *this;
    }

    // Move assignment operator
    String& operator=(String&& other) noexcept {
        if (this != &other) {
            delete[] data;
            data = other.data;
            length = other.length;
            other.data = nullptr;
            other.length = 0;
            std::cout << "Move assignment operator called!" <<
            ↪  std::endl;
```

```
            }
            return *this;
        }

        // Destructor
        ~String() noexcept {
            delete[] data;
            std::cout << "Destructor called!" << std::endl;
        }

        // Member function to display the string
        void display() const {
            std::cout << "String: " << (data ? data : "nullptr") <<
            ↪  std::endl;
        }
};

int main() {
    String str1("Hello, World!");  // Parameterized constructor called
    String str2 = str1;            // Copy constructor called
    String str3 = std::move(str1); // Move constructor called

    str1.display();  // Output: String: nullptr
    str2.display();  // Output: String: Hello, World!
    str3.display();  // Output: String: Hello, World!

    return 0;
}
```

In this example:

- The `String` class follows the Rule of Five and uses RAII to manage a dynamically allocated string.

2.3.6 Summary

- **Destructors:** Automatically called when an object goes out of scope or is deleted. Used to release resources.
- **RAII:** Ties resource management to object lifetime, ensuring resources are properly released.
- **Exception Safety:** RAII guarantees resource cleanup even in the presence of exceptions.
- **Best Practices:** Release resources in destructors, use RAII for resource management, and follow the Rule of Five.

By mastering destructors and the RAII idiom, you can write safe, efficient, and exception-resistant C++ code.

Chapter 3

Inheritance and Polymorphism:

3.1 Base and Derived Classes

3.1.1 Introduction to Inheritance

Inheritance is one of the core concepts of Object-Oriented Programming (OOP) in C++. It allows you to create a new class (called a **derived class**) based on an existing class (called a **base class**). The derived class inherits the properties (member variables) and behaviors (member functions) of the base class, enabling code reuse and the creation of hierarchical relationships between classes.

This section will explain the concepts of base and derived classes, how to define them, and how inheritance works in C++. By the end of this section, you will understand how to use inheritance to build more modular and reusable code.

3.1.2 Base Class

- **Definition:** A base class (also called a parent class or superclass) is an existing class from which other classes can inherit. It defines the common properties and behaviors that are shared by all derived classes.

- **Use Case:** Used to encapsulate common functionality that can be reused across multiple derived classes.

Example: Base Class

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

// Base class
class Animal {
    private:
        std::string name;

    public:
        // Constructor
        Animal(const std::string& name) : name(name) {}

        // Public member function
        void speak() const {
            std::cout << name << " makes a sound." << std::endl;
        }

        // Getter for name
        std::string getName() const {
            return name;
```

```
        }
};
```

In this example:

- The `Animal` class is a base class with a private member variable `name` and public member functions `speak` and `getName`.
- The `speak` function provides a default behavior for all animals.

3.1.3 Derived Class

- **Definition:** A derived class (also called a child class or subclass) is a class that inherits from a base class. It can add new properties and behaviors or override existing ones.
- **Use Case:** Used to extend or specialize the functionality of the base class.

Syntax for Defining a Derived Class:

```
class DerivedClass : access-specifier BaseClass {
    // Derived class members
};
```

- **Access Specifier:** Determines the accessibility of the base class members in the derived class. It can be `public`, `private`, or `protected`.

Example: Derived Class

```
// Derived class
class Dog : public Animal {
    public:
        // Constructor
        Dog(const std::string& name) : Animal(name) {}

        // Override the speak function
        void speak() const {
            std::cout << getName() << " barks." << std::endl;
        }
};
```

In this example:

- The `Dog` class is a derived class that inherits from the `Animal` class using the `public` access specifier.
- The `Dog` class overrides the `speak` function to provide a specialized behavior for dogs.

3.1.4 Access Specifiers in Inheritance

The access specifier used in inheritance determines how the base class members are accessible in the derived class:

1. **`public` Inheritance:**
    - Public members of the base class become public members of the derived class.
    - Protected members of the base class become protected members of the derived class.
    - Private members of the base class are not accessible in the derived class.
2. **`protected` Inheritance:**

- Public and protected members of the base class become protected members of the derived class.
- Private members of the base class are not accessible in the derived class.

3. **`private` Inheritance:**

- Public and protected members of the base class become private members of the derived class.
- Private members of the base class are not accessible in the derived class.

Example: Access Specifiers in Inheritance

```
#include <iostream>

class Base {
    public:
        Base() {
            std::cout << "Base constructor called!" << std::endl;
        }

        ~Base() {
            std::cout << "Base destructor called!" << std::endl;
        }
};

class Derived : public Base {
    public:
        Derived() {
            std::cout << "Derived constructor called!" << std::endl;
        }
```

```
        ~Derived() {
            std::cout << "Derived destructor called!" << std::endl;
        }
};

int main() {
    Derived d;  // Base constructor called, then Derived constructor
    ↪  called
    return 0;   // Derived destructor called, then Base destructor called
}
```

In this example:

- The accessibility of base class members in the derived class depends on the access specifier used in inheritance.

3.1.5 Constructors and Destructors in Inheritance

- **Constructors:** When a derived class object is created, the base class constructor is called first, followed by the derived class constructor.
- **Destructors:** When a derived class object is destroyed, the derived class destructor is called first, followed by the base class destructor.

Example: Constructors and Destructors in Inheritance

```
#include <iostream>

class Base {
    public:
        Base() {
```

```
            std::cout << "Base constructor called!" << std::endl;
        }

        ~Base() {
            std::cout << "Base destructor called!" << std::endl;
        }
};

class Derived : public Base {
    public:
        Derived() {
            std::cout << "Derived constructor called!" << std::endl;
        }

        ~Derived() {
            std::cout << "Derived destructor called!" << std::endl;
        }
};

int main() {
    Derived d;  // Base constructor called, then Derived constructor
    ↪  called
    return 0;   // Derived destructor called, then Base destructor called
}
```

In this example:

- The base class constructor is called before the derived class constructor.
- The derived class destructor is called before the base class destructor.

3.1.6 Function Overriding

- **Definition:** Function overriding occurs when a derived class defines a function with the same name and signature as a function in the base class. The derived class function overrides the base class function.
- **Use Case:** Used to provide specialized behavior in the derived class.

Example: Function Overriding

```
#include <iostream>

class Base {
    public:
        void show() {
            std::cout << "Base class show function." << std::endl;
        }
};

class Derived : public Base {
    public:
        void show() {
            std::cout << "Derived class show function." << std::endl;
        }
};

int main() {
    Derived d;
    d.show();  // Output: Derived class show function.
    return 0;
}
```

In this example:

- The `show` function in the `Derived` class overrides the `show` function in the `Base` class.

3.1.7 Summary

- **Base Class:** Defines common properties and behaviors shared by derived classes.
- **Derived Class:** Inherits from a base class and can add or override functionality.
- **Access Specifiers:** Control the accessibility of base class members in derived classes.
- **Constructors and Destructors:** Base class constructors are called before derived class constructors, and derived class destructors are called before base class destructors.
- **Function Overriding:** Allows derived classes to provide specialized behavior for base class functions.

By understanding base and derived classes, you can leverage inheritance to create modular, reusable, and hierarchical class structures.

3.2 Virtual Functions and Overriding

3.2.1 Introduction to Virtual Functions

Virtual functions are a key feature of C++ that enable **runtime polymorphism**. They allow a derived class to provide a specific implementation of a function that is already defined in its base class. When you call a virtual function on a base class pointer or reference, the actual function that gets executed is determined by the type of the object being pointed to or referenced, not the type of the pointer or reference itself.
This section will explain virtual functions, how to override them in derived classes, and how they enable dynamic binding in C++. By the end of this section, you will understand how to use virtual functions to achieve polymorphic behavior in your programs.

3.2.2 Virtual Functions

- **Definition:** A virtual function is a member function declared in a base class with the `virtual` keyword. It can be overridden in derived classes to provide specific implementations.
- **Use Case:** Used to achieve runtime polymorphism, where the function to be executed is determined at runtime based on the actual object type.

Syntax for Declaring a Virtual Function:

```
virtual ReturnType FunctionName(Parameters) {
    // Function implementation
}
```

Example: Virtual Function

```
#include <iostream>

// Base class
class Animal {
    public:
        // Virtual function
        virtual void speak() const {
            std::cout << "Animal speaks." << std::endl;
        }
};

// Derived class
class Dog : public Animal {
    public:
```

```
        // Override the virtual function
        void speak() const override {
            std::cout << "Dog barks." << std::endl;
        }
};

// Derived class
class Cat : public Animal {
    public:
        // Override the virtual function
        void speak() const override {
            std::cout << "Cat meows." << std::endl;
        }
};

int main() {
    Animal* animal1 = new Dog();  // Base class pointer pointing to a Dog
    ↪ object
    Animal* animal2 = new Cat();  // Base class pointer pointing to a Cat
    ↪ object

    animal1->speak();  // Output: Dog barks.
    animal2->speak();  // Output: Cat meows.

    delete animal1;
    delete animal2;

    return 0;
}
```

In this example:

- The `Animal` class declares a virtual function `speak`.

- The `Dog` and `Cat` classes override the `speak` function to provide specific implementations.
- When `speak` is called on a base class pointer (`Animal*`), the actual function executed depends on the type of the object being pointed to (`Dog` or `Cat`).

3.2.3 Overriding Virtual Functions

- **Definition:** Overriding a virtual function means providing a new implementation of the function in a derived class. The function in the derived class must have the same signature (name, return type, and parameters) as the virtual function in the base class.
- **Use Case:** Used to customize or extend the behavior of a base class function in derived classes.

Syntax for Overriding a Virtual Function:

```
ReturnType FunctionName(Parameters) override {
    // New implementation
}
```

- The `override` keyword is optional but recommended. It ensures that the function is actually overriding a virtual function in the base class, providing better readability and error detection.

Example: Overriding Virtual Functions

```
#include <iostream>
```

```cpp
// Base class
class Shape {
    public:
        // Virtual function
        virtual void draw() const {
            std::cout << "Drawing a shape." << std::endl;
        }
};

// Derived class
class Circle : public Shape {
    public:
        // Override the virtual function
        void draw() const override {
            std::cout << "Drawing a circle." << std::endl;
        }
};

// Derived class
class Rectangle : public Shape {
    public:
        // Override the virtual function
        void draw() const override {
            std::cout << "Drawing a rectangle." << std::endl;
        }
};

int main() {
    Shape* shape1 = new Circle();     // Base class pointer pointing to a
    ↪ Circle object
    Shape* shape2 = new Rectangle();  // Base class pointer pointing to a
    ↪ Rectangle object
```

```
    shape1->draw();  // Output: Drawing a circle.
    shape2->draw();  // Output: Drawing a rectangle.

    delete shape1;
    delete shape2;

    return 0;
}
```

In this example:

- The `Shape` class declares a virtual function `draw`.
- The `Circle` and `Rectangle` classes override the `draw` function to provide specific implementations.
- When `draw` is called on a base class pointer (`Shape*`), the actual function executed depends on the type of the object being pointed to (`Circle` or `Rectangle`).

3.2.4 Pure Virtual Functions and Abstract Classes

- **Pure Virtual Function:** A virtual function with no implementation in the base class. It is declared using `= 0` in the function declaration.
- **Abstract Class:** A class that contains at least one pure virtual function. It cannot be instantiated directly and serves as a base class for derived classes.

Syntax for Declaring a Pure Virtual Function:

```
virtual ReturnType FunctionName(Parameters) = 0;
```

Example: Pure Virtual Function and Abstract Class

```
#include <iostream>

// Abstract base class
class Shape {
    public:
        // Pure virtual function
        virtual void draw() const = 0;
};

// Derived class
class Circle : public Shape {
    public:
        // Override the pure virtual function
        void draw() const override {
            std::cout << "Drawing a circle." << std::endl;
        }
};

// Derived class
class Rectangle : public Shape {
    public:
        // Override the pure virtual function
        void draw() const override {
            std::cout << "Drawing a rectangle." << std::endl;
        }
};

int main() {
```

```
    // Shape shape;  // Error: Cannot instantiate an abstract class
    Shape* shape1 = new Circle();     // Base class pointer pointing to a
    ↪  Circle object
    Shape* shape2 = new Rectangle();  // Base class pointer pointing to a
    ↪  Rectangle object

    shape1->draw();  // Output: Drawing a circle.
    shape2->draw();  // Output: Drawing a rectangle.

    delete shape1;
    delete shape2;

    return 0;
}
```

In this example:

- The `Shape` class is an abstract class because it contains a pure virtual function `draw`.
- The `Circle` and `Rectangle` classes override the `draw` function to provide specific implementations.
- You cannot create an object of the `Shape` class directly, but you can use pointers or references to achieve polymorphic behavior.

3.2.5 Dynamic Binding (Late Binding)

- **Definition:** Dynamic binding (or late binding) is the mechanism by which the function to be executed is determined at runtime based on the actual object type, rather than at compile time based on the pointer or reference type.
- **Use Case:** Achieved through virtual functions, enabling runtime polymorphism.

Example: Dynamic Binding

```cpp
#include <iostream>

// Base class
class Animal {
    public:
        // Virtual function
        virtual void speak() const {
            std::cout << "Animal speaks." << std::endl;
        }
};

// Derived class
class Dog : public Animal {
    public:
        // Override the virtual function
        void speak() const override {
            std::cout << "Dog barks." << std::endl;
        }
};

// Derived class
class Cat : public Animal {
    public:
        // Override the virtual function
        void speak() const override {
            std::cout << "Cat meows." << std::endl;
        }
};

void makeAnimalSpeak(const Animal& animal) {
    animal.speak();  // Dynamic binding: calls the appropriate speak()
    ↪ function
```

```
}

int main() {
    Dog dog;
    Cat cat;

    makeAnimalSpeak(dog);  // Output: Dog barks.
    makeAnimalSpeak(cat);  // Output: Cat meows.

    return 0;
}
```

In this example:

- The `makeAnimalSpeak` function takes a reference to an `Animal` object.
- When `speak` is called, the actual function executed depends on the type of the object passed (`Dog` or `Cat`), demonstrating dynamic binding.

3.2.6 Summary

- **Virtual Functions:** Enable runtime polymorphism by allowing derived classes to override base class functions.
- **Overriding:** Derived classes provide specific implementations of virtual functions.
- **Pure Virtual Functions:** Functions with no implementation in the base class, making the class abstract.
- **Dynamic Binding:** The function to be executed is determined at runtime based on the actual object type.

By mastering virtual functions and overriding, you can create flexible and extensible C++ programs that leverage the power of runtime polymorphism.

3.3 Abstract Classes and Interfaces

3.3.1 Introduction to Abstract Classes and Interfaces

Abstract classes and interfaces are powerful concepts in C++ that allow you to define **blueprints** for derived classes. They enable you to specify what a class should do without providing a complete implementation. This promotes code reusability, modularity, and flexibility in your designs.

This section will explain abstract classes, interfaces, and how they are used in C++. By the end of this section, you will understand how to define and use abstract classes and interfaces to create robust and extensible class hierarchies.

3.3.2 Abstract Classes

- **Definition:** An abstract class is a class that cannot be instantiated directly. It is meant to serve as a base class for other classes. An abstract class typically contains one or more **pure virtual functions**.
- **Use Case:** Used to define a common interface or behavior that derived classes must implement.

Pure Virtual Functions

A pure virtual function is a virtual function that has no implementation in the base class. It is declared using `= 0` in the function declaration. A class containing at least one pure virtual function becomes an abstract class.

Syntax for Declaring a Pure Virtual Function:

```
virtual ReturnType FunctionName(Parameters) = 0;
```

Example: Abstract Class

```
#include <iostream>

// Abstract base class
class Shape {
    public:
        // Pure virtual function
        virtual void draw() const = 0;

        // Non-pure virtual function
        virtual void printArea() const {
            std::cout << "Area not defined for this shape." << std::endl;
        }
};

// Derived class
class Circle : public Shape {
    private:
        double radius;

    public:
        Circle(double r) : radius(r) {}

        // Override the pure virtual function
        void draw() const override {
            std::cout << "Drawing a circle with radius " << radius << "."
            ↪  << std::endl;
```

```
        }

        // Override the non-pure virtual function
        void printArea() const override {
            std::cout << "Area of circle: " << 3.14159 * radius * radius
            ↪  << std::endl;
        }
};

// Derived class
class Rectangle : public Shape {
    private:
        double width, height;

    public:
        Rectangle(double w, double h) : width(w), height(h) {}

        // Override the pure virtual function
        void draw() const override {
            std::cout << "Drawing a rectangle with width " << width << "
            ↪  and height " << height << "." << std::endl;
        }

        // Override the non-pure virtual function
        void printArea() const override {
            std::cout << "Area of rectangle: " << width * height <<
            ↪  std::endl;
        }
};

int main() {
    // Shape shape;  // Error: Cannot instantiate an abstract class
```

```
    Shape* shape1 = new Circle(5.0);      // Base class pointer pointing
    ↪ to a Circle object
    Shape* shape2 = new Rectangle(4.0, 6.0);  // Base class pointer
    ↪ pointing to a Rectangle object

    shape1->draw();      // Output: Drawing a circle with radius 5.
    shape1->printArea(); // Output: Area of circle: 78.5397

    shape2->draw();      // Output: Drawing a rectangle with width 4 and
    ↪ height 6.
    shape2->printArea(); // Output: Area of rectangle: 24

    delete shape1;
    delete shape2;

    return 0;
}
```

In this example:

- The `Shape` class is an abstract class because it contains a pure virtual function `draw`.
- The `Circle` and `Rectangle` classes override the `draw` and `printArea` functions to provide specific implementations.
- You cannot create an object of the `Shape` class directly, but you can use pointers or references to achieve polymorphic behavior.

3.3.3 Interfaces

- **Definition:** An interface is a class that contains only pure virtual functions and no data members. It defines a contract that derived classes must implement.

- **Use Case:** Used to define a set of methods that must be implemented by any class that adheres to the interface.

Example: Interface

```
#include <iostream>

// Interface
class Drawable {
    public:
        // Pure virtual function
        virtual void draw() const = 0;
};

// Derived class implementing the interface
class Circle : public Drawable {
    private:
        double radius;

    public:
        Circle(double r) : radius(r) {}

        // Implement the pure virtual function
        void draw() const override {
            std::cout << "Drawing a circle with radius " << radius << "."
            ↪  << std::endl;
        }
};

// Derived class implementing the interface
class Rectangle : public Drawable {
    private:
```

```
        double width, height;

    public:
        Rectangle(double w, double h) : width(w), height(h) {}

        // Implement the pure virtual function
        void draw() const override {
            std::cout << "Drawing a rectangle with width " << width << "
            ↪  and height " << height << "." << std::endl;
        }
};

int main() {
    Drawable* drawable1 = new Circle(5.0);      // Interface pointer
    ↪  pointing to a Circle object
    Drawable* drawable2 = new Rectangle(4.0, 6.0);  // Interface pointer
    ↪  pointing to a Rectangle object

    drawable1->draw();  // Output: Drawing a circle with radius 5.
    drawable2->draw();  // Output: Drawing a rectangle with width 4 and
    ↪  height 6.

    delete drawable1;
    delete drawable2;

    return 0;
}
```

In this example:

- The `Drawable` class is an interface because it contains only a pure virtual function `draw`.

- The `Circle` and `Rectangle` classes implement the `Drawable` interface by providing specific implementations of the `draw` function.

3.3.4 Abstract Classes vs. Interfaces

Feature	Abstract Class	Interface
Definition	Can contain both pure and non-pure virtual functions, as well as data members.	Contains only pure virtual functions and no data members.
Instantiation	Cannot be instantiated directly.	Cannot be instantiated directly.
Use Case	Used to provide a partial implementation and define a common interface for derived classes.	Used to define a contract that derived classes must implement.
Inheritance	Supports single and multiple inheritance.	Supports multiple inheritance.

3.3.5 Best Practices for Abstract Classes and Interfaces

1. **Use Abstract Classes for Partial Implementation:**

    - Use abstract classes when you want to provide a common implementation for some methods while leaving others to be implemented by derived classes.

2. **Use Interfaces for Defining Contracts:**

    - Use interfaces when you want to define a set of methods that must be implemented by any class that adheres to the interface.

3. **Follow the Liskov Substitution Principle:**

    - Ensure that derived classes can be used interchangeably with their base classes without altering the correctness of the program.

4. **Avoid Multiple Inheritance of State:**

    - Prefer using interfaces for multiple inheritance to avoid the complexities and ambiguities associated with inheriting state from multiple base classes.

3.3.6 Summary

- **Abstract Classes:** Classes that cannot be instantiated directly and contain at least one pure virtual function. They can provide partial implementations and define a common interface for derived classes.
- **Interfaces:** Classes that contain only pure virtual functions and no data members. They define a contract that derived classes must implement.
- **Use Cases:** Abstract classes are used for partial implementation and common interfaces, while interfaces are used for defining contracts.
- **Best Practices:** Use abstract classes and interfaces to promote code reusability, modularity, and flexibility.

By mastering abstract classes and interfaces, you can design robust and extensible class hierarchies that adhere to the principles of Object-Oriented Programming.

Chapter 4

Advanced OOP Concepts:

4.1 Multiple Inheritance and Virtual Base Classes

4.1.1 Introduction to Multiple Inheritance

Multiple inheritance is a feature in C++ that allows a class to inherit from more than one base class. This enables a derived class to combine the properties and behaviors of multiple base classes. While powerful, multiple inheritance can introduce complexities such as ambiguity and the ”diamond problem.” Virtual base classes are used to resolve some of these issues.
This section will explain multiple inheritance, the diamond problem, and how virtual base classes can be used to address these challenges. By the end of this section, you will understand how to use multiple inheritance effectively in your C++ programs.

4.1.2 Multiple Inheritance

- **Definition:** Multiple inheritance allows a derived class to inherit from two or more base classes. The derived class combines the members (variables and functions) of all its base

classes.

- **Use Case:** Used when a class needs to exhibit the behavior of multiple unrelated classes.

Syntax for Multiple Inheritance:

```
class DerivedClass : access-specifier BaseClass1, access-specifier
↪  BaseClass2, ... {
    // Derived class members
};
```

Example: Multiple Inheritance

```
#include <iostream>

// Base class 1
class Animal {
    public:
        void eat() {
            std::cout << "Animal is eating." << std::endl;
        }
};

// Base class 2
class Bird {
    public:
        void fly() {
            std::cout << "Bird is flying." << std::endl;
        }
};

// Derived class inheriting from both Animal and Bird
```

```
class Bat : public Animal, public Bird {
    public:
        void display() {
            std::cout << "Bat is a mammal that can fly." << std::endl;
        }
};

int main() {
    Bat bat;
    bat.eat();     // Output: Animal is eating.
    bat.fly();     // Output: Bird is flying.
    bat.display(); // Output: Bat is a mammal that can fly.
    return 0;
}
```

In this example:

- The `Bat` class inherits from both the `Animal` and `Bird` classes.
- It can access the `eat` function from `Animal` and the `fly` function from `Bird`.

4.1.3 The Diamond Problem

- **Definition:** The diamond problem occurs in multiple inheritance when a derived class inherits from two or more classes that themselves inherit from a common base class. This creates ambiguity in the derived class about which path to take to access the common base class members.
- **Use Case:** A common scenario in complex class hierarchies where multiple inheritance is used.

Example: Diamond Problem

```cpp
#include <iostream>

// Common base class
class Animal {
    public:
        void eat() {
            std::cout << "Animal is eating." << std::endl;
        }
};

// Derived class 1
class Mammal : public Animal {
    public:
        void breathe() {
            std::cout << "Mammal is breathing." << std::endl;
        }
};

// Derived class 2
class WingedAnimal : public Animal {
    public:
        void fly() {
            std::cout << "WingedAnimal is flying." << std::endl;
        }
};

// Derived class inheriting from both Mammal and WingedAnimal
class Bat : public Mammal, public WingedAnimal {
    public:
        void display() {
            std::cout << "Bat is a mammal that can fly." << std::endl;
        }
```

```
};

int main() {
    Bat bat;
    bat.breathe();  // Output: Mammal is breathing.
    bat.fly();      // Output: WingedAnimal is flying.
    // bat.eat();   // Error: Ambiguous access to 'eat' (from Animal)
    return 0;
}
```

In this example:

- The `Bat` class inherits from both `Mammal` and `WingedAnimal`, which both inherit from `Animal`.
- This creates ambiguity when trying to access the `eat` function from `Animal`, as there are two paths to reach it (`Bat -> Mammal -> Animal` and `Bat -> WingedAnimal -> Animal`).

4.1.4 Virtual Base Classes

- **Definition:** A virtual base class is used to resolve the diamond problem by ensuring that only one instance of the common base class is inherited by the derived class, regardless of how many paths exist in the inheritance hierarchy.
- **Use Case:** Used to prevent duplication of the common base class and resolve ambiguity in multiple inheritance.

Syntax for Virtual Base Class:

```
class BaseClass {
    // Base class members
};

class DerivedClass1 : virtual public BaseClass {
    // Derived class 1 members
};

class DerivedClass2 : virtual public BaseClass {
    // Derived class 2 members
};

class FinalDerivedClass : public DerivedClass1, public DerivedClass2 {
    // Final derived class members
};
```

Example: Virtual Base Class

```
#include <iostream>

// Common base class
class Animal {
    public:
        void eat() {
            std::cout << "Animal is eating." << std::endl;
        }
};

// Derived class 1 with virtual inheritance
class Mammal : virtual public Animal {
    public:
        void breathe() {
```

```
            std::cout << "Mammal is breathing." << std::endl;
        }
};

// Derived class 2 with virtual inheritance
class WingedAnimal : virtual public Animal {
    public:
        void fly() {
            std::cout << "WingedAnimal is flying." << std::endl;
        }
};

// Final derived class inheriting from both Mammal and WingedAnimal
class Bat : public Mammal, public WingedAnimal {
    public:
        void display() {
            std::cout << "Bat is a mammal that can fly." << std::endl;
        }
};

int main() {
    Bat bat;
    bat.breathe();  // Output: Mammal is breathing.
    bat.fly();      // Output: WingedAnimal is flying.
    bat.eat();      // Output: Animal is eating. (No ambiguity)
    bat.display();  // Output: Bat is a mammal that can fly.
    return 0;
}
```

In this example:

- The `Mammal` and `WingedAnimal` classes use virtual inheritance to inherit from `Animal`.

- This ensures that only one instance of `Animal` is included in the `Bat` class, resolving the ambiguity in accessing the `eat` function.

4.1.5 Best Practices for Multiple Inheritance

1. **Use Multiple Inheritance Sparingly:**
    - Multiple inheritance can lead to complex and hard-to-maintain code. Use it only when necessary and when it provides clear benefits.
2. **Prefer Composition Over Multiple Inheritance:**
    - In many cases, composition (using member variables of other classes) is a better alternative to multiple inheritance.
3. **Use Virtual Base Classes to Resolve Ambiguity:**
    - When dealing with the diamond problem, use virtual base classes to ensure that only one instance of the common base class is inherited.
4. **Follow the Single Responsibility Principle:**
    - Ensure that each class has a single responsibility and that multiple inheritance is used to combine orthogonal functionalities.

4.1.6 Summary

- **Multiple Inheritance:** Allows a class to inherit from more than one base class, combining their properties and behaviors.
- **Diamond Problem:** Occurs when a derived class inherits from two or more classes that share a common base class, leading to ambiguity.

- **Virtual Base Classes:** Resolve the diamond problem by ensuring that only one instance of the common base class is inherited.
- **Best Practices:** Use multiple inheritance sparingly, prefer composition, and use virtual base classes to resolve ambiguity.

By understanding multiple inheritance and virtual base classes, you can design complex class hierarchies that are both powerful and maintainable.

4.2 `override` and `final` Keywords

4.2.1 Introduction to `override` and `final`

In modern C++ (C++11 and later), the `override` and `final` keywords were introduced to improve code clarity, safety, and maintainability. These keywords help developers explicitly express their intentions when working with inheritance and polymorphism, reducing the likelihood of subtle bugs and misunderstandings.
This section will explain the `override` and `final` keywords, their purpose, and how to use them effectively in your C++ programs. By the end of this section, you will understand how these keywords enhance the robustness of your object-oriented designs.

4.2.2 The `override` Keyword

- **Definition:** The `override` keyword is used to explicitly indicate that a member function in a derived class is intended to override a virtual function in the base class.
- **Purpose:** It ensures that the function signature in the derived class matches the virtual function in the base class. If the function does not override a base class function, the compiler will generate an error.

- **Use Case:** Used to prevent accidental mistakes when overriding virtual functions, such as mismatched function signatures.

Syntax for `override`:

```
class DerivedClass : public BaseClass {
    ReturnType FunctionName(Parameters) override {
        // Function implementation
    }
};
```

Example: Using `override`

```
#include <iostream>

// Base class
class Animal {
    public:
        virtual void speak() const {
            std::cout << "Animal speaks." << std::endl;
        }
};

// Derived class
class Dog : public Animal {
    public:
        // Correctly overrides the base class function
        void speak() const override {
            std::cout << "Dog barks." << std::endl;
        }
};
```

```
// Derived class with a mistake
class Cat : public Animal {
    public:
        // Incorrect function signature (missing const)
        void speak() override {  // Error: Does not override any base
        ↪  class function
            std::cout << "Cat meows." << std::endl;
        }
};

int main() {
    Dog dog;
    dog.speak();  // Output: Dog barks.

    // Cat cat;  // Error: 'speak' does not override a base class function
    return 0;
}
```

In this example:

- The `Dog` class correctly overrides the `speak` function from the `Animal` class using the `override` keyword.
- The `Cat` class attempts to override `speak` but has a mismatched function signature (missing `const`). The `override` keyword causes the compiler to generate an error, preventing the mistake.

4.2.3 The `final` Keyword

- **Definition:** The `final` keyword is used to prevent further overriding of a virtual function or further inheritance of a class.

- **Purpose:** It ensures that a function or class cannot be overridden or inherited, respectively, providing control over the class hierarchy.
- **Use Case:** Used to enforce design decisions, such as sealing a class or preventing further modifications to a function.

Syntax for `final`:

1. **For Functions:**

```
virtual ReturnType FunctionName(Parameters) final {
    // Function implementation
}
```

2. **For Classes:**

```
class ClassName final {
    // Class members
};
```

Example: Using `final` for Functions

```
#include <iostream>

// Base class
class Animal {
    public:
        virtual void speak() const {
            std::cout << "Animal speaks." << std::endl;
```

```
        }
};

// Derived class
class Dog : public Animal {
    public:
        // Override and mark as final
        void speak() const final {
            std::cout << "Dog barks." << std::endl;
        }
};

// Further derived class
class Puppy : public Dog {
    public:
        // Attempt to override the final function
        void speak() const override {  // Error: Cannot override 'final'
        ↪  function
            std::cout << "Puppy barks softly." << std::endl;
        }
};

int main() {
    Dog dog;
    dog.speak();  // Output: Dog barks.

    // Puppy puppy;  // Error: 'speak' cannot override 'final' function
    return 0;
}
```

In this example:

- The `Dog` class overrides the `speak` function and marks it as `final`, preventing further

overriding in derived classes.

- The `Puppy` class attempts to override `speak`, but the compiler generates an error because the function is marked as `final`.

Example: Using `final` for Classes

```
#include <iostream>

// Base class
class Animal {
    public:
        virtual void speak() const {
            std::cout << "Animal speaks." << std::endl;
        }
};

// Derived class marked as final
class Dog final : public Animal {
    public:
        void speak() const override {
            std::cout << "Dog barks." << std::endl;
        }
};

// Attempt to inherit from a final class
class Puppy : public Dog {  // Error: Cannot inherit from 'final' class
    public:
        void speak() const override {
            std::cout << "Puppy barks softly." << std::endl;
        }
};
```

```
int main() {
    Dog dog;
    dog.speak();  // Output: Dog barks.

    // Puppy puppy;  // Error: Cannot inherit from 'final' class
    return 0;
}
```

In this example:

- The `Dog` class is marked as `final`, preventing further inheritance.
- The `Puppy` class attempts to inherit from `Dog`, but the compiler generates an error because `Dog` is marked as `final`.

4.2.4 Benefits of `override` and `final`

1. **Improved Code Clarity:**
    - The `override` keyword makes it explicit that a function is intended to override a base class function, improving readability.
    - The `final` keyword clearly indicates that a function or class cannot be further overridden or inherited.
2. **Enhanced Safety:**
    - The `override` keyword prevents accidental mistakes, such as mismatched function signatures, by generating compile-time errors.
    - The `final` keyword enforces design decisions, preventing unintended modifications to the class hierarchy.

3. **Better Maintainability:**

    - By using `override` and `final`, developers can create more robust and maintainable code, reducing the likelihood of bugs and misunderstandings.

4.2.5 Best Practices for `override` and `final`

1. **Always Use `override` for Overriding Functions:**

    - Use the `override` keyword whenever you intend to override a virtual function. This ensures that the function signature matches the base class function and prevents accidental mistakes.

2. **Use `final` to Enforce Design Decisions:**

    - Use the `final` keyword to prevent further overriding of a function or inheritance of a class when you want to enforce specific design constraints.

3. **Avoid Overusing `final`:**

    - While `final` is useful for enforcing design decisions, avoid overusing it, as it can limit the flexibility of your class hierarchy.

4. **Combine `override` and `final` for Clarity:**

    - When marking a function as `final`, also use the `override` keyword to make it clear that the function is overriding a base class function.

Example: Combining `override` and `final`

```
class Animal {
    public:
        virtual void speak() const {
            std::cout << "Animal speaks." << std::endl;
        }
};

class Dog : public Animal {
    public:
        void speak() const override final {  // Override and mark as final
            std::cout << "Dog barks." << std::endl;
        }
};
```

In this example:

- The `speak` function in the `Dog` class is marked with both `override` and `final`, making it clear that it overrides a base class function and cannot be further overridden.

4.2.6 Summary

- **`override`:** Ensures that a function in a derived class correctly overrides a virtual function in the base class, preventing accidental mistakes.
- **`final`:** Prevents further overriding of a function or inheritance of a class, enforcing design decisions.
- **Benefits:** Improved code clarity, enhanced safety, and better maintainability.
- **Best Practices:** Always use `override` for overriding functions, use `final` judiciously, and combine `override` and `final` for clarity.

By mastering the `override` and `final` keywords, you can write more robust, maintainable, and expressive C++ code.

4.3 CRTP (Curiously Recurring Template Pattern)

4.3.1 Introduction to CRTP

The **Curiously Recurring Template Pattern (CRTP)** is an advanced C++ idiom that leverages templates and inheritance to achieve **static polymorphism**. Unlike dynamic polymorphism (achieved through virtual functions), CRTP enables polymorphism at compile time, which can lead to performance improvements and more efficient code.
This section will explain the CRTP, how it works, and its use cases. By the end of this section, you will understand how to use CRTP to implement static polymorphism and other advanced design patterns in C++.

4.3.2 What is CRTP?

- **Definition:** CRTP is a design pattern in which a class template derives from a template specialization of itself. The derived class passes itself as a template argument to the base class.
- **Purpose:** CRTP is used to achieve compile-time polymorphism, avoid the overhead of virtual functions, and enable code reuse through static inheritance.
- **Use Case:** Commonly used in scenarios where you want to provide a common interface or behavior to multiple derived classes without the runtime cost of virtual functions.

Syntax for CRTP:

```
template <typename Derived>
class Base {
    // Base class implementation
};

class Derived : public Base<Derived> {
    // Derived class implementation
};
```

In this pattern:

- The `Base` class is a template that takes a single template parameter (`Derived`).
- The `Derived` class inherits from `Base<Derived>`, effectively passing itself as a template argument to the base class.

4.3.3 How CRTP Works

The key idea behind CRTP is that the base class (`Base`) can access the derived class (`Derived`) through the template parameter. This allows the base class to call derived class methods or access derived class members at compile time, without the need for virtual functions.

Example: Basic CRTP

```
#include <iostream>

// Base class template
template <typename Derived>
class Base {
    public:
        void interface() {
            // Call the derived class implementation
```

```
            static_cast<Derived*>(this)->implementation();
        }

        void implementation() {
            std::cout << "Base implementation." << std::endl;
        }
};

// Derived class
class Derived : public Base<Derived> {
    public:
        void implementation() {
            std::cout << "Derived implementation." << std::endl;
        }
};

int main() {
    Derived d;
    d.interface();  // Output: Derived implementation.
    return 0;
}
```

In this example:

- The `Base` class template defines a method `interface` that calls the `implementation` method of the derived class using `static_cast`.
- The `Derived` class inherits from `Base<Derived>` and provides its own implementation of the `implementation` method.
- When `interface` is called on a `Derived` object, it invokes the `implementation` method of the `Derived` class.

4.3.4 Use Cases of CRTP

CRTP is a powerful pattern with several use cases, including:

1. **Static Polymorphism:**
    - CRTP enables polymorphism at compile time, avoiding the runtime overhead of virtual functions.
2. **Mixin Classes:**
    - CRTP can be used to create mixin classes that add functionality to derived classes without requiring virtual functions.
3. **Code Reuse:**
    - CRTP allows you to reuse common functionality in multiple derived classes while maintaining type safety.
4. **Optimization:**
    - CRTP can be used to optimize performance-critical code by eliminating the need for dynamic dispatch.

4.3.5 Example: CRTP for Static Polymorphism

Problem:
You want to implement a set of classes that perform similar operations but with different implementations, without using virtual functions.

Solution:

Use CRTP to define a common interface in the base class and provide specific implementations in the derived classes.

```
#include <iostream>

// Base class template
template <typename Derived>
class Shape {
    public:
        void draw() {
            static_cast<Derived*>(this)->drawImpl();
        }

        void drawImpl() {
            std::cout << "Base shape drawing." << std::endl;
        }
};

// Derived class 1
class Circle : public Shape<Circle> {
    public:
        void drawImpl() {
            std::cout << "Drawing a circle." << std::endl;
        }
};

// Derived class 2
class Square : public Shape<Square> {
    public:
        void drawImpl() {
            std::cout << "Drawing a square." << std::endl;
        }
};
```

```
int main() {
    Circle circle;
    Square square;

    circle.draw();  // Output: Drawing a circle.
    square.draw();  // Output: Drawing a square.

    return 0;
}
```

In this example:

- The `Shape` class template defines a common `draw` method that calls the `drawImpl` method of the derived class.
- The `Circle` and `Square` classes inherit from `Shape` and provide their own implementations of `drawImpl`.
- When `draw` is called on a `Circle` or `Square` object, it invokes the appropriate `drawImpl` method at compile time.

4.3.6 Example: CRTP for Mixin Classes

Problem:
You want to add common functionality (e.g., logging) to multiple classes without duplicating code.

Solution:
Use CRTP to create a mixin class that provides the common functionality.

```cpp
#include <iostream>

// Mixin class template
template <typename Derived>
class Logger {
    public:
        void log(const std::string& message) {
            std::cout << "Log: " << message << std::endl;
            static_cast<Derived*>(this)->performAction();
        }
};

// Derived class 1
class TaskA : public Logger<TaskA> {
    public:
        void performAction() {
            std::cout << "Performing Task A." << std::endl;
        }
};

// Derived class 2
class TaskB : public Logger<TaskB> {
    public:
        void performAction() {
            std::cout << "Performing Task B." << std::endl;
        }
};

int main() {
    TaskA taskA;
    TaskB taskB;
```

```
    taskA.log("Starting Task A");  // Output: Log: Starting Task A \n
    ↪  Performing Task A.
    taskB.log("Starting Task B");  // Output: Log: Starting Task B \n
    ↪  Performing Task B.

    return 0;
}
```

In this example:

- The `Logger` class template provides a `log` method that logs a message and calls the `performAction` method of the derived class.
- The `TaskA` and `TaskB` classes inherit from `Logger` and provide their own implementations of `performAction`.
- The `log` method adds logging functionality to both `TaskA` and `TaskB` without duplicating code.

4.3.7 Advantages of CRTP

1. **Performance:**
    - CRTP avoids the overhead of virtual function calls, making it suitable for performance-critical code.
2. **Type Safety:**
    - CRTP ensures that the correct derived class methods are called at compile time, reducing the risk of runtime errors.
3. **Code Reuse:**

- CRTP allows you to reuse common functionality across multiple derived classes without duplicating code.

4. **Flexibility:**

   - CRTP can be used to implement a wide range of design patterns, including static polymorphism, mixins, and more.

4.3.8 Limitations of CRTP

1. **Complexity:**

   - CRTP can make the code more complex and harder to understand, especially for developers unfamiliar with the pattern.

2. **Limited to Compile Time:**

   - CRTP is limited to compile-time polymorphism and cannot be used for runtime polymorphism.

3. **Tight Coupling:**

   - CRTP creates a tight coupling between the base and derived classes, which can make the code less flexible.

4.3.9 Summary

- **CRTP:** A design pattern where a class template derives from a template specialization of itself, enabling static polymorphism.
- **Use Cases:** Static polymorphism, mixin classes, code reuse, and performance optimization.

- **Advantages:** Improved performance, type safety, code reuse, and flexibility.
- **Limitations:** Increased complexity, limited to compile time, and tight coupling.

By mastering CRTP, you can implement advanced design patterns and optimize your C++ code for performance and maintainability.

Chapter 5

Practical Examples

5.1 Designing a Class Hierarchy (e.g., Shapes, Vehicles)

5.1.1 Introduction to Class Hierarchies

Designing a class hierarchy is a fundamental aspect of Object-Oriented Programming (OOP). A well-designed hierarchy promotes code reuse, modularity, and extensibility. In this section, we will explore how to design a class hierarchy using two practical examples: **shapes** and **vehicles**. These examples will demonstrate how to use inheritance, polymorphism, and abstraction to create flexible and maintainable class structures.

5.1.2 Designing a Shape Hierarchy

Problem:
You want to create a hierarchy of geometric shapes, where each shape can calculate its area and perimeter. The hierarchy should support common shapes like circles, rectangles, and triangles, and allow for easy extension to new shapes.

Solution:
Design a base class `Shape` that defines common interfaces for calculating area and perimeter. Derive specific shape classes (e.g., `Circle`, `Rectangle`, `Triangle`) from the base class and override the methods to provide specific implementations.

Step-by-Step Design:

1. **Define the Base Class (`Shape`):**
    - The base class will declare pure virtual functions for calculating area and perimeter, making it an abstract class.
    - This ensures that all derived classes must implement these methods.
2. **Derive Specific Shape Classes:**
    - Create derived classes for specific shapes (e.g., `Circle`, `Rectangle`, `Triangle`).
    - Override the pure virtual functions to provide specific implementations.
3. **Use Polymorphism:**
    - Use pointers or references to the base class to achieve polymorphic behavior, allowing you to work with different shapes interchangeably.

Example: Shape Hierarchy

```
#include <iostream>
#include <cmath>
#include <vector>
```

```cpp
// Base class
class Shape {
    public:
        // Pure virtual functions
        virtual double area() const = 0;
        virtual double perimeter() const = 0;

        // Virtual destructor
        virtual ~Shape() = default;

        // Virtual function for displaying shape type
        virtual void displayType() const {
            std::cout << "This is a generic shape." << std::endl;
        }
};

// Derived class: Circle
class Circle : public Shape {
    private:
        double radius;

    public:
        Circle(double r) : radius(r) {}

        double area() const override {
            return 3.14159 * radius * radius;
        }

        double perimeter() const override {
            return 2 * 3.14159 * radius;
        }
```

```
        void displayType() const override {
            std::cout << "This is a circle." << std::endl;
        }
};

// Derived class: Rectangle
class Rectangle : public Shape {
    private:
        double width, height;

    public:
        Rectangle(double w, double h) : width(w), height(h) {}

        double area() const override {
            return width * height;
        }

        double perimeter() const override {
            return 2 * (width + height);
        }

        void displayType() const override {
            std::cout << "This is a rectangle." << std::endl;
        }
};

// Derived class: Triangle
class Triangle : public Shape {
    private:
        double side1, side2, side3;

    public:
```

```
        Triangle(double s1, double s2, double s3) : side1(s1), side2(s2),
        ↪  side3(s3) {}

        double area() const override {
            // Using Heron's formula
            double s = (side1 + side2 + side3) / 2;
            return std::sqrt(s * (s - side1) * (s - side2) * (s - side3));
        }

        double perimeter() const override {
            return side1 + side2 + side3;
        }

        void displayType() const override {
            std::cout << "This is a triangle." << std::endl;
        }
};

int main() {
    std::vector<Shape*> shapes = {
        new Circle(5.0),
        new Rectangle(4.0, 6.0),
        new Triangle(3.0, 4.0, 5.0)
    };

    for (Shape* shape : shapes) {
        shape->displayType();
        std::cout << "Area: " << shape->area() << ", Perimeter: " <<
        ↪  shape->perimeter() << std::endl;
        delete shape;
    }
```

```
    return 0;
}
```

In this example:

- The `Shape` class is an abstract base class with pure virtual functions `area` and `perimeter`.
- The `Circle`, `Rectangle`, and `Triangle` classes inherit from `Shape` and provide specific implementations for calculating area and perimeter.
- Polymorphism is used to store different shapes in a vector of `Shape*` and call their methods.

5.1.3 Designing a Vehicle Hierarchy

Problem:
You want to create a hierarchy of vehicles, where each vehicle can start, stop, and display its type. The hierarchy should support different types of vehicles like cars, motorcycles, and trucks, and allow for easy extension to new vehicle types.

Solution:
Design a base class `Vehicle` that defines common interfaces for starting, stopping, and displaying the vehicle type. Derive specific vehicle classes (e.g., `Car`, `Motorcycle`, `Truck`) from the base class and override the methods to provide specific implementations.

Step-by-Step Design:

1. **Define the Base Class (`Vehicle`):**

- The base class will declare virtual functions for starting, stopping, and displaying the vehicle type.
- The `displayType` function can be overridden by derived classes to provide specific behavior.

2. **Derive Specific Vehicle Classes:**
   - Create derived classes for specific vehicles (e.g., `Car`, `Motorcycle`, `Truck`).
   - Override the virtual functions to provide specific implementations.
3. **Use Polymorphism:**
   - Use pointers or references to the base class to achieve polymorphic behavior, allowing you to work with different vehicles interchangeably.

Example: Vehicle Hierarchy

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

// Base class
class Vehicle {
    public:
        // Virtual functions
        virtual void start() const {
            std::cout << "Vehicle started." << std::endl;
        }

        virtual void stop() const {
            std::cout << "Vehicle stopped." << std::endl;
        }
```

```
        virtual void displayType() const {
            std::cout << "This is a generic vehicle." << std::endl;
        }

        // Virtual destructor
        virtual ~Vehicle() = default;
};

// Derived class: Car
class Car : public Vehicle {
    public:
        void displayType() const override {
            std::cout << "This is a car." << std::endl;
        }
};

// Derived class: Motorcycle
class Motorcycle : public Vehicle {
    public:
        void displayType() const override {
            std::cout << "This is a motorcycle." << std::endl;
        }
};

// Derived class: Truck
class Truck : public Vehicle {
    public:
        void displayType() const override {
            std::cout << "This is a truck." << std::endl;
        }
};
```

```
int main() {
    std::vector<Vehicle*> vehicles = {
        new Car(),
        new Motorcycle(),
        new Truck()
    };

    for (Vehicle* vehicle : vehicles) {
        vehicle->start();
        vehicle->displayType();
        vehicle->stop();
        delete vehicle;
    }

    return 0;
}
```

In this example:

- The `Vehicle` class is a base class with virtual functions `start`, `stop`, and `displayType`.
- The `Car`, `Motorcycle`, and `Truck` classes inherit from `Vehicle` and override the `displayType` function to provide specific behavior.
- Polymorphism is used to store different vehicles in a vector of `Vehicle*` and call their methods.

5.1.4 Extending the Shape Hierarchy

To further extend the shape hierarchy, we can add more shapes and additional functionalities, such as calculating the volume for 3D shapes.

Example: Extending the Shape Hierarchy with 3D Shapes

```
#include <iostream>
#include <cmath>
#include <vector>

// Base class
class Shape {
    public:
        // Pure virtual functions
        virtual double area() const = 0;
        virtual double perimeter() const = 0;

        // Virtual destructor
        virtual ~Shape() = default;

        // Virtual function for displaying shape type
        virtual void displayType() const {
            std::cout << "This is a generic shape." << std::endl;
        }
};

// Derived class: Circle
class Circle : public Shape {
    private:
        double radius;

    public:
        Circle(double r) : radius(r) {}

        double area() const override {
            return 3.14159 * radius * radius;
        }
```

```
        double perimeter() const override {
            return 2 * 3.14159 * radius;
        }

        void displayType() const override {
            std::cout << "This is a circle." << std::endl;
        }
};

// Derived class: Rectangle
class Rectangle : public Shape {
    private:
        double width, height;

    public:
        Rectangle(double w, double h) : width(w), height(h) {}

        double area() const override {
            return width * height;
        }

        double perimeter() const override {
            return 2 * (width + height);
        }

        void displayType() const override {
            std::cout << "This is a rectangle." << std::endl;
        }
};

// Derived class: Triangle
```

```
class Triangle : public Shape {
    private:
        double side1, side2, side3;

    public:
        Triangle(double s1, double s2, double s3) : side1(s1), side2(s2),
        ↪  side3(s3) {}

        double area() const override {
            // Using Heron's formula
            double s = (side1 + side2 + side3) / 2;
            return std::sqrt(s * (s - side1) * (s - side2) * (s - side3));
        }

        double perimeter() const override {
            return side1 + side2 + side3;
        }

        void displayType() const override {
            std::cout << "This is a triangle." << std::endl;
        }
};

// Derived class: Sphere (3D shape)
class Sphere : public Shape {
    private:
        double radius;

    public:
        Sphere(double r) : radius(r) {}

        double area() const override {
```

```
            return 4 * 3.14159 * radius * radius;
        }

        double perimeter() const override {
            return 0; // Not applicable for 3D shapes
        }

        double volume() const {
            return (4.0 / 3.0) * 3.14159 * radius * radius * radius;
        }

        void displayType() const override {
            std::cout << "This is a sphere." << std::endl;
        }
};

int main() {
    std::vector<Shape*> shapes = {
        new Circle(5.0),
        new Rectangle(4.0, 6.0),
        new Triangle(3.0, 4.0, 5.0),
        new Sphere(3.0)
    };

    for (Shape* shape : shapes) {
        shape->displayType();
        std::cout << "Area: " << shape->area() << ", Perimeter: " <<
        ↪  shape->perimeter() << std::endl;
        if (Sphere* sphere = dynamic_cast<Sphere*>(shape)) {
            std::cout << "Volume: " << sphere->volume() << std::endl;
        }
        delete shape;
```

```
    }

    return 0;
}
```

In this extended example:

- The `Sphere` class is added as a 3D shape, inheriting from `Shape`.
- The `Sphere` class overrides the `area` and `perimeter` methods and adds a `volume` method.
- The `dynamic_cast` operator is used to check if a shape is a `Sphere` and call the `volume` method.

5.1.5 Best Practices for Designing Class Hierarchies

1. **Favor Composition Over Inheritance:**
    - Use inheritance only when there is a clear ”is-a” relationship between the base and derived classes. Prefer composition for ”has-a” relationships.
2. **Use Abstract Base Classes for Common Interfaces:**
    - Define abstract base classes to enforce a common interface for all derived classes.
3. **Follow the Liskov Substitution Principle:**
    - Ensure that derived classes can be used interchangeably with the base class without altering the correctness of the program.
4. **Use Virtual Destructors:**

- Always declare a virtual destructor in the base class to ensure proper cleanup of derived class objects.

5. **Keep Hierarchies Shallow:**

   - Avoid deep inheritance hierarchies, as they can become complex and hard to maintain.

5.1.6 Summary

- **Class Hierarchy Design:** A well-designed hierarchy promotes code reuse, modularity, and extensibility.
- **Shapes Example:** Demonstrates how to create a hierarchy of geometric shapes with common interfaces for area and perimeter calculations.
- **Vehicles Example:** Demonstrates how to create a hierarchy of vehicles with common interfaces for starting, stopping, and displaying the vehicle type.
- **Extending Hierarchies:** Shows how to extend the shape hierarchy with 3D shapes and additional functionalities.
- **Best Practices:** Favor composition over inheritance, use abstract base classes, follow the Liskov Substitution Principle, and keep hierarchies shallow.

By mastering class hierarchy design, you can create flexible and maintainable C++ programs that are easy to extend and adapt to new requirements.

Chapter 6

Design patterns

6.1 Design Patterns in Modern C++ (e.g., Singleton, Factory, Observer, Strategy, Decorator, Adapter)

6.1.1 Introduction to Design Patterns

Design patterns are reusable solutions to common problems that arise in software design. They provide a template for solving specific design issues and promote best practices in software development. In this section, we will explore some of the most widely used design patterns in Modern C++: **Singleton**, **Factory**, **Observer**, **Strategy**, **Decorator**, and **Adapter**. These patterns will help you write more modular, maintainable, and scalable code.

6.1.2 Singleton Pattern

- **Definition:** The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.

- **Use Case:** Used when exactly one object is needed to coordinate actions across the system, such as a configuration manager, logging service, or connection pool.

Key Features:

1. **Private Constructor:** Prevents external instantiation.
2. **Static Instance:** A static member variable holds the single instance.
3. **Static Method:** A static method provides access to the instance.

Example: Singleton Pattern

```
#include <iostream>
#include <memory>
#include <mutex>

class Singleton {
    private:
        // Private constructor
        Singleton() {
            std::cout << "Singleton instance created." << std::endl;
        }

        // Delete copy constructor and assignment operator
        Singleton(const Singleton&) = delete;
        Singleton& operator=(const Singleton&) = delete;

        // Static instance
        static std::unique_ptr<Singleton> instance;
        static std::mutex mutex;
```

```
    public:
        // Static method to access the instance
        static Singleton& getInstance() {
            std::lock_guard<std::mutex> lock(mutex);
            if (!instance) {
                instance = std::unique_ptr<Singleton>(new Singleton());
            }
            return *instance;
        }

        void doSomething() {
            std::cout << "Doing something." << std::endl;
        }

        // Destructor
        ~Singleton() {
            std::cout << "Singleton instance destroyed." << std::endl;
        }
};

// Initialize static members
std::unique_ptr<Singleton> Singleton::instance = nullptr;
std::mutex Singleton::mutex;

int main() {
    Singleton& singleton = Singleton::getInstance();
    singleton.doSomething();

    // Attempt to create another instance (will return the same instance)
    Singleton& anotherSingleton = Singleton::getInstance();
    anotherSingleton.doSomething();
```

```
    return 0;
}
```

In this example:

- The `Singleton` class has a private constructor, preventing external instantiation.
- The `getInstance` method provides access to the single instance, creating it if it doesn't exist.
- The `mutex` ensures thread safety during instance creation.

6.1.3 Factory Pattern

- **Definition:** The Factory pattern defines an interface for creating objects but lets subclasses alter the type of objects that will be created.
- **Use Case:** Used when a class cannot anticipate the type of objects it needs to create or when the creation logic should be decoupled from the main class.

Key Features:

1. **Factory Interface:** Defines a method for creating objects.
2. **Concrete Factories:** Implement the factory interface to create specific types of objects.
3. **Product Interface:** Defines the interface for the objects created by the factory.

Example: Factory Pattern

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

// Product interface
class Product {
    public:
        virtual void use() = 0;
        virtual ~Product() = default;
};

// Concrete products
class ProductA : public Product {
    public:
        void use() override {
            std::cout << "Using Product A." << std::endl;
        }
};

class ProductB : public Product {
    public:
        void use() override {
            std::cout << "Using Product B." << std::endl;
        }
};

// Factory interface
class Factory {
    public:
        virtual std::unique_ptr<Product> createProduct() = 0;
        virtual ~Factory() = default;
};
```

```
// Concrete factories
class FactoryA : public Factory {
    public:
        std::unique_ptr<Product> createProduct() override {
            return std::make_unique<ProductA>();
        }
};

class FactoryB : public Factory {
    public:
        std::unique_ptr<Product> createProduct() override {
            return std::make_unique<ProductB>();
        }
};

int main() {
    std::unique_ptr<Factory> factoryA = std::make_unique<FactoryA>();
    std::unique_ptr<Product> productA = factoryA->createProduct();
    productA->use();

    std::unique_ptr<Factory> factoryB = std::make_unique<FactoryB>();
    std::unique_ptr<Product> productB = factoryB->createProduct();
    productB->use();

    return 0;
}
```

In this example:

- The `Factory` interface defines a method `createProduct` for creating objects.
- The `FactoryA` and `FactoryB` classes implement the `Factory` interface to create `ProductA` and `ProductB` objects, respectively.

- The `Product` interface defines the `use` method, which is implemented by `ProductA` and `ProductB`.

6.1.4 Observer Pattern

- **Definition:** The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Use Case:** Used when you need to maintain consistency between related objects without making them tightly coupled, such as in event handling systems or MVC architectures.

Key Features:

1. **Subject:** Maintains a list of observers and notifies them of state changes.
2. **Observer:** Defines an interface for objects that should be notified of changes.
3. **Concrete Observers:** Implement the `Observer` interface to react to state changes.

Example: Observer Pattern

```
#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>

// Observer interface
class Observer {
    public:
        virtual void update(const std::string& message) = 0;
```

```
        virtual ~Observer() = default;
};

// Concrete observer
class ConcreteObserver : public Observer {
    private:
        std::string name;

    public:
        ConcreteObserver(const std::string& name) : name(name) {}

        void update(const std::string& message) override {
            std::cout << name << " received message: " << message <<
            ↪  std::endl;
        }
};

// Subject
class Subject {
    private:
        std::vector<std::shared_ptr<Observer>> observers;

    public:
        void attach(std::shared_ptr<Observer> observer) {
            observers.push_back(observer);
        }

        void detach(std::shared_ptr<Observer> observer) {
            observers.erase(std::remove(observers.begin(), observers.end(),
            ↪  observer), observers.end());
        }
```

```
        void notify(const std::string& message) {
            for (const auto& observer : observers) {
                observer->update(message);
            }
        }
};

int main() {
    Subject subject;

    auto observer1 = std::make_shared<ConcreteObserver>("Observer 1");
    auto observer2 = std::make_shared<ConcreteObserver>("Observer 2");

    subject.attach(observer1);
    subject.attach(observer2);

    subject.notify("Hello, Observers!");

    subject.detach(observer1);
    subject.notify("Observer 1 has been detached.");

    return 0;
}
```

In this example:

- The `Subject` class maintains a list of observers and notifies them of state changes.
- The `ConcreteObserver` class implements the `Observer` interface to react to notifications.
- The `attach` and `detach` methods manage the list of observers, and the `notify` method sends updates to all observers.

6.1.5 Strategy Pattern

- **Definition:** The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows the algorithm to vary independently from clients that use it.
- **Use Case:** Used when you need to dynamically change the behavior of an object at runtime, such as in sorting algorithms, compression strategies, or payment methods.

Key Features:

1. **Strategy Interface:** Defines a common interface for all concrete strategies.
2. **Concrete Strategies:** Implement the strategy interface with specific algorithms.
3. **Context:** Maintains a reference to a strategy object and delegates work to it.

Example: Strategy Pattern

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

// Strategy interface
class Strategy {
    public:
        virtual void execute(int a, int b) = 0;
        virtual ~Strategy() = default;
};

// Concrete strategies
class AddStrategy : public Strategy {
    public:
```

```
        void execute(int a, int b) override {
            std::cout << "Addition: " << a + b << std::endl;
        }
};

class SubtractStrategy : public Strategy {
    public:
        void execute(int a, int b) override {
            std::cout << "Subtraction: " << a - b << std::endl;
        }
};

class MultiplyStrategy : public Strategy {
    public:
        void execute(int a, int b) override {
            std::cout << "Multiplication: " << a * b << std::endl;
        }
};

// Context
class Context {
    private:
        std::unique_ptr<Strategy> strategy;

    public:
        void setStrategy(std::unique_ptr<Strategy> newStrategy) {
            strategy = std::move(newStrategy);
        }

        void executeStrategy(int a, int b) {
            if (strategy) {
                strategy->execute(a, b);
```

```
            }
        }
};

int main() {
    Context context;

    context.setStrategy(std::make_unique<AddStrategy>());
    context.executeStrategy(5, 3);  // Output: Addition: 8

    context.setStrategy(std::make_unique<SubtractStrategy>());
    context.executeStrategy(5, 3);  // Output: Subtraction: 2

    context.setStrategy(std::make_unique<MultiplyStrategy>());
    context.executeStrategy(5, 3);  // Output: Multiplication: 15

    return 0;
}
```

In this example:

- The `Strategy` interface defines a method `execute` for performing an operation.
- The `AddStrategy`, `SubtractStrategy`, and `MultiplyStrategy` classes implement the `Strategy` interface with specific algorithms.
- The `Context` class maintains a reference to a strategy object and delegates work to it.

6.1.6 Decorator Pattern

- **Definition:** The Decorator pattern allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.

- **Use Case:** Used when you need to add responsibilities to objects at runtime, such as adding logging, encryption, or compression to data streams.

Key Features:

1. **Component Interface:** Defines the interface for objects that can have responsibilities added to them.
2. **Concrete Component:** Implements the component interface.
3. **Decorator:** Maintains a reference to a component object and defines an interface that conforms to the component's interface.
4. **Concrete Decorators:** Add responsibilities to the component.

Example: Decorator Pattern

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

// Component interface
class Component {
    public:
        virtual void operation() = 0;
        virtual ~Component() = default;
};

// Concrete component
class ConcreteComponent : public Component {
    public:
        void operation() override {
            std::cout << "ConcreteComponent operation." << std::endl;
```

```
        }
};

// Decorator
class Decorator : public Component {
    protected:
        std::unique_ptr<Component> component;

    public:
        Decorator(std::unique_ptr<Component> component) :
        ↪  component(std::move(component)) {}

        void operation() override {
            if (component) {
                component->operation();
            }
        }
};

// Concrete decorators
class ConcreteDecoratorA : public Decorator {
    public:
        ConcreteDecoratorA(std::unique_ptr<Component> component) :
        ↪  Decorator(std::move(component)) {}

        void operation() override {
            Decorator::operation();
            addedBehavior();
        }

        void addedBehavior() {
            std::cout << "Added behavior from ConcreteDecoratorA." <<
            ↪  std::endl;
```

```
        }
};

class ConcreteDecoratorB : public Decorator {
    public:
        ConcreteDecoratorB(std::unique_ptr<Component> component) :
        ↪  Decorator(std::move(component)) {}

        void operation() override {
            Decorator::operation();
            addedBehavior();
        }

        void addedBehavior() {
            std::cout << "Added behavior from ConcreteDecoratorB." <<
            ↪  std::endl;
        }
};

int main() {
    std::unique_ptr<Component> component =
    ↪  std::make_unique<ConcreteComponent>();
    component =
    ↪  std::make_unique<ConcreteDecoratorA>(std::move(component));
    component =
    ↪  std::make_unique<ConcreteDecoratorB>(std::move(component));

    component->operation();

    return 0;
}
```

In this example:

- The `Component` interface defines the `operation` method.
- The `ConcreteComponent` class implements the `Component` interface.
- The `Decorator` class maintains a reference to a `Component` object and delegates work to it.
- The `ConcreteDecoratorA` and `ConcreteDecoratorB` classes add responsibilities to the component.

6.1.7 Adapter Pattern

- **Definition:** The Adapter pattern allows incompatible interfaces to work together by converting the interface of a class into another interface that a client expects.
- **Use Case:** Used when you need to integrate existing classes with new systems or libraries that have different interfaces.

Key Features:

1. **Target Interface:** Defines the interface that the client expects.
2. **Adaptee:** The existing class with an incompatible interface.
3. **Adapter:** Implements the target interface and delegates work to the adaptee.

Example: Adapter Pattern

```
#include <iostream>

// Target interface
class Target {
```

```
    public:
        virtual void request() = 0;
        virtual ~Target() = default;
};

// Adaptee
class Adaptee {
    public:
        void specificRequest() {
            std::cout << "Adaptee's specific request." << std::endl;
        }
};

// Adapter
class Adapter : public Target {
    private:
        Adaptee adaptee;

    public:
        void request() override {
            adaptee.specificRequest();
        }
};

int main() {
    std::unique_ptr<Target> target = std::make_unique<Adapter>();
    target->request();  // Output: Adaptee's specific request.

    return 0;
}
```

In this example:

- The `Target` interface defines the `request` method that the client expects.
- The `Adaptee` class has an incompatible interface with a `specificRequest` method.
- The `Adapter` class implements the `Target` interface and delegates work to the `Adaptee`.

6.1.8 Summary

- **Singleton Pattern:** Ensures a class has only one instance and provides a global point of access to it.
- **Factory Pattern:** Defines an interface for creating objects, allowing subclasses to alter the type of objects created.
- **Observer Pattern:** Defines a one-to-many dependency between objects, ensuring that changes in one object are propagated to its dependents.
- **Strategy Pattern:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **Decorator Pattern:** Allows behavior to be added to individual objects at runtime.
- **Adapter Pattern:** Allows incompatible interfaces to work together by converting the interface of a class into another interface that a client expects.

By mastering these design patterns, you can write more modular, maintainable, and scalable C++ code

Appendices

Appendix A: C++ Standard Library Quick Reference

This appendix provides a concise reference to the most commonly used components of the C++ Standard Library, including containers, algorithms, and utilities.

Contents:

1. **Containers:**
    - Sequence Containers: `std::vector`, `std::list`, `std::deque`, `std::array`, `std::forward_list`.
    - Associative Containers: `std::set`, `std::map`, `std::multiset`, `std::multimap`.
    - Unordered Containers: `std::unordered_set`, `std::unordered_map`, `std::unordered_multiset`, `std::unordered_multimap`.
    - Container Adapters: `std::stack`, `std::queue`, `std::priority_queue`.
2. **Algorithms:**
    - Sorting: `std::sort`, `std::stable_sort`, `std::partial_sort`.

- Searching: `std::find`, `std::binary_search`, `std::lower_bound`, `std::upper_bound`.
- Modifying Operations: `std::copy`, `std::move`, `std::transform`, `std::replace`.
- Numeric Operations: `std::accumulate`, `std::inner_product`, `std::iota`.

3. **Utilities:**

- Smart Pointers: `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`.
- Function Objects: `std::function`, `std::bind`, lambda expressions.
- Time Utilities: `std::chrono`, `std::time`, `std::clock`.

4. **Input/Output:**

- Streams: `std::cin`, `std::cout`, `std::ifstream`, `std::ofstream`.
- Formatting: `std::setw`, `std::setprecision`, `std::fixed`.

Appendix B: Modern C++ Features (C++11 to C++23)

This appendix provides an overview of the key features introduced in Modern C++ (C++11, C++14, C++17, and C++20), along with examples.

Contents:

1. **C++11 Features:**

- Auto Type Deduction: `auto`.
- Range-based For Loops: `for (auto& x : container)`.

- Lambda Expressions: `[capture](params) -> return_type { body }`.
- Smart Pointers: `std::unique_ptr`, `std::shared_ptr`.
- Move Semantics: `std::move`, rvalue references.

2. **C++14 Features:**

- Generalized Lambda Captures: `[x = 42]() { return x; }`.
- Return Type Deduction: `auto function() { return 42; }`.
- Binary Literals: `0b1010`.

3. **C++17 Features:**

- Structured Bindings: `auto [x, y] = std::pair(1, 2)`.
- `std::optional`: `std::optional<int> result = 42`.
- `std::variant`: `std::variant<int, float> v = 3.14f`.
- `std::filesystem`: Filesystem library for path manipulation.

4. **C++20 Features:**

- Concepts: `template <typename T> concept Numeric = std::is_arithmetic_v<T>`.
- Ranges: `std::ranges::sort(container)`.
- Coroutines: `co_await`, `co_yield`.
- `std::span`: Non-owning view over a contiguous sequence.

Appendix C: Common Design Patterns in C++

This appendix provides a detailed explanation of common design patterns, including their structure, use cases, and implementation in C++.

Contents:

1. **Creational Patterns:**
    - Singleton, Factory, Abstract Factory, Builder, Prototype.
2. **Structural Patterns:**
    - Adapter, Decorator, Proxy, Composite, Bridge, Flyweight.
3. **Behavioral Patterns:**
    - Observer, Strategy, Command, Template Method, State, Chain of Responsibility, Iterator, Mediator, Memento, Visitor.
4. **Examples:**
    - Code snippets and UML diagrams for each pattern.

Appendix D: Best Practices for Modern C++

This appendix outlines best practices for writing clean, efficient, and maintainable C++ code.

Contents:

1. **Code Organization:**

- Header Guards: `#pragma once` or `#ifndef`.
- Namespaces: `namespace my_namespace { ... }`.
- Modular Design: Separation of concerns.

2. **Memory Management:**

- Use Smart Pointers: `std::unique_ptr`, `std::shared_ptr`.
- Avoid Raw Pointers: Prefer RAII (Resource Acquisition Is Initialization).
- Rule of Five: Destructor, copy constructor, copy assignment operator, move constructor, move assignment operator.

3. **Performance Optimization:**

- Avoid Unnecessary Copies: Use move semantics.
- Prefer `const` and `constexpr`: Immutable data and compile-time evaluation.
- Use `std::vector` for Dynamic Arrays: Avoid manual memory management.

4. **Error Handling:**

- Use Exceptions for Exceptional Cases: `try`, `catch`, `throw`.
- Avoid Exceptions in Destructors: Use `noexcept`.

5. **Concurrency:**

- Use `std::thread` and `std::async`: Avoid raw threads.
- Synchronization: `std::mutex`, `std::lock_guard`, `std::condition_variable`.

Appendix E: Debugging and Profiling in C++

This appendix provides practical guidance on debugging and profiling C++ programs.

Contents:

1. **Debugging Tools:**
    - GDB (GNU Debugger): Breakpoints, stepping, watchpoints.
    - LLDB: Modern debugger for LLVM.
    - IDE Debuggers: Visual Studio, CLion, Eclipse.
2. **Profiling Tools:**
    - Valgrind: Memory leak detection.
    - gprof: Performance profiling.
    - Perf: Linux performance analysis tool.
3. **Common Debugging Techniques:**
    - Logging: `std::cout`, `std::cerr`.
    - Assertions: `assert(condition)`.
    - Unit Testing: `Google Test`, `Catch2`.
4. **Optimization Techniques:**
    - Inline Functions: `inline`.
    - Loop Unrolling: Manual or compiler optimizations.
    - Cache-Friendly Code: Avoid cache misses.

Appendix F: C++ Coding Standards and Style Guides

This appendix provides guidelines for writing consistent and readable C++ code.

Contents:

1. **Naming Conventions:**
    - Variables: `snake_case` or `camelCase`.
    - Functions: `snake_case` or `camelCase`.
    - Classes: `PascalCase`.
2. **Formatting:**
    - Indentation: 4 spaces or tabs.
    - Braces: K&R style or Allman style.
    - Line Length: 80-120 characters.
3. **Documentation:**
    - Doxygen: `///` for comments.
    - Inline Comments: `//` for brief explanations.
4. **Code Reviews:**
    - Checklist: Memory management, error handling, performance.
    - Tools: `clang-format`, `cppcheck`.

Appendix G: Advanced Topics in C++

This appendix covers advanced topics for readers who want to deepen their understanding of C++.

Contents:

1. **Template Metaprogramming:**
    - Type Traits: `std::is_integral`, `std::is_pointer`.
    - SFINAE: Substitution Failure Is Not An Error.
    - Variadic Templates: `template <typename... Args>`.
2. **Concurrency and Parallelism:**
    - Thread Pools: `std::thread`, `std::async`.
    - Atomic Operations: `std::atomic`.
    - Parallel Algorithms: `std::execution::par`.
3. **Custom Allocators:**
    - `std::allocator`: Custom memory allocation.
    - Memory Pools: Efficient allocation for small objects.
4. **Interfacing with Other Languages:**
    - C Interoperability: `extern "C"`.
    - Python Bindings: `pybind11`.

Appendix H: Resources for Further Learning

This appendix provides a curated list of resources for readers who want to continue their C++ learning journey.

Contents:

1. **Books:**
    - ”Effective Modern C++” by Scott Meyers.
    - ”The C++ Programming Language” by Bjarne Stroustrup.
    - ”C++ Primer” by Stanley B. Lippman.
2. **Online Courses:**
    - Coursera: ”C++ for C Programmers”.
    - Udemy: ”Advanced C++ Programming”.
    - Pluralsight: ”Modern C++ Design Patterns”.
3. **Websites and Blogs:**
    - cppreference.com: Comprehensive C++ reference.
    - Stack Overflow: Q&A for C++ developers.
    - ISO C++ Blog: Updates on C++ standards.
4. **Communities:**
    - Reddit: r/cpp.
    - Discord: C++ Community Server.
    - Meetup: Local C++ user groups.

Appendix I: Glossary of C++ Terms

This appendix provides definitions for key terms and concepts used throughout the book.

Contents:

1. **General Terms:**
    - RAII, OOP, STL, RTTI, SFINAE.
2. **Language Features:**
    - Lambda, Template, Constexpr, Move Semantics, Smart Pointer.
3. **Design Patterns:**
    - Singleton, Factory, Observer, Strategy, Decorator.
4. **Tools and Libraries:**
    - GDB, Valgrind, CMake, Google Test.

Appendix J: Sample Projects and Exercises

This appendix provides sample projects and exercises to help readers practice and apply the concepts covered in the book.

Contents:

1. **Projects:**
    - A Simple Game Engine using OOP and Design Patterns.

- A Multi-threaded Web Server using C++ Concurrency.
- A Custom STL-like Container.

2. **Exercises:**

   - Implement a Singleton Logger.
   - Create a Factory for Geometric Shapes.
   - Write a Decorator for a Text Processing System.

3. **Solutions:**

   - Detailed solutions and explanations for all exercises.

References

Books

Books are an excellent way to gain in-depth knowledge of C++ concepts, best practices, and advanced techniques. Here are some highly recommended books for intermediate learners:

1. **"Effective Modern C++" by Scott Meyers**
    - Focuses on best practices for using C++11 and C++14 features.
    - Covers topics like type deduction, smart pointers, lambda expressions, and concurrency.
2. **"The C++ Programming Language" by Bjarne Stroustrup**
    - Written by the creator of C++, this book provides a comprehensive overview of the language.
    - Covers both basic and advanced topics, including templates, STL, and design patterns.
3. **"C++ Primer" by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo**
    - A thorough introduction to C++ for beginners and intermediate learners.

- Covers modern C++ features, including C++11 and C++14.

4. **"Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides**

    - The classic book on design patterns, often referred to as the "Gang of Four" (GoF) book.
    - Provides detailed explanations of common design patterns and their implementations.

5. **"Modern C++ Design" by Andrei Alexandrescu**

    - Focuses on advanced C++ techniques, including template metaprogramming and policy-based design.
    - A must-read for those interested in pushing the boundaries of C++.

6. **"Concurrency in Action" by Anthony Williams**

    - A comprehensive guide to writing concurrent and parallel programs in C++.
    - Covers threads, mutexes, condition variables, and the C++11/14/17 concurrency features.

Online Courses

Online courses provide interactive learning experiences and are a great way to reinforce the concepts covered in the book. Here are some recommended courses:

1. **"C++ for C Programmers" (Coursera)**

    - A course designed for C programmers transitioning to C++.

- Covers object-oriented programming, STL, and modern C++ features.

2. **"Advanced C++ Programming" (Udemy)**

   - Focuses on advanced C++ topics, including templates, smart pointers, and multithreading.
   - Includes hands-on exercises and projects.

3. **"Modern C++ Design Patterns" (Pluralsight)**

   - A course that delves into design patterns and their implementation in modern C++.
   - Covers creational, structural, and behavioral patterns.

4. **"C++ Fundamentals" (edX)**

   - A beginner-to-intermediate course that covers the basics of C++ and modern features.
   - Includes practical exercises and real-world examples.

Websites and Blogs

Websites and blogs are valuable resources for staying updated with the latest trends, best practices, and community discussions. Here are some recommended sites:

1. **cppreference.com**

   - A comprehensive and up-to-date reference for the C++ Standard Library.
   - Includes detailed documentation for all standard library components.

2. **ISO C++ Blog**

- The official blog of the ISO C++ Standards Committee.
- Provides updates on new standards, proposals, and best practices.

3. **Stack Overflow**

   - A Q&A platform where developers can ask questions and share knowledge.
   - A great resource for troubleshooting and learning from real-world problems.

4. **Herb Sutter's Blog**

   - A blog by Herb Sutter, a prominent C++ expert and member of the ISO C++ Standards Committee.
   - Covers advanced topics, best practices, and updates on C++ standards.

5. **Bartek's Coding Blog**

   - A blog by Bartłomiej Filipek, focusing on modern C++ features, performance, and best practices.
   - Includes tutorials, examples, and in-depth articles.

Communities and Forums

Engaging with the C++ community can provide valuable insights, support, and networking opportunities. Here are some recommended communities:

1. **Reddit: r/cpp**

   - A subreddit dedicated to C++ programming.
   - A place to discuss news, ask questions, and share projects.

2. **Discord: C++ Community Server**

    - A Discord server for C++ developers to chat, collaborate, and share knowledge.
    - Includes channels for beginners, advanced topics, and job postings.

3. **Meetup: Local C++ User Groups**

    - Local meetups and user groups for C++ developers.
    - A great way to network, attend talks, and participate in workshops.

4. **C++ Slack Community**

    - A Slack workspace for C++ developers.
    - Includes channels for discussions, help, and collaboration.

Tools and Libraries

Using the right tools and libraries can significantly enhance productivity and code quality. Here are some recommended tools and libraries for C++ developers:

1. **Compiler and Build Tools:**

    - **GCC (GNU Compiler Collection):** A widely used C++ compiler.
    - **Clang:** A modern C++ compiler with excellent diagnostics and performance.
    - **CMake:** A cross-platform build system for managing C++ projects.

2. **Debugging and Profiling Tools:**

    - **GDB (GNU Debugger):** A powerful debugger for C++ programs.

- **Valgrind:** A tool for memory leak detection and profiling.
- **Perf:** A Linux performance analysis tool.

3. **Testing Frameworks:**

- **Google Test:** A C++ testing framework for unit testing.
- **Catch2:** A modern, header-only testing framework.

4. **Code Formatting and Linting:**

- **clang-format:** A tool for automatically formatting C++ code.
- **cppcheck:** A static analysis tool for detecting errors in C++ code.

5. **Libraries:**

- **Boost:** A collection of peer-reviewed, portable C++ libraries.
- **STL (Standard Template Library):** The standard library for C++.
- **Eigen:** A C++ template library for linear algebra.

Conferences and Events

Attending conferences and events is a great way to learn from experts, network with peers, and stay updated with the latest developments in C++. Here are some notable C++ conferences:

1. **CppCon**

- The largest annual conference for the C++ community.
- Features talks, workshops, and networking opportunities.

2. **Meeting C++**

    - A conference focused on C++ programming and community.
    - Includes talks, panels, and social events.

3. **ACCU Conference**

    - A conference for C++ and software development professionals.
    - Covers a wide range of topics, including C++, design patterns, and software engineering.

4. **C++ Now**

    - A conference focused on advanced C++ topics and techniques.
    - Features in-depth talks and hands-on workshops.

Online Tutorials and Documentation

Online tutorials and documentation provide quick references and step-by-step guides for learning C++. Here are some recommended resources:

1. **LearnCpp.com**

    - A free online tutorial for learning C++ from scratch.
    - Covers basic to advanced topics with examples.

2. **C++ Tutorial for Beginners (GeeksforGeeks)**

    - A comprehensive tutorial series for beginners and intermediate learners.

- Includes examples, quizzes, and practice problems.

3. **C++ Documentation (Microsoft Docs)**

   - Official documentation for C++ from Microsoft.
   - Covers language features, libraries, and tools.

4. **C++ Core Guidelines**

   - A set of guidelines for writing modern C++ code.
   - Maintained by the ISO C++ Standards Committee.

Open Source Projects

Contributing to open source projects is a great way to gain practical experience and improve your C++ skills. Here are some notable C++ open source projects:

1. **LLVM**

   - A collection of modular and reusable compiler and toolchain technologies.
   - A great project for those interested in compilers and language design.

2. **Boost**

   - A set of peer-reviewed C++ libraries.
   - A great way to learn advanced C++ techniques and contribute to the community.

3. **OpenCV**

   - An open source computer vision library.

- A great project for those interested in image processing and machine learning.

4. **Qt**

    - A cross-platform framework for GUI and application development.
    - A great project for those interested in UI/UX and application development.