Lecture Notes: C++ Inheritance, Polymorphism, and Generic Programming

Course Notes

November 6, 2025

Contents

1 Overview & Roadmap

These notes expand upon the provided slide deck, structuring the topics into a logical teaching sequence. We begin with the fundamental mechanics of object-oriented design in C++, move to the core concepts of inheritance and runtime polymorphism, and then contrast this with C++'s powerful compile-time polymorphism (templates). The goal is to understand the "why" and the precise trade-offs of each approach.

1. **Encapsulation & RAII**
    - **Scope:** Reviewing class mechanics (`public/private`), data hiding, and the core C++ idiom of Resource Acquisition Is Initialization (RAII) using constructors and destructors.
    - **Prerequisites:** Basic understanding of C++ `struct` or `class`.
2. **Inheritance**
    - **Scope:** The "is-a" relationship (`class Derived : public Base`), what is (and isn't) inherited, access specifiers (`protected`), and constructor/destructor call order.
    - **Prerequisites:** Encapsulation & RAII.
3. **Runtime Polymorphism**
    - **Scope:** Dynamic dispatch using `virtual` functions, base-class pointers/references, vtables, `override`, `final`, and the critical importance of virtual destructors.
    - **Prerequisites:** Inheritance.
4. **Abstract Base Classes (ABCs)**
    - **Scope:** Defining interfaces with pure virtual functions (`= 0`), creating abstract classes that cannot be instantiated, and their canonical use with smart pointers.
    - **Prerequisites:** Runtime Polymorphism.
5. **Compile-Time Polymorphism (Templates)**
    - **Scope:** Function and class templates as a mechanism for generic programming, a.k.a. compile-time polymorphism.
    - **Prerequisites:** Basic C++ function/class syntax.
6. **Paradigms in Practice I: The `Stack`**
    - **Scope:** Implementing a `Stack` data structure using four different programming paradigms (Procedural, Modular, OOP, Generic) to compare their trade-offs, safety, and reusability.
    - **Prerequisites:** All preceding topics.

7. **Paradigms in Practice II: Numerical Integration**
    - **Scope:** Implementing a numerical integration algorithm to directly compare the runtime (OOP/virtual) and compile-time (template) polymorphism strategies, focusing on performance and flexibility.
    - **Prerequisites:** All preceding topics.

2 Expanded Topics

2.1 Topic 1: Encapsulation & RAII

Concept

Encapsulation is the bundling of data with the methods that operate on that data. It's a core pillar of Object-Oriented Programming (OOP). In C++, this is achieved with `class` or `struct`, using access specifiers (`public`, `protected`, `private`) to hide implementation details. `private` members can only be accessed by the class's own methods, creating a "public interface" and a "private implementation." This prevents external code from creating invalid state.

RAII (Resource Acquisition Is Initialization) is the most important idiom in C++. It states that resource lifetime (memory, files, locks, network sockets) should be tied to object lifetime. A resource is acquired in the constructor (ctor) and released in the destructor (dtor). This makes resource management automatic and exception-safe, as the destructor is guaranteed to run when an object goes out of scope, whether by normal return or by stack unwinding from an exception.

Syntax Patterns

```cpp
class MyResourceHolder {
private:
    // Data members (implementation details)
    ResourceType* resource_;
    int data_;

public:
    // Constructor (Acquisition)
    explicit MyResourceHolder(int data) : data_(data) {
        resource_ = new ResourceType(); // Acquire resource
    }

    // Destructor (Release)
    ~MyResourceHolder() {
        delete resource_; // Release resource
        // No need to check for null
    }

    // Public interface (methods)
    void do_something() {
        // ... uses resource_ and data_ ...
    }

    // Prevent copying (or implement it correctly)
    MyResourceHolder(const MyResourceHolder&) = delete;
    MyResourceHolder& operator=(const MyResourceHolder&) = delete;
};
```

(Note: Modern C++ prefers `std::unique_ptr` over raw `new/delete`, which we'll see next.)

Minimal Example (RAII with `std::unique_ptr`)

This example shows modern RAII. We don't need a custom destructor because `std::unique_ptr` is itself a RAII object that handles deletion.

```cpp
#include <iostream>
#include <memory> // For std::unique_ptr
#include <string>

// A simple resource
struct Resource {
    std::string name;
    Resource(std::string n) : name(std::move(n)) {
        std::cout << "Acquiring Resource: " << name << "\n";
    }
    ~Resource() {
        std::cout << "Releasing Resource: " << name << "\n";
    }
};

// A class demonstrating RAII and Encapsulation
class Manager {
private:
    std::unique_ptr<Resource> res_; // Data is private
    int id_;

public:
    // Ctor acquires the resource
    Manager(int id, std::string name)
      : res_(std::make_unique<Resource>(std::move(name))), id_(id) {
        std::cout << "Manager " << id_ << " created.\n";
    }

    void Greet() {
        if (res_) {
            std::cout << "Manager " << id_ << " says hello with "
                      << res_->name << "\n";
        }
    }

    // ~Manager() destructor is auto-generated by the compiler.
    // It will automatically call the destructor for res_,
    // which in turn calls delete on the Resource.
};

int main() {
    std::cout << "--- Entering main ---\n";
    Manager m(1, "Res-A");
    m.Greet();
    std::cout << "--- Leaving main ---\n";
    // m goes out of scope here.
    // Dtor ~Manager() is called.
    // ~unique_ptr() is called.
    // ~Resource() is called.
    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
--- Entering main ---
Acquiring Resource: Res-A
Manager 1 created.
Manager 1 says hello with Res-A
--- Leaving main ---
Releasing Resource: Res-A
```

Worked Example (Slide 30: OOP Stack)

This example from the slides demonstrates RAII for manual memory management. This is how `std::vector` is implemented internally.

```cpp
#include <iostream>
#include <stdexcept> // For std::runtime_error

namespace Stack {
class stack {
private:
    double* s_; // Pointer to start of memory (private)
    double* p_; // Pointer to current top (private)
    int n_;     // Max size (private)

public:
    // Ctor: Acquires memory
    explicit stack(int l) : n_(l) {
        s_ = new double[n_];
        p_ = s_;
        std::cout << "Stack created (size " << n_ << ")\n";
    }

    // Dtor: Releases memory
    ~stack() {
        delete[] s_;
        std::cout << "Stack destroyed\n";
    }

    // Rule of 5: For simplicity, delete copy/move
    stack(const stack&) = delete;
    stack& operator=(const stack&) = delete;
    stack(stack&&) = delete;
    stack& operator=(stack&&) = delete;

    void push(double v) {
        if (p_ == s_ + n_) {
            throw std::runtime_error("Stack overflow");
        }
        *p_++ = v;
    }

    double pop() {
        if (p_ == s_) {
            throw std::runtime_error("Stack underflow");
        }
        return *--p_;
    }
};
} // namespace Stack

int main() {
    try {
        Stack::stack s(10); // Ctor called
        s.push(1.1);
        s.push(2.2);
        std::cout << "Popped: " << s.pop() << "\n";
        std::cout << "Popped: " << s.pop() << "\n";
        // s.pop(); // This would throw "Stack underflow"
    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
    // s goes out of scope here, Dtor is called automatically
    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
Stack created (size 10)
Popped: 2.2
Popped: 1.1
Stack destroyed
```

Common Pitfalls & UB

1. **Dangling Reference:** Returning a reference to a `private` data member that is later destroyed.
2. **`protected` Data:** Often a design smell. It breaks encapsulation for derived classes. Prefer `private` data with `protected` *methods* if derived classes need controlled access.
3. **Leaking Resources:** Forgetting to release a resource in the destructor (if not using RAII types like `std::unique_ptr`).
4. **Exception-Unsafe Code:** If resource release is not in a destructor, an exception thrown mid-function will skip the release code, causing a leak.

RAII solves this.

Performance Notes

- **Ctor/Dtor Overhead:** Constructors and destructors are functions. If they are complex (e.g., acquire locks, allocate memory), they have a cost. Trivial ctor/dtor are often inlined and free.
- **Smart Pointers:** `std::unique_ptr` has zero overhead vs. a raw pointer (when optimized). `std::shared_ptr` has overhead: it must be heap-allocated (with `make_shared`) and use atomic operations for its reference count.

Quick Self-Check

Q: What is RAII and what problem does it solve?

A: Resource Acquisition Is Initialization. It solves resource management (memory, files, etc.) by tying resource lifetime to object lifetime, making it automatic and exception-safe.

Q: Why prefer `private` data members?

A: Encapsulation. It prevents external code from directly modifying the object's internal state, allowing the class to maintain its own invariants (rules about its state).

Q: When is a destructor called?

A: When an object goes out of scope, or when `delete` is called on a pointer to it.

References

cppreference: `class`, `RAII`, `std::unique_ptr`

2.2 Topic 2: Inheritance

Concept

Inheritance allows a new class (the **derived** class) to be based on an existing class (the **base** class). The derived class inherits the members (data and functions) of the base class. This models an "is-a" relationship: a `Student` "is-a" `Person`. The goal is code re-use and, more importantly, establishing a type hierarchy that enables polymorphism. With `public` inheritance, the derived class can be used anywhere the base class is expected.

Syntax Patterns

```cpp
class Base {
private:
    int b_data;
protected:
    int prot_data; // Accessible by Base and Derived
public:
    Base(int d) : b_data(d), prot_data(0) {}
    void base_func() { /* ... */ }
};

// Public inheritance: "is-a"
class Derived : public Base {
private:
    int d_data;
public:
    // Derived ctor MUST initialize Base ctor
    Derived(int d1, int d2) : Base(d1), d_data(d2) {}

    void derived_func() {
        // base_func();      // OK (public)
        // prot_data = 1;    // OK (protected)
        // b_data = 1;       // ERROR: b_data is private to Base
    }
};
```

Minimal Example (Slide 7: Person/Student)

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

class Person {
protected:
    std::string name_; // protected: accessible by derived classes

public:
    Person(std::string name) : name_(std::move(name)) {}

    // Non-virtual: We'll fix this in the Polymorphism section
    void eat() const {
        std::cout << name_ << " is eating.\n";
    }

    std::string get_name() const { return name_; }
};

class Student : public Person {
private:
    std::string major_;

public:
    Student(std::string name, std::string major)
      : Person(std::move(name)), major_(std::move(major)) {}

    void study() const {
        // We can access name_ because it is protected
        std::cout << name_ << " is studying " << major_ << ".\n";
    }
};

int main() {
    Student s("Alice", "Computer Science");

    s.study(); // Call method from Derived
    s.eat();   // Call method inherited from Base

    std::cout << "Student's name is: " << s.get_name() << "\n";
    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
Alice is studying Computer Science.
Alice is eating.
Student's name is: Alice
```

Worked Example (Constructor/Destructor Chaining)

This example shows the order of construction and destruction. Base classes are always constructed *before* derived classes, and destructed *after*.

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

struct Base {
    std::string id;
    Base(std::string s) : id(std::move(s)) {
        std::cout << "Base ctor (" << id << ")\n";
    }
    ~Base() {
        std::cout << "Base dtor (" << id << ")\n";
    }
};

struct Derived : public Base {
    std::string id_d;
    Derived(std::string s_b, std::string s_d)
      : Base(std::move(s_b)), id_d(std::move(s_d)) {
        std::cout << "Derived ctor (" << id_d << ")\n";
    }
    ~Derived() {
        std::cout << "Derived dtor (" << id_d << ")\n";
    }
};

int main() {
    std::cout << "--- Creating Derived ---\n";
    Derived d("BasePart", "DerivedPart");
    std::cout << "--- Deleting Derived ---\n";
    return 0;
}
```

```
Build & Output
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
--- Creating Derived ---
Base ctor (BasePart)
Derived ctor (DerivedPart)
--- Deleting Derived ---
Derived dtor (DerivedPart)
Base dtor (BasePart)
```

Common Pitfalls & UB

1. **Object Slicing:** This is the most dangerous pitfall. If you assign a `Derived` object to a `Base` object (by value), the `Derived` part is "sliced off." Only the `Base` subobject is copied. This breaks polymorphism.

```cpp
Derived d;
Base b = d; // SLICING! b is only a Base.

```

2. **What isn't inherited (Slide 9):** Constructors, destructors, assignment operators (`operator=`), and `friend` relationships are not inherited.

3. **Shadowing:** If a `Derived` class defines a function with the *same name* as a `Base` class function but a different signature, it hides *all* `Base` class overloads of that name. This is rarely intended. Use `using Base::func`$_n$*ame; toun — hidethem.*

Performance Notes

- **Object Layout:** A `Derived` object is typically laid out in memory as the `Base` subobject, followed by the `Derived` data members.
- **Ctor/Dtor Chain:** As shown above, constructing a `Derived` object invokes a chain of constructors (Base then Derived). Destructors run in the reverse order (Derived then Base).
- **Zero-Cost (without `virtual`):** Static inheritance has no runtime overhead. A call to `d.base_func()` is a direct function call.

Quick Self-Check

Q: What is object slicing? How do you prevent it?

A: Slicing is when a derived object is copied into a base object, losing its derived-class data and behavior. Prevent it by always using pointers or references (e.g., `Base*`, `Base`, `std::unique_ptr<Base>`) to refer to polymorphic objects.

Q: What is the difference between `private` and `protected`?

A: `private` members are accessible only to the class itself. `protected` members are accessible to the class *and* all classes derived from it.

Q: In what order are constructors and destructors called for a `Derived` object?

A: Constructors: Base, then Derived. Destructors: Derived, then Base.

References

cppreference: `inheritance`, `access specifiers`, `object slicing`

2.3 Topic 3: Runtime Polymorphism

Concept

Runtime polymorphism (or dynamic dispatch) is the ability to use a single interface (a base class pointer or reference) to interact with objects of different derived types. The program determines *at runtime* which specific derived-class function to call. This is the mechanism that enables ‘std::vector<Shape*>‘, where each ‘Shape‘ could be a ‘Circle‘ or a ‘Square‘, and calling ‘draw()‘ on each one invokes the correct, specific function.

This is enabled in C++ by the `virtual` keyword. When a function is marked `virtual` in a base class, the compiler builds a **virtual function table (vtable)** for that class. Each object of a polymorphic class contains a hidden **vpointer** that points to its class's vtable. A call to a virtual function becomes an indirect call: find object's vpointer → find vtable → find function address in vtable → call function.

Syntax Patterns

```cpp
class Base {
public:
    // Mark function as virtual
    virtual void do_something() { /* base implementation */ }

    // CRITICAL: Virtual destructor
    virtual ~Base() = default; // Or {}
};

class Derived : public Base {
public:
    // Mark as override: compiler checks that it IS overriding a base
    virtual func
    void do_something() override { /* derived implementation */ }

    // Virtual destructor is inherited
};

class FinalDerived final : public Derived {
public:
    // Mark as final: no other class can override this
    void do_something() override final { /* ... */ }
};

void polymorphic_call(Base& b) {
    b.do_something(); // Calls Derived::do_something() if b is a
    Derived
}
```

Minimal Example (Slide 11-12: Fixing `Person`)

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

class Person {
protected:
    std::string name_;
public:
    Person(std::string name) : name_(std::move(name)) {}

    // 1. Add virtual keyword to the base function
    virtual void occupation() const {
        std::cout << name_ << " does stuff.\n";
    }

    // 2. Add virtual destructor!
    virtual ~Person() = default;
};

class Student : public Person {
public:
    Student(std::string name) : Person(std::move(name)) {}

    // 3. Use override (optional, but strongly recommended)
    void occupation() const override {
        std::cout << name_ << " studies stuff.\n";
    }
};

class Teacher : public Person {
public:
    Teacher(std::string name) : Person(std::move(name)) {}

    void occupation() const override {
        std::cout << name_ << " teaches stuff.\n";
    }
};

// This function works polymorphically
void what_r_u_doing(const Person& person) {
    person.occupation(); // Dynamic dispatch!
}

int main() {
    Teacher teacher("Alice");
    Student student("Bob");

    what_r_u_doing(teacher); // Passes Teacher as Person&
    what_r_u_doing(student); // Passes Student as Person&
    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
Alice teaches stuff.
Bob studies stuff.
```

Worked Example (Slide 24-26: Penna Model Motivation)

This shows *why* virtuals are needed. A base class algorithm (`simulate`) needs to be customizable by derived classes (`step`).

```cpp
#include <iostream>
#include <memory> // For std::unique_ptr

// Base class defines a "template method" algorithm
class Population {
public:
    void simulate(int years) {
        for (int i = 0; i < years; ++i) {
            // This call must be polymorphic
            step();
        }
    }

    virtual ~Population() = default;

protected:
    // 1. Define the customizable "step" as virtual
    virtual void step() {
        std::cout << "  Base step (aging, breeding...)\n";
    }
};

// Derived class customizes the "step"
class FishingPopulation : public Population {
protected:
    // 2. Override the virtual function
    void step() override {
        Population::step(); // Call base version
        std::cout << "  + Fishing step (removing fish)\n"; // Add new
    behavior
    }
};

int main() {
    std::cout << "--- Base Population Sim ---\n";
    std::unique_ptr<Population> base_pop =
        std::make_unique<Population>();
    base_pop->simulate(2);

    std::cout << "\n--- Fishing Population Sim ---\n";
    std::unique_ptr<Population> fishing_pop =
        std::make_unique<FishingPopulation>();

    // simulate() is called, which calls the *overridden* step()
    fishing_pop->simulate(2);

    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
--- Base Population Sim ---
  Base step (aging, breeding...)
  Base step (aging, breeding...)

--- Fishing Population Sim ---
  Base step (aging, breeding...)
  + Fishing step (removing fish)
  Base step (aging, breeding...)
  + Fishing step (removing fish)
```

Common Pitfalls & UB

1. **UB: Deleting without a Virtual Destructor (CRITICAL):**

```cpp
Base* b = new Derived();
delete b; // If ~Base() is NOT virtual, this is UNDEFINED
    BEHAVIOR.
          // Only ~Base() is called. ~Derived() is not, leaking
    resources.

```

2. **Calling Virtual Functions from Ctor/Dtor:** This is a subtle trap. Inside a `Base` constructor, the object is only a `Base`. The vtable points to `Base` functions. A call to a virtual function from a ctor or dtor will *always* be statically dispatched to the implementation in *that* class, not a derived one.

3. **Slicing:** As mentioned in Topic 2, slicing destroys polymorphism. A `Base b = Derived();` object is not polymorphic; it's just a `Base`.

4. **Forgetting `override`:** If you misspell a function (`ocuppation()`) and don't use `override`, you create a *new* function instead of overriding. The compiler won't warn you. `override` makes this a compile error.

Performance Notes

- **vtable/vptr Overhead:** Every polymorphic object is larger by one pointer (the vptr).

- **Call Overhead:** A virtual call is more expensive than a direct call. It's (at minimum) two pointer indirections and a register setup.
- **Inlining Prevention:** The compiler cannot inline a virtual call (in most cases) because the function to be called is unknown until runtime. This can be a significant performance hit in tight loops.

Quick Self-Check

Q: What is the "Rule of Thumb" for virtual destructors?

A: If a class has *any* virtual functions, it should have a virtual destructor. If a class is intended as a base class, it should have a virtual destructor.

Q: What does `override` do?

A: It's a promise to the compiler that the function is intended to override a virtual function from a base class. If it doesn't, the compiler errors out.

Q: What is dynamic (runtime) dispatch?

A: The process of selecting which function implementation to call (Base vs. Derived) at runtime, based on the *actual* type of the object, via the vtable.

References

cppreference: `virtual`, `override`, `final`, `dynamic dispatch`

2.4 Topic 4: Abstract Base Classes (ABCs)

Concept

An Abstract Base Class (ABC) is a class that cannot be instantiated on its own. It is designed purely to be a **base class** that defines an **interface**. An interface is a contract: it specifies *what* a derived class must be able to do, but not *how*.

This is achieved by one or more **pure virtual functions**. A pure virtual function is a virtual function that is declared but not (usually) defined, and is marked with `= 0`. Any class that inherits from an ABC *must* override all of its pure virtual functions, or it, too, becomes an abstract class.

ABCs are the primary way to build component-based, "pluggable" systems in C++ (Slide 17).

Syntax Patterns

```cpp
// Abstract Base Class (Interface)
class ISimulation { // "I" prefix is a common (but not required)
    convention
public:
    // Pure virtual function
    virtual void run() = 0;

    // Another pure virtual function
    virtual std::string get_name() const = 0;

    // Abstract class still needs a virtual destructor!
    virtual ~ISimulation() = default;
};

// Concrete (non-abstract) Derived Class
class ConcreteSim : public ISimulation {
public:
    // Must implement ALL pure virtual functions
    void run() override {
        // ... implementation ...
    }

    std::string get_name() const override {
        return "ConcreteSim";
    }
};

// ISimulation sim; // ERROR: Cannot instantiate abstract class
ConcreteSim sim;   // OK
ISimulation& ref = sim; // OK
```

Minimal Example (Slide 17: `Simulation`)

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

// The ABC (Interface)
class Simulation {
public:
    virtual void run() = 0; // Pure virtual
    virtual std::string name() const = 0; // Pure virtual
    virtual ~Simulation() = default;
};

// A concrete implementation
class PennaSim : public Simulation {
public:
    void run() override {
        std::cout << "Running Penna model...\n";
    }
    std::string name() const override {
        return "Penna";
    }
};

// Another concrete implementation
class IsingSim : public Simulation {
public:
    void run() override {
        std::cout << "Running Ising model...\n";
    }
    std::string name() const override {
        return "Ising";
    }
};

// A "driver" function that only knows the interface
void perform(Simulation& sim) {
    std::cout << "Performing simulation: " << sim.name() << "\n";
    sim.run();
}

int main() {
    PennaSim p_sim;
    IsingSim i_sim;

    perform(p_sim); // Passes PennaSim as Simulation&
    perform(i_sim); // Passes IsingSim as Simulation&

    // Simulation s; // COMPILE ERROR
    return 0;
}
```

```
Build & Output
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
↪  ./a.out
Performing simulation: Penna
Running Penna model...
Performing simulation: Ising
Running Ising model...
```

Worked Example (Heterogeneous Collection)

This is the canonical use case: managing a collection of *different* objects that all share the same interface. We use `std::vector<std::unique_ptr<Base»` to store them without slicing and to manage their memory automatically (RAII).

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <memory> // For std::unique_ptr

// Interface
class Shape {
public:
    virtual double area() const = 0;
    virtual std::string name() const = 0;
    virtual ~Shape() = default;
};

// Concrete class 1
class Circle : public Shape {
private:
    double radius_;
public:
    Circle(double r) : radius_(r) {}
    double area() const override { return 3.14159 * radius_ * radius_;
     }
    std::string name() const override { return "Circle"; }
};

// Concrete class 2
class Rectangle : public Shape {
private:
    double w_, h_;
public:
    Rectangle(double w, double h) : w_(w), h_(h) {}
    double area() const override { return w_ * h_; }
    std::string name() const override { return "Rectangle"; }
};

int main() {
    // A heterogeneous collection of shapes
    std::vector<std::unique_ptr<Shape>> shapes;

    // Use std::make_unique to create and add objects
    shapes.push_back(std::make_unique<Circle>(10.0));
    shapes.push_back(std::make_unique<Rectangle>(5.0, 10.0));
    shapes.push_back(std::make_unique<Circle>(2.0));

    double total_area = 0.0;

    // Polymorphic loop
    for (const auto& s_ptr : shapes) {
        // s_ptr is a unique_ptr<Shape>
        // s_ptr->area() calls the virtual function
        std::cout << "Shape: " << s_ptr->name()
                  << ", Area: " << s_ptr->area() << "\n";
        total_area += s_ptr->area();
    }

    std::cout << "Total area: " << total_area << "\n";
    // All memory is automatically freed when vector is destroyed
    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪ ./a.out
Shape: Circle, Area: 314.159
Shape: Rectangle, Area: 50
Shape: Circle, Area: 12.5664
Total area: 376.725
```

Common Pitfalls & UB

1. **Forgetting to implement a pure virtual function:** The compiler will stop you from instantiating the derived class, giving an "is abstract" error.
2. **Forgetting the virtual destructor:** Even if an ABC has no data and no ctor, it *must* have a virtual destructor. Otherwise, `delete` on a base pointer is Undefined Behavior.

3. **Interface Bloat:** Defining an interface that is too large ("fat interface"). Clients are forced to implement functions they don't need. This violates the Interface Segregation Principle.

Performance Notes

- **No difference:** The performance characteristics are identical to any other class using virtual functions (Topic 3). The `= 0` is a compile-time concept; the runtime dispatch mechanism (vtable) is the same.

Quick Self-Check

Q: What is a pure virtual function?

A: A virtual function assigned `= 0`. It makes the class abstract and forces derived classes to provide an implementation.

Q: What is the purpose of an ABC?

A: To define an *interface* (a contract) that concrete classes can implement, enabling polymorphism without coupling code to specific implementations.

Q: What is the "canonical" way to store a heterogeneous collection in C++?

A: `std::vector<std::unique_ptr<Base»` or `std::vector<std::shared_ptr<Base»`.

References

cppreference: `abstract class`, `std::unique_ptr`

2.5 Topic 5: Compile-Time Polymorphism (Templates)

Concept

Templates are the cornerstone of generic programming in C++. They are not functions or classes, but rather **blueprints** that the compiler uses to **generate** functions or classes at compile time. This "polymorphism" occurs at *compile time* because a single template, like `std::sort`, can operate on "many forms" (`vector<int>`, `deque<string>`, etc.), but the code for each specific version is generated by the compiler.

This process is called **monomorphization**. When you use `std::vector<int>`, the compiler writes a `vector_int` class. When you use `std::vector<double>`, it writes a *separate* `vector_double` class.

This approach requires no common base class or `virtual` functions. It works via "duck typing": if a type `T` has the required functions/operators (e.g., `operator<` for `std::sort`), it will compile. C++20 **concepts** make this explicit, allowing us to put constraints on `T`.

Syntax Patterns

```cpp
// Function Template
template <typename T>
T add(T a, T b) {
    return a + b; // Requires T to have operator+
}

// Class Template
template <typename T>
class Box {
private:
    T value;
public:
    Box(T v) : value(v) {}
    T get() const { return value; }
};

// C++20 Concept
template <typename T>
concept Integral = std::is_integral_v<T>;

// Constrained Function Template (C++20)
template <Integral T>
T add_integral(T a, T b) {
    return a + b;
}
// Or: template <typename T> requires Integral<T>
// T add_integral(T a, T b) { /* ... */ }
```

Minimal Example (Generic add)

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

// Function template
template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    // Compiler generates add(int, int)
    std::cout << "Int: " << add(5, 10) << "\n";

    // Compiler generates add(double, double)
    std::cout << "Double: " << add(1.5, 2.3) << "\n";

    // Compiler generates add(std::string, std::string)
    std::string s1 = "Hello, ";
    std::string s2 = "world!";
    std::cout << "String: " << add(s1, s2) << "\n";

    return 0;
}
```

```
Build & Output
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
Int: 15
Double: 3.8
String: Hello, world!
```

Worked Example (Slide 31: Generic Stack)

This is the final, most reusable version of the `Stack` from the slides.

```cpp
#include <iostream>
#include <stdexcept>
#include <string>
#include <memory> // For std::allocator

namespace Stack {
template <typename T> // Make the whole class a template
class stack {
private:
    T* s_;
    T* p_;
    int n_;

public:
    explicit stack(int l) : n_(l) {
        // Use allocator to separate allocation from construction
        std::allocator<T> alloc;
        s_ = alloc.allocate(n_);
        p_ = s_; // p_ points to the next free slot
    }

    ~stack() {
        // Must manually destruct elements
        for (T* it = s_; it != p_; ++it) {
            it->~T(); // Call destructor
        }
        // Deallocate memory
        std::allocator<T> alloc;
        alloc.deallocate(s_, n_);
    }

    // Rule of 5: Deleted
    stack(const stack&) = delete;
    stack& operator=(const stack&) = delete;
    stack(stack&&) = delete;
    stack& operator=(stack&&) = delete;

    void push(T v) { // Pass T by value (or T&& and T const&)
        if (p_ == s_ + n_) {
            throw std::runtime_error("Stack overflow");
        }
        // Construct object in-place
        std::construct_at(p_, std::move(v));
        ++p_;
    }

    T pop() {
        if (p_ == s_) {
            throw std::runtime_error("Stack underflow");
        }
        --p_;
        T val = std::move(*p_); // Move value out
        p_->~T(); // Destruct object
        return val;
    }
};
} // namespace Stack

int main() {
    Stack::stack<std::string> s_str(5);
    s_str.push("hello");
    s_str.push("world");
    std::cout << "Popped: " << s_str.pop() << "\n";

    Stack::stack<int> s_int(5);
    s_int.push(100);
    s_int.push(200);
    std::cout << "Popped: " << s_int.pop() << "\n";
    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
Popped: world
Popped: 200
```

(Note: The above implementation is complex! This is why we prefer `std::vector` as shown in Topic 6.)

Common Pitfalls & UB

1. **Error Messages:** Template metaprogramming can produce multi-page, unreadable error messages when a type `T` doesn't match the template's requirements. C++20 **concepts** are the solution.
2. **Code Bloat:** Monomorphization can increase binary size if a large template (e.g., `std::vector`) is instantiated for many different types.
3. **Instantiation-Time Errors:** A template can be syntactically correct, but fail to compile only when instantiated with a specific type.
4. **Definition in Headers:** Template definitions (bodies) must almost always be in the header file, as the compiler needs the full definition to instantiate it. This differs from non-template functions.

Performance Notes

- **Zero-Cost Abstraction:** This is the key benefit. All "polymorphism" is resolved at compile time.
- **Inlining:** The compiler generates a specific function (e.g., `add(int, int)`) and can inline it just like a normal function. This is *much* faster than a vtable call. (See Topic 7).
- **Monomorphization Cost:** The *compile time* can increase as the compiler has to generate all the template code.

Quick Self-Check

Q: What is the difference between runtime and compile-time polymorphism?

A: Runtime (virtuals) uses one function and a vtable to decide at runtime. Compile-time (templates) generates *many* specific functions at compile time.

Q: What is monomorphization?

A: The compiler's process of "stamping out" a concrete class or function (e.g., `stack<int>`) from a generic template (`template <T> class stack`).

Q: What C++20 feature improves template error messages?

A: Concepts. They allow you to name and enforce requirements on template parameters.

References

cppreference: `templates`, `C++20 concepts`

2.6 Paradigms in Practice I: The `Stack`

Concept

This topic (Slides 27-31) uses a single problem—implementing a LIFO Stack—to compare four programming paradigms. This is a powerful pedagogical tool for understanding *why* we use C++ features.

1. **Procedural (C-style):** (Slide 28) Global data, free functions. Dangerous, no encapsulation, no error checking.
2. **Modular:** (Slide 29) Bundles data into a `struct` and functions into a `namespace`. Better, but requires manual `create/destroy` calls. Not exception-safe.
3. **Object-Oriented (RAII):** (Slide 30) A `class` with `private` data and public methods. Ctor/Dtor handle memory (RAII). This is safe, encapsulated, and exception-safe.

4. **Generic (Templates):** (Slide 31) The OOP version is made generic with `template <typename T>`. This is the most reusable, safe, and powerful version.

We already showed the OOP (Topic 1) and Generic (Topic 5) versions. The modern C++20 "best practice" version would not do manual memory management at all, but would *adapt* an existing container, like `std::vector`.

Worked Example (Modern C++20 Generic Stack)

Instead of manual `new/delete`, we implement our `stack` by *wrapping* a `std::vector`. This is the "Adapter" pattern and is how `std::stack` works. It's safer, simpler, and more efficient.

```cpp
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector> // Use vector for backing storage
#include <utility> // For std::move

namespace ModernStack {
template <typename T>
class stack {
private:
    // "Composition over Inheritance"
    // We *contain* a vector to do the work.
    std::vector<T> storage_;

public:
    // No explicit ctor/dtor needed!
    // vector's default ctor/dtor do the work. RAII!

    // C++20: Check if stack is empty
    [[nodiscard]] bool empty() const noexcept {
        return storage_.empty();
    }

    // C++20: Get current size
    [[nodiscard]] size_t size() const noexcept {
        return storage_.size();
    }

    // Add an element (copy)
    void push(const T& value) {
        storage_.push_back(value); // Delegate to vector
    }

    // Add an element (move)
    void push(T&& value) {
        storage_.push_back(std::move(value)); // Delegate to vector
    }

    // Emplace: construct in-place (most efficient)
    template <typename... Args>
    void emplace(Args&&... args) {
        storage_.emplace_back(std::forward<Args>(args)...);
    }

    // Remove an element
    void pop() {
        if (empty()) {
            throw std::runtime_error("Stack underflow");
        }
        storage_.pop_back(); // Delegate to vector
    }

    // Get a reference to the top element
    T& top() {
        if (empty()) {
            throw std::runtime_error("Stack underflow");
        }
        return storage_.back(); // Delegate to vector
    }

    const T& top() const {
        if (empty()) {
            throw std::runtime_error("Stack underflow");
        }
        return storage_.back();
    }
};
} // namespace ModernStack

int main() {
    ModernStack::stack<std::string> s;
    s.push("This");
    s.push("is");
    s.emplace("much"); // Efficiently construct "much"
    s.emplace("safer!");

    while (!s.empty()) {
        std::cout << s.top() << " ";
        s.pop();
    }
    std::cout << "\n";
    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪ ./a.out
safer! much is This
```

Pitfalls & Performance

- **Pitfall (Procedural):** Massive. No error checking, global state, easy to corrupt the stack pointer.
- **Pitfall (Modular):** Forgetting to call `destroy` is a memory leak. Not exception-safe (an exception after `create` but before `destroy` leaks).

- **Pitfall (OOP/Manual):** Correctly implementing the Rule of 5 (copy/move/dtor) is extremely difficult, especially for exception safety.
- **Performance (Vector-based):** `push_back` is *amortized* O(1). When the vector is full, it reallocates (O(N)), which is a large single-frame cost. `pop_back` and `top` are O(1).

References

cppreference: `std::stack`, `std::vector`, `Container adaptors`

2.7 Paradigms in Practice II: Numerical Integration

Concept

This topic (Slides 33-36) is the "grand finale" comparison. We need to pass a function (e.g., $f(x) = x\sin(x)$) to an `integrate` algorithm. How we pass f demonstrates the trade-offs.

1. **Procedural (C-style):** (Slide 34) Pass a **function pointer** (`double (*f)(double)`).
2. **Object-Oriented (Runtime):** (Slide 35) Pass a **base-class reference** (`SimpleFunction f`). `SimpleFunction` is an ABC with a `virtual double operator()(double) const = 0;`. This allows us to select the function at runtime.
3. **Generic (Compile-Time):** (Slide 36) Pass a **template parameter** (`F f`). `F` can be *anything* that is callable, like a function pointer, a lambda, or a struct with `operator()`.

The clear winner for performance is the Generic/Template approach. The compiler knows the exact function f at compile time, so it can **inline** the call to `f(xi)` inside the integration loop, completely eliminating all function call overhead. The OOP/Virtual approach *cannot* be inlined and must pay the vtable-call-cost on *every single iteration* of the loop, which is disastrous for performance in a numerical simulation.

Worked Example (Comparing OOP vs. Generic Integration)

```cpp
#include <iostream>
#include <cmath> // For std::sin
#include <memory> // For std::unique_ptr
#include <iomanip> // For std::setprecision

// --- 1. Object-Oriented (Runtime) Approach ---
namespace OOP {
// The ABC interface
struct SimpleFunction {
    virtual double operator()(double x) const = 0;
    virtual ~SimpleFunction() = default;
};

// The algorithm: takes the interface
```

```cpp
double integrate(const SimpleFunction& f, double a, double b, int N) {
    double dx = (b - a) / N;
    double I = 0.0;
    for (int i = 0; i <= N; ++i) {
        double x = a + i * dx;
        double w = (i == 0 || i == N) ? 0.5 : 1.0;
        I += w * f(x); // VIRTUAL CALL inside loop! (slow)
    }
    return I * dx;
}

// A concrete implementation
struct MyFunc1 : SimpleFunction {
    double operator()(double x) const override { return x * std::sin(x
    ); }
};
} // namespace OOP

// --- 2. Generic (Compile-Time) Approach ---
namespace Generic {
// The algorithm: template
template <typename Function>
double integrate(Function f, double a, double b, int N) {
    double dx = (b - a) / N;
    double I = 0.0;
    for (int i = 0; i <= N; ++i) {
        double x = a + i * dx;
        double w = (i == 0 || i == N) ? 0.5 : 1.0;
        I += w * f(x); // DIRECT INLINED CALL! (fast)
    }
    return I * dx;
}

// A concrete implementation (just a struct, no inheritance)
struct MyFunc1 {
    double operator()(double x) const { return x * std::sin(x); }
};
} // namespace Generic

int main() {
    double a = 0.0, b = 3.14159;
    int N = 10000;

    std::cout << std::fixed << std::setprecision(8);

    // OOP Version
    OOP::MyFunc1 oop_func;
    double res_oop = OOP::integrate(oop_func, a, b, N);
    std::cout << "OOP (Virtual):      " << res_oop << "\n";

    // Generic Version (Functor)
    Generic::MyFunc1 gen_func;
    double res_gen = Generic::integrate(gen_func, a, b, N);
    std::cout << "Generic (Functor): " << res_gen << "\n";

    // Generic Version (Lambda - C++11 and later)
    // The compiler creates a struct for this automatically!
    auto lambda_func = [](double x) { return x * std::sin(x); };
    double res_lambda = Generic::integrate(lambda_func, a, b, N);
    std::cout << "Generic (Lambda):  " << res_lambda << "\n";

    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
OOP (Virtual):      3.14117033
Generic (Functor): 3.14117033
Generic (Lambda):  3.14117033
```

Pitfalls & Performance

- **Performance (OOP):** The virtual call to `f(x)` inside the loop is a "performance pessimization." It blocks inlining and adds vtable overhead to *every* step. This is the wrong tool for this job.
- **Performance (Generic):** This is the "zero-cost abstraction." The call to `f(x)` is fully inlined by the compiler, resulting in code equivalent to `I += w * (x * std::sin(x));`. This is as fast as manually writing the C-style procedural code.
- **Flexibility (OOP):** The OOP version's *only* benefit is if you needed to select the function at runtime (e.g., from a user input) and store it in a `std::unique_ptr<SimpleFunction>`.
- **Flexibility (Generic):** The Generic version is *also* flexible, as it accepts function pointers, functors (structs with `operator()`), and lambdas, all with zero overhead.

References

cppreference: `function pointers`, `lambda expressions`, `functors`

3 Integration Mini-Projects

3.1 Project 1: Polymorphic Shape Manager

Goal: Combine Inheritance, ABCs, Runtime Polymorphism, and RAII (`std::unique_ptr`) to manage a heterogeneous list of shapes.

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <memory> // For std::unique_ptr, std::make_unique
#include <cmath>  // For M_PI (may need -lm)
#include <numeric> // For std::accumulate
#include <iomanip> // For std::setprecision
```

```cpp

// 1. The Interface (ABC)
class Shape {
public:
    virtual ~Shape() = default;
    virtual double area() const = 0;
    virtual std::string name() const = 0;
};

// 2. Concrete Implementations
class Circle : public Shape {
    double r_;
public:
    Circle(double r) : r_(r) {}
    double area() const override { return M_PI * r_ * r_; }
    std::string name() const override { return "Circle"; }
};

class Rectangle : public Shape {
    double w_, h_;
public:
    Rectangle(double w, double h) : w_(w), h_(h) {}
    double area() const override { return w_ * h_; }
    std::string name() const override { return "Rectangle"; }
};

int main() {
    std::cout << std::fixed << std::setprecision(3);

    // 3. The Heterogeneous Collection (RAII)
    std::vector<std::unique_ptr<Shape>> shapes;
    shapes.push_back(std::make_unique<Circle>(5.0));
    shapes.push_back(std::make_unique<Rectangle>(4.0, 6.0));
    shapes.push_back(std::make_unique<Circle>(1.0));

    // 4. Polymorphic Processing
    std::cout << "Processing shapes:\n";
    for (const auto& pShape : shapes) {
        // pShape is const std::unique_ptr<Shape>&
        // pShape->area() is a virtual call
        std::cout << " - A " << pShape->name() << " with area "
                  << pShape->area() << "\n";
    }

    // 5. Using STL Algorithms (C++20 Range-based)
    auto get_area = [](const auto& p) { return p->area(); };
    auto areas = shapes | std::views::transform(get_area);
    double total_area = std::accumulate(areas.begin(), areas.end(),
    0.0);

    // C++17 way (for std::accumulate)
    // double total_area = std::accumulate(shapes.begin(), shapes.end
    (), 0.0,
    //     [](double sum, const auto& p) { return sum + p->area(); });

    std::cout << "Total area: " << total_area << "\n";

    // Memory is freed automatically
    return 0;
}
```

Build & Output

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
Processing shapes:
 - A Circle with area 78.540
 - A Rectangle with area 24.000
 - A Circle with area 3.142
Total area: 105.681
```

Extension Ideas:

- Add a `virtual double perimeter() const = 0;` to the interface and implement it.
- Create a `Triangle` class.
- Write a function `find_largest_shape(const std::vector<...>)` that returns a `const Shape*` to the shape with the biggest area.

3.2 Project 2: Generic Statistics Calculator

Goal: Combine Templates (Compile-Time Polymorphism), `<algorithm>`, and C++20 Ranges to create a statistics utility that works on *any* container of numbers.

```cpp
#include <iostream>
#include <vector>
#include <list>      // To show it works on different containers
#include <numeric>   // For std::accumulate
#include <algorithm> // For std::sort, std::minmax_element
#include <cmath>     // For std::sqrt
#include <stdexcept> // For std::runtime_error
#include <iomanip>   // For std::setprecision

template <typename T>
struct StatsResult {
    T min;
    T max;
    T mean;
    T stddev;
};

// C++20 Version using Ranges
#include <ranges>

template <std::ranges::forward_range R>
// Requires that the element type of R is convertible to double
requires std::convertible_to<std::ranges::range_value_t<R>, double>
auto calculate_stats(R&& range) -> StatsResult<double> {

    // We need a non-destructive copy for sorting, etc.
    // This is a trade-off. We could also just iterate.
    // For this example, let's copy to a vector.
    std::vector<double> data;
    std::ranges::copy(range, std::back_inserter(data));

    if (data.empty()) {
        throw std::runtime_error("Empty range");
    }

    // Min/Max
    const auto [min_it, max_it] = std::ranges::minmax_element(data);

    // Mean
    double sum = std::accumulate(data.begin(), data.end(), 0.0);
    double mean = sum / data.size();

    // StdDev
    double sq_sum = std::accumulate(data.begin(), data.end(), 0.0,
        [mean](double acc, double val) {
            return acc + (val - mean) * (val - mean);
        });
    double stddev = std::sqrt(sq_sum / data.size());

    return {*min_it, *max_it, mean, stddev};
}

// Helper to print stats
template <typename T>
void print_stats(const std::string& title, const StatsResult<T>& stats
    ) {
    std::cout << title << ":\n"
              << "  Min:    " << stats.min << "\n"
              << "  Max:    " << stats.max << "\n"
              << "  Mean:   " << stats.mean << "\n"
              << "  StdDev: " << stats.stddev << "\n";
}

int main() {
    std::cout << std::fixed << std::setprecision(4);

    std::vector<int> v = {1, 5, 10, 3, 7, 20, 8};
    auto stats_v = calculate_stats(v);
    print_stats("std::vector<int>", stats_v);

    std::list<double> l = {1.5, 2.5, 3.5, 4.5, 5.5};
    auto stats_l = calculate_stats(l);
    print_stats("std::list<double>", stats_l);

    // C++20 view: a non-owning range
    auto even_v = v | std::views::filter([](int n){ return n % 2 == 0;
     });
    auto stats_v_even = calculate_stats(even_v);
    print_stats("std::vector<int> (evens)", stats_v_even);

    return 0;
}
```

```
Build & Output
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp &&
 ↪  ./a.out
std::vector<int>:
  Min:    1.0000
  Max:    20.0000
  Mean:   7.7143
  StdDev: 5.8678
std::list<double>:
  Min:    1.5000
  Max:    5.5000
  Mean:   3.5000
  StdDev: 1.4142
std::vector<int> (evens):
  Min:    8.0000
  Max:    20.0000
  Mean:   14.0000
  StdDev: 5.6569
```

Extension Ideas:

- Add median calculation (requires sorting and finding the middle element).
- Make the function work "in-place" without copying, if the range is a `random_access_range` and we are allowed to mutate it (e.g., by sorting).
- Change `StatsResult` to be a template on the `value_type` (e.g., `StatsResult<T>`).

4 Cheat Sheet (C++20)

4.1 Syntax Capsules

Initialization

- `T x(v);` (Direct): Calls constructor.
- `T x = v;` (Copy): Calls constructor (or move-ctor), though copy elision is likely.
- `T x{v};` (Direct-list): **Prefer this.** Calls constructor, disallows "narrowing" conversions.
- `T x = {v};` (Copy-list): As above, also preferred.
- `T x{};` (Value): Zero/default initializes. `int x{};` → `x` is 0.

Value Categories

- **lvalue** (locator): Has an identity/address. Can take its address. (e.g., `int x;`, `x` is an lvalue).
- **prvalue** (pure rvalue): A "temporary." The result of an expression. (e.g., `42`, `x+y`).

- **xvalue** (expiring): An lvalue that can be "moved from." The result of `std::move(x)`.
- **rvalue**: A prvalue or an xvalue. Binds to `T`.
- **lvalue-ref**: `T`. Binds only to lvalues.
- **const lvalue-ref**: `const T`. Binds to lvalues and rvalues.
- **rvalue-ref**: `T`. Binds only to rvalues.

Polymorphism Rules
- Use `virtual` on the base class function.
- Use `override` on all derived class functions.
- Use `final` to prevent further overriding.
- **CRITICAL:** Any base class must have a `public virtual` destructor.

Container Invalidation (Common)
- `std::vector`: `push_back` invalidates all iterators/references if it reallocates. `insert/erase` invalidates all iterators/references at/after the insertion/erasure point.
- `std::deque`: `push_front/back` invalidates iterators, but not references/pointers.
- `std::list`: `insert/erase` only invalidates iterators to the element(s) erased.
- `std::map/std::set`: `insert/erase` only invalidates iterators to the element(s) erased.

4.2 "Do/Don't" Checklist

- **DO** use RAII for all resource management (`std::unique_ptr`, `std::vector`, `std::string`, `std::lock_guard`).
- **DON'T** use raw `new` or `delete`. Let smart pointers and containers do it.
- **DO** prefer `std::unique_ptr` as the default smart pointer. Only use `std::shared_ptr` when shared ownership is *required*.
- **DON'T** use owning raw pointers (e.g., `MyClass* m_ptr;` in a class).
- **DO** pass polymorphic objects by pointer (`Base*`) or reference (`Base`).
- **DON'T** pass polymorphic objects by value (causes slicing).
- **DO** make base class destructors `public` and `virtual`.
- **DON'T** call `virtual` functions from a constructor or destructor.
- **DO** prefer algorithms (`std::ranges::sort`, `std::find_if`) over hand-written loops.

- **DON'T** use `protected` data. Prefer `private` data and `protected` accessors.
- **DO** prefer templates (generic) for algorithms where performance is critical.
- **DO** use `virtual` (OOP) for heterogeneous collections and stable ABIs.
- **DO** mark single-argument constructors `explicit` to prevent implicit conversions.

5 Glossary

ABI (Application Binary Interface) The low-level interface between compiled code modules (e.g., function calling conventions, vtable layout). Changes to a class (e.g., adding a `virtual` function) can break ABI.

ADL (Argument-Dependent Lookup) A C++ rule for finding functions. If you call `func(obj)`, the compiler searches not only the global/current scope, but also the `namespace` where `obj`'s type is defined.

as-if rule The compiler can make any optimization it wants, as long as the observable behavior of the program is "as if" it had executed the code exactly as written.

Copy Elision A compiler optimization that avoids (elides) unnecessary copy/move operations. Mandatory in C++17 for function return values.

Lifetime The period during which an object exists, from the end of its constructor to the end of its destructor. Accessing an object outside its lifetime is UB.

ODR (One Definition Rule) An object or function must have exactly one definition in a program. Violations (e.g., two different definitions of `int foo()` in two `.cpp` files) are linker errors or UB.

RAII (Resource Acquisition Is Initialization) The core C++ idiom of tying resource lifetime to object lifetime (ctor/dtor).

SSO (Small String Optimization) An optimization where `std::string` avoids heap allocation for small strings, storing them inside the object's own 24-32 bytes of storage.

Standard-Layout A class property (a `struct` with no `virtual` functions, all `public` members, etc.) that allows it to be treated like a C `struct` and safely memory-mapped.

Trivial A class property (trivial ctor/dtor/copy/move) that means it can be safely `memcpy`'d.

UB (Undefined Behavior) An action (e.g., deleting via non-virtual dtor, array out-of-bounds, data race) for which the C++ standard imposes *no requirements*. The program may crash, corrupt data, or appear to work.

6 Appendix: Build & Tooling

6.1 Compiler Invocations

All examples in this document are designed to be compiled with a C++20-compliant compiler. We enable all warnings, as you should never ignore compiler warnings.

GCC (g++)

```
g++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp -o my_program
./my_program
```

Clang (clang++)

```
clang++ -std=c++20 -O2 -Wall -Wextra -pedantic file.cpp -o my_program
./my_program
```

Warning Flags Explained

- `-std=c++20`: Enforce the C++20 standard.
- `-O2`: Enable level 2 optimizations. This is important, as some C++ features (like templates, move semantics) are designed to be optimized away.
- `-Wall`: Enable a large, common set of warnings (e.g., unused variables).
- `-Wextra`: Enable *more* warnings not covered by `-Wall`.
- `-pedantic`: Enforce strict ISO C++ compliance, disabling compiler-specific extensions.

6.2 Sanitizers (Debugging)

For debugging, compiling with `-O0 -g` (no optimization, debug symbols) and using sanitizers is invaluable.

- **AddressSanitizer (ASan):** Finds memory errors (out-of-bounds, use-after-free, memory leaks).

  ```
  g++ -std=c++20 -g -fsanitize=address file.cpp && ./a.out
  ```

- **UndefinedBehaviorSanitizer (UBSan):** Finds UB (integer overflow, null pointer dereference, etc.).

```
g++ -std=c++20 -g -fsanitize=undefined file.cpp && ./a.out
```

These tools are your best friends for finding subtle bugs, especially those related to pointers, lifetime, and undefined behavior.

7 Assumptions Made

In expanding the provided slide deck, the following assumptions were made:

- The core topics are **Inheritance**, **Runtime Polymorphism** (`virtual`), and its contrast with **Compile-Time Polymorphism** (templates).
- The "Penna model" slides serve to motivate *why* `virtual` functions are necessary (a base class algorithm needing to call a derived class's customization).
- The "Stack" and "Numerical Integration" examples are pedagogical tools to compare the four programming paradigms (Procedural, Modular, OOP, Generic), highlighting the safety (RAII) and performance (inlining) trade-offs.
- The target audience is familiar with C++ basics (loops, functions, `std::vector`) but needs a rigorous, modern understanding of OOP and generic programming design.
- All code should adhere to modern C++20 best practices (RAII, smart pointers, algorithms, ranges) and explicitly discourage C-style or manual resource management (raw `new/delete`).