C++ Advanced

A Complete 6-Chapter Programming Course

Topics covered:

Multiple & virtual inheritance · Templates, deeper & C++20 concepts

The Rule of Five in full mechanical depth · Concurrency in modern C++

C++-specific undefined behavior · Capstone: a polymorphic shape inventory

Exercises: 18 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · framed against both C and Rust

Course 3 of 3 · completes the full C++ track

Table of Contents

CHAPTER 1 OF 6

Multiple Inheritance & Virtual Inheritance

COURSE 3 · CH 1

Multiple Inheritance & Virtual Inheritance

A capability C never had, and Rust deliberately never built — for good reason

Course 3 goes deeper into C++'s own mechanics, starting with something neither of this course's two comparison languages support in the same form.

Multiple Inheritance

```
class Derived : public Base1, public Base2 {
    // inherits members and methods from BOTH bases directly
};
```

A class can inherit from more than one base class at once — something neither C's plain structs (`c2-1` , no inheritance concept at all) nor Rust's traits support in this exact form.

The Diamond Problem

The classic scenario: class `D` inherits from both `B` and `C` , and both `B` and `C` themselves inherit from a common base `A` . Without intervention, `D` ends up with **two separate copies** of `A` 's data — one via each path — genuinely ambiguous which copy a reference to an inherited `A` member refers to, producing a real compile error unless explicitly disambiguated.

A Worked Example

```
class Animal { public: std::string name; };
class Swimmer : public Animal {};
class Flyer : public Animal {};
class Duck : public Swimmer, public Flyer {};

Duck d;
d.name = "Donald"; // compile error — ambiguous: Swimmer::name or Flyer::name?
```

`virtual` Inheritance — The Fix

```
class Swimmer : virtual public Animal {};
class Flyer : virtual public Animal {};
// now Duck has exactly one shared Animal sub-object
```

With both paths declared `virtual`, `Duck` gets exactly one shared `Animal` sub-object, resolved correctly. Under the hood, a hidden pointer/offset table tracks the shared base's actual location — genuinely comparable in spirit to `cpp1-7`'s own vtable: hidden compiler machinery solving an ambiguity problem.

Why This Is Genuinely Complex and Often Avoided

Honestly: multiple inheritance combined with virtual inheritance is widely considered one of C++'s more error-prone, rarely-actually-needed corners. Most style guides recommend avoiding multiple inheritance of *implementation* entirely, reserving it for multiple inheritance of pure *interfaces* — abstract classes containing only pure virtual functions (`cpp1-7`'s own `= 0` syntax) — which sidesteps the diamond problem entirely, since there's no actual data to duplicate.

Contrasted With Rust's Trait-Based Composition

Rust deliberately has **no multiple inheritance of structs at all**. A struct can implement any number of traits, and traits can have default method implementations — but a struct never inherits *data* from multiple sources the way a C++ class can. This sidesteps the diamond problem structurally, by design, rather than needing a virtual-inheritance fix bolted on afterward. Rust's own answer to "I need behavior from multiple sources" is composition and multiple trait implementation, not multiple inheritance of state.

CONCEPT	C++ MULTIPLE INHERITANCE	RUST TRAITS
Inheriting data from multiple sources	possible — the diamond problem's real cause	not possible at all — structurally avoided
Inheriting behavior from multiple sources	possible — via multiple abstract interfaces	possible — implement any number of traits
Fix for ambiguous shared bases	virtual inheritance — a real, non-trivial mechanism	n/a — the ambiguity was never possible to create

PREFER MULTIPLE INHERITANCE OF PURE INTERFACES

Multiple inheritance of all-pure-virtual base classes sidesteps the diamond problem entirely in the vast majority of legitimate real-world use cases — reserve multiple inheritance of actual implementation for the rare cases that genuinely need it.

BOTH INHERITANCE PATHS MUST AGREE ON VIRTUAL — ONE ALONE ISN'T ENOUGH

Marking only one of the two paths to a shared base as `virtual`, but not the other, produces inconsistent, confusing behavior. Both paths need to agree on using virtual inheritance for the fix to actually work.

Coding Challenges

CHALLENGE 1

Reproduce this chapter's Animal/Swimmer/Flyer/Duck diamond WITHOUT virtual inheritance, and report the exact compile error produced when trying to access d.name.

View solution

CHALLENGE 2

Fix Challenge 1 by adding virtual to both Swimmer's and Flyer's inheritance from Animal, then successfully set and print d.name.

View solution

CHALLENGE 3

Explain precisely why the diamond problem cannot occur in Rust at all, tying your answer to the specific structural difference between how a C++ class inherits from a base and how a Rust struct implements a trait.

View solution

CHAPTER 1 QUICK REFERENCE

- `class D : public B1, public B2 {}` — multiple inheritance, unavailable in C or Rust in this form
- **Diamond problem** — a shared base reached through two paths produces two ambiguous data copies
- `virtual public Base` on *both* paths — the fix, ensuring exactly one shared sub-object
- Prefer multiple inheritance of pure abstract interfaces over multiple inheritance of implementation
- Rust avoids this entirely — traits provide behavior, never inherited data, so the ambiguity can't arise

- **Next chapter:** templates, deeper — specialization, variadic templates, and a light touch on C++20 concepts

CHAPTER 2 OF 6

Templates, Deeper

COURSE 3 · CH 2

Templates, Deeper

cpp2-1's own deferred promise, kept — and a discipline Rust simply had from day one

`cpp2-1` deferred two things: the deeply confusing pre-C++20 template error, and full template depth. This chapter delivers both.

Template Specialization

A different implementation of a template for one specific type.

```
template<typename T>
class Box { /* generic version */ };

template<>
class Box<bool> { /* a completely different implementation, just for bool */ };
```

A real, well-known example: `std::vector<bool>` has its own famous specialization, bit-packing elements instead of storing full bytes — a genuinely different implementation from every other `std::vector<T>`.

Partial Specialization

Specializing for a *category* of types rather than one exact type — a real, useful middle ground.

```
template<typename T>
class Box<T*> { /* a different implementation for ANY pointer type */ };
```

Variadic Templates

`template<typename... Args>` accepts an arbitrary number of template arguments.

```cpp
template<typename... Args>
void print_all(Args... args) {
    ((std::cout << args << " "), ...); // C++17 fold expression
}

print_all(1, "two", 3.0);
```

This is genuinely what powers `std::make_unique` — used unexplained since `cpp2-4` — and similar "forward any number of constructor arguments" functions throughout the STL.

The Confusing Error, Resolved — C++20 Concepts

`cpp2-1`'s own deferred promise: concepts let a template constrain what operations `T` must support, directly in the template's own declaration.

```cpp
template<typename T>
concept Addable = requires(T a, T b) { a + b; };

template<Addable T>
T sum(T a, T b) { return a + b; }
```

This moves the error from deep inside the template's own instantiated body to a clear, readable message at the call site itself, naming exactly which requirement wasn't met — a genuine, real quality-of-life improvement.

Contrasted With Rust's Own Trait Bounds

Rust's `fn sum<T: Add>(a: T, b: T) -> T` has **always** worked this way, since Rust's very first stable release. Trait bounds constraining a generic parameter aren't a recent addition to Rust the way concepts are to C++ (added in C++20, decades after templates themselves) — another point where Rust simply had, by default and from day one, a discipline C++ has only recently, and optionally, caught up to.

CONCEPT	C++ PRE-C++20	C++20 CONCEPTS	RUST TRAIT BOUNDS
Constraint expressed where?	nowhere — implicit, discovered on instantiation	in the template declaration itself	in the function signature itself, since day one
Error on unmet requirement	deep, confusing, nested inside the instantiation	clear, at the call site, naming the requirement	clear, at the call site, always
When introduced	templates since ~1998	concepts added in C++20	present from Rust's first stable release

REACH FOR CONCEPTS IN NEW CODE

In C++20 and later, prefer concept-constrained templates over raw, unconstrained ones whenever possible — the improved error messages alone are a genuinely significant quality-of-life win.

VARIADIC TEMPLATE CODE CAN BECOME GENUINELY HARD TO READ

Overused, variadic templates and fold expressions can produce code that's flexible but genuinely difficult to read and debug — a real, practical readability cost worth weighing against the flexibility gained.

Coding Challenges

CHALLENGE 1

Write a class template Box with a generic implementation, then write a full specialization Box with a different implementation (e.g. storing a single bit-flag style member instead of a plain bool). Demonstrate both are used correctly for their respective types.

View solution

CHALLENGE 2

Write a variadic template function sum_all(Args... args) that adds together an arbitrary number of arguments using a fold expression, and call it with 2, 3, and 5 arguments, printing each result.

View solution

CHALLENGE 3

Explain precisely what C++20 concepts change about WHERE a template's requirements are checked and reported, and why Rust never needed an equivalent feature added later in its own history.

View solution

CHAPTER 2 QUICK REFERENCE

- **Full specialization** — a completely different implementation for one specific type
- **Partial specialization** — a different implementation for a category of types (e.g. all pointers)
- **Variadic templates** — `typename... Args`, an arbitrary number of arguments; powers `make_unique` and similar STL functions

- **C++20 concepts** — constraints declared up front, clear errors at the call site instead of deep inside instantiation
- Rust's trait bounds have worked this way since its first stable release — no equivalent retrofit was ever needed
- **Next chapter:** the Rule of Five and copy/move semantics, in full mechanical depth

CHAPTER 3 OF 6

The Rule of Five & Copy/Move Semantics In Depth

COURSE 3 · CH 3

The Rule of Five & Copy/Move Semantics In Depth

The reference table every C++ programmer eventually looks up

`cpp1-5` and `cpp2-8` both previewed the Rule of Five. This chapter is the full mechanical depth — exactly which declaration suppresses which auto-generation.

The Five Special Member Functions, Named Precisely

- Destructor — `~T()`
- Copy constructor — `T(const T&)`
- Copy assignment — `T& operator=(const T&)`
- Move constructor — `T(T&&)`
- Move assignment — `T& operator=(T&&)`

What the Compiler Generates by Default

If none are declared, the compiler generates all five automatically — each doing the "obvious" member-wise thing. This default is genuinely correct for classes made entirely of other well-behaved types. This is the Rule of Zero (`cpp2-8`), explained mechanically: it works precisely because the compiler-generated defaults are already correct when every member is itself well-behaved.

The Suppression Rules — When Declaring One Function Silently Deletes Others

The real mechanical depth. Declaring a destructor suppresses automatic generation of the move constructor and move assignment entirely — not a bad default, simply not generated at all. Declaring *any* of copy constructor, copy assignment, move constructor, or move assignment suppresses *both* move operations from being auto-generated.

A Real Table — What Suppresses What

YOU DECLARE	MOVE CTOR/ASSIGNMENT	COPY CTOR/ASSIGNMENT
`Nothing`	auto-generated	auto-generated
`Destructor`	suppressed	still generated (deprecated, legacy-only)
`Copy ctor or copy assignment`	suppressed	the other is still auto-generated
`Move ctor or move assignment`	the other is still auto-generated	suppressed entirely

`= default` and `= delete`

Modern C++11 syntax to be explicit about intent, rather than relying on implicit suppression rules.

```
Circle(const Circle&) = default; // explicitly request the compiler-generated version
Circle(const Circle&) = delete;  // explicitly forbid copying — a compile error at any call site
```

`cpp2-4`'s own `unique_ptr` uses exactly `= delete` internally on its own copy constructor — not magic, simply this syntax.

Rust's Approach Contrasted

Rust has no equivalent implicit-generation-with-suppression-rules system at all. `Copy` / `Clone` must be explicitly derived or implemented — `#[derive(Clone, Copy)]` — and **move is simply the default** for every type unless it opts into `Copy`. No five-function dance, no suppression table to memorize. A genuinely simpler, more explicit design, deliberately trading C++'s flexibility for predictability.

CONCEPT	C++	RUST
Default behavior	`complex, order-dependent generation/suppression rules`	`move by default; Copy/Clone must be explicitly opted into`
Explicit intent	`= default / = delete, optional`	`#[derive(...)], required to opt in`
Rules to memorize	`a real, non-trivial suppression table`	`none — one consistent default`

PREFER EXPLICIT = DEFAULT / = DELETE OVER RELYING ON IMPLICIT RULES

Stating intent directly in a class's own declaration makes it readable without requiring the reader to have memorized this chapter's own suppression table.

SILENTLY SUPPRESSED MOVE OPERATIONS ARE A REAL, QUIET PERFORMANCE REGRESSION

A class that unintentionally has its move operations suppressed doesn't produce a compile error — it silently falls back to (potentially expensive) copying instead. Not a bug that announces itself; a real class of performance issue worth knowing to look for.

Coding Challenges

CHALLENGE 1

Write a class with a hand-written destructor and nothing else declared. Attempt to move-construct an instance of it, and report whether the move constructor or the copy constructor actually runs (add print statements to both to observe).

View solution

CHALLENGE 2

Write a class that explicitly deletes its copy constructor with = delete, then attempt to copy an instance of it and report the resulting compile error.

View solution

CHALLENGE 3

Explain precisely why a class declaring only a destructor still ends up copying (not moving) when passed by value, tying your answer directly to this chapter's own suppression table.

View solution

CHAPTER 3 QUICK REFERENCE

- The five: destructor, copy ctor, copy assignment, move ctor, move assignment
- Declare nothing — all five are auto-generated correctly (the Rule of Zero, mechanically explained)
- Declare a destructor — move operations are suppressed entirely; copy is still generated (deprecated)
- Declare any copy or move operation — both move operations are suppressed
- `= default` / `= delete` — state intent explicitly, rather than relying on implicit rules
- Rust has no equivalent system — move by default, Copy/Clone opted into explicitly, no suppression table
- **Next chapter:** concurrency in modern C++ — std::thread, std::mutex, std::atomic

CHAPTER 4 OF 6

Concurrency in Modern C++

COURSE 3 · CH 4

Concurrency in Modern C++

c3-3's own pthreads material, revisited — a real ergonomic win, but the same honest gap against Rust remains

`c3-3` built a race condition with raw `pthreads`. This chapter shows modern C++'s own equivalents — genuinely nicer to use, but not a different safety story.

`std::thread`

A higher-level replacement for `pthread_create` / `pthread_join` — but with a real, important gotcha: a `std::thread` still joinable when its destructor runs calls `std::terminate`. It does *not* automatically join or detach the way an RAII-styled course might suggest; `.join()` or `.detach()` must be called explicitly before destruction.

```
std::thread t([]() { std::cout << "Hello from a thread\n"; });
t.join(); // required — must be called before t is destroyed
```

A Worked Example — Revisiting `c3-3`'s Race Condition

```
int counter = 0;

void increment() {
    for (int i = 0; i < 100000; i++) counter++;
}

std::thread t1(increment), t2(increment);
t1.join(); t2.join();
std::cout << counter << std::endl; // same non-deterministic race as c3-3, just different syntax
```

The exact same race — the higher-level wrapper changes nothing about the underlying problem, only the syntax used to create the threads.

`std::mutex` and `std::lock_guard`

`std::mutex` is the same concept as `c3-3`'s own `pthread_mutex_t`. `std::lock_guard<std::mutex>` is a genuine RAII wrapper around locking — the direct fix to `c3-3`'s own "forgotten unlock deadlocks" warning. Its destructor unlocks automatically, on *any* exit path, including an exception — exactly `cpp2-6`'s own exception-safety mechanism, applied specifically to locking.

```
std::mutex lock;

void increment() {
    for (int i = 0; i < 100000; i++) {
        std::lock_guard<std::mutex> guard(lock); // locks now, unlocks automatically at scope exit
        counter++;
    }
}
```

`std::atomic`

A genuinely different, lower-overhead alternative for simple shared counters and flags specifically.

```
std::atomic<int> counter{0};
counter++; // a single atomic operation — no mutex needed at all
```

Resolves `c3-3`'s own race condition with no lock/unlock pairing whatsoever — a real, meaningful alternative worth knowing for exactly this narrow case.

Contrasted With Rust's `Send` and `Sync`, Revisited

`std::mutex` / `std::lock_guard` are a genuine ergonomic improvement over `c3-3`'s raw `pthread_mutex_t` — automatic unlocking via RAII, no more forgotten-unlock deadlocks. But the *same fundamental gap* from `c3-3` remains: nothing in C++'s type system ties a `std::mutex` to the specific data it protects, and nothing prevents accessing that data without holding the lock at all. Rust's `Mutex<T>` — wrapping the data itself, checked by the `Send` / `Sync` marker traits at compile time — is still the genuinely stronger guarantee. `lock_guard` is a real ergonomic win, not a safety win at the type-system level.

CONCEPT	PTHREADS (C3-3)	MODERN C++	RUST
Thread creation	`pthread_create/join`	`std::thread, .join()/.detach()`	`std::thread::spawn`
Locking	`manual lock/unlock, forgettable`	`std::lock_guard — RAII, automatic unlock`	`Mutex<T> wraps the data itself`

CONCEPT	PTHREADS (C3-3)	MODERN C++	RUST
Mutex-to-data relationship	none — separate variables	still none — separate variables	enforced by the type system
Accessing data without locking	compiles, UB at runtime	compiles, UB at runtime	not possible — no name refers to the data directly

PREFER STD::ATOMIC FOR SIMPLE CASES, LOCK_GUARD FOR THE REST

Use `std::atomic` for simple counters/flags; `std::lock_guard` (or `std::unique_lock` for more flexibility) for anything needing genuine mutex protection. Avoid raw `lock()` / `unlock()` pairs in new code entirely.

A JOINABLE STD::THREAD DESTROYED WITHOUT JOIN/DETACH CRASHES THE PROGRAM

Unlike `lock_guard`, `std::thread` does *not* solve this via RAII — a thread still joinable when destroyed calls `std::terminate` immediately. Genuinely easy to forget on an early return or exception path, echoing `cpp2-6`'s own exception-safety themes, but not automatically handled here the way locking is.

Coding Challenges

CHALLENGE 1

Rewrite this chapter's own counter race using std::thread instead of pthreads, run it a few times, and report whether the final value is consistently 200,000.

View solution

CHALLENGE 2

Fix Challenge 1 two different ways: once using std::lock_guard around the increment, and once using std::atomic instead of a plain int with no lock at all. Confirm both produce a consistent, correct 200,000.

View solution

CHALLENGE 3

Explain precisely why std::lock_guard is described as a real ergonomic improvement over c3-3's raw pthread_mutex_t but NOT a safety improvement at the type-system level, tying your answer to what Rust's Mutex<T> does differently.

View solution

CHAPTER 4 QUICK REFERENCE

- `std::thread` — must be explicitly `.join()` ed or `.detach()` ed, or its destructor calls `std::terminate`
- `std::lock_guard<std::mutex>` — RAII locking, unlocks automatically on any exit path including exceptions
- `std::atomic<T>` — lock-free, for simple counters/flags specifically
- Modern C++'s tools are a real ergonomic improvement over raw pthreads — not a type-system safety improvement
- The mutex-to-data gap from `c3-3` remains — Rust's `Mutex<T>` still closes it, C++'s doesn't
- **Next chapter:** undefined behavior in C++ — building on `c3-4` 's own C-level UB catalog

CHAPTER 5 OF 6

Undefined Behavior in C++

COURSE 3 · CH 5

Undefined Behavior in C++

c3-4's own catalog, inherited unchanged — plus what's genuinely new once classes and virtual dispatch enter the picture

This chapter doesn't re-derive `c3-4`'s own UB deep dive — everything there still applies, unchanged. This is only what's genuinely new to C++ specifically.

Everything From `c3-4` Still Applies

Signed overflow, out-of-bounds access, dangling pointers, use-after-free, double-free, strict aliasing, excessive shifts, data races — every category `c3-4` catalogued applies unchanged, since C++ inherits C's entire UB model at the language-core level.

Object Slicing

A genuinely new-to-C++ trap: assigning a derived-class object to a base-class object *by value* — not through a pointer or reference — "slices off" the derived part entirely, leaving only the base portion. Not technically undefined behavior in the strict sense (copying just the base part is well-defined), but almost always a serious logic bug, since `cpp1-7`'s own virtual dispatch is lost entirely.

```
Circle c(5.0);
Shape s = c; // SLICED — s is now just a plain Shape, radius and Circle's own area() are gone
s.area();    // calls Shape::area(), never Circle::area() — silently
```

This is exactly what `cpp2-6`'s own "catch by reference to avoid slicing" tip was pointing toward — now fully explained.

Pure Virtual Function Called From a Constructor

A genuinely real, subtle trap: calling a virtual function from a *base* class's own constructor does **not** dispatch to a derived override, even if the object being constructed is ultimately a derived type. Calling a *pure* virtual function this way is genuinely undefined behavior, not merely surprising.

Why Construction Order Causes This

During a base class's own constructor, the object's hidden vtable pointer is set to point at the *base* class's vtable — not the derived one. It's updated to point at successively more-derived vtables as each level of construction completes, finishing only once the most-derived constructor runs. Calling a virtual function mid-construction sees whatever vtable is *currently* set, never the final one.

Uninitialized Member Variables

C++ class members of built-in types (`int`, pointers, etc.) are **not** automatically zero-initialized unless explicitly done so in the constructor — a real, common gotcha, especially coming from a language with different defaults. Rust simply doesn't allow reading an uninitialized variable at all — a compile error, closing this exact class of bug structurally.

CONCEPT	C (C3-4)	C++-SPECIFIC ADDITION	RUST
Core UB categories	`signed overflow, OOB, dangling pointers, etc.`	`inherited unchanged`	`structurally prevented in safe code`
Slicing a polymorphic object	`n/a — no inheritance`	`a real logic bug, virtual dispatch silently lost`	`n/a — no equivalent value-slicing concept`
Uninitialized member read	`UB — garbage value`	`UB — garbage value, same as C`	`compile error — cannot read before initializing`

NEVER PASS OR RETURN A POLYMORPHIC OBJECT BY VALUE

`cpp2-6`'s own advice, now fully justified: catch exceptions by reference, and generally pass/return any polymorphic object by pointer or reference, never by value — value semantics and virtual dispatch simply don't mix safely.

SLICING COMPILES SILENTLY — NOTHING ANNOUNCES IT HAPPENED

Object slicing isn't flagged by the compiler by default in most cases. It silently compiles, silently loses data and behavior, and produces no warning at the point it happens — a genuinely dangerous class of bug specifically because nothing announces it.

Coding Challenges

CHALLENGE 1

Reproduce this chapter's own Shape/Circle slicing example: assign a Circle to a plain Shape variable by value, call area() on the sliced Shape, and confirm it calls Shape's own implementation rather than Circle's.

View solution

CHALLENGE 2

Fix Challenge 1 so that calling area() correctly dispatches to Circle's own implementation, without changing anything about how the Circle object itself is constructed.

View solution

CHALLENGE 3

Explain precisely why calling a virtual function from a base class's own constructor doesn't dispatch to a derived override, tying your answer to exactly when the object's vtable pointer is updated during construction.

View solution

CHAPTER 5 QUICK REFERENCE

- Every UB category from `c3-4` applies unchanged in C++ — nothing re-derived here
- **Object slicing** — assigning a derived object to a base BY VALUE loses the derived part and virtual dispatch, silently
- Calling a virtual (especially pure virtual) function from a base constructor doesn't see the derived override — the vtable pointer isn't finalized yet
- C++ built-in-type members are not zero-initialized by default — unlike Rust, which refuses to compile a read before initialization
- Never pass/return polymorphic objects by value — always by pointer or reference
- **Next chapter:** the capstone — a modern C++ project combining everything from this entire track

CHAPTER 6 OF 6

Capstone: Building a Small Project

COURSE 3 · CH 6 — TRACK FINALE

Capstone: Building a Small Project

A polymorphic shape inventory, combining nearly every chapter across all three C++ courses

Twenty-two chapters, one piece at a time. This closing chapter builds a small shape inventory — polymorphic shapes, RAII, templates, smart pointers, exceptions, and STL algorithms combined into one working, modern C++ program.

The Project — A Shape Inventory

```
Inventory: 3 shapes, total area 103.32
  1. Circle    — area 78.54
  2. Rectangle — area 24.00
  3. Circle    — area 0.79
```

The Shape Hierarchy

`cpp1-4` / `cpp1-5` / `cpp1-7`: a `Shape` base class with virtual `area()` and `name()`, derived classes validating their own input in the constructor and throwing on invalid data (`cpp2-6`).

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

class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {
        if (r <= 0) throw std::invalid_argument("radius must be positive");
    }
    double area() const override { return 3.14159 * radius * radius; }
    std::string name() const override { return "Circle"; }
};
```

Exception Safety in the Constructors

`Circle(-5.0)` throws before construction ever completes — a real, direct application of `cpp2-6`'s own constructor-exception material. No RAII resource is ever left half-acquired.

A Generic `Repository<T>` Template

`cpp2-1`'s own templates, applied for real — a small, genuinely reusable class template, not shape-specific at all.

```
template<typename T>
class Repository {
    std::vector<std::unique_ptr<T>> items;
public:
    void add(std::unique_ptr<T> item) {
        items.push_back(std::move(item)); // cpp2-5's own move semantics
    }
    size_t size() const { return items.size(); }
    const std::unique_ptr<T>& operator[](size_t i) const { return items[i]; }
};
```

Ownership via `unique_ptr` — No Slicing, Ever

Shapes are never stored or passed by value — only via `unique_ptr<Shape>`, deliberately avoiding `cpp3-5`'s own object-slicing trap by construction. `add()` takes ownership via `std::move`, exactly `cpp2-4`'s and `cpp2-5`'s own material applied together.

Sorting With a Lambda

`cpp2-3`'s `std::sort` plus `cpp2-7`'s own lambda syntax, comparing shapes by area through their owning `unique_ptr`s.

```
std::sort(shapes.begin(), shapes.end(), [](const auto &a, const auto &b) {
    return a->area() < b->area();
});
```

`const`-Correctness Throughout

`cpp2-8`'s own discipline, applied everywhere: `area()`/`name()` are `const`; `Repository`'s own `size()`/`operator[]` are `const`-correct too — nothing in this project mutates state it doesn't need to.

Where Each Piece Came From

PIECE	CHAPTER
Shape hierarchy, RAII, polymorphism	cpp1-4, cpp1-5, cpp1-7
Exception-validated constructors	cpp2-6
Repository<T> template	cpp2-1
unique_ptr ownership, no slicing	cpp2-4, cpp3-5
std::move into the repository	cpp2-5
std::sort with a lambda	cpp2-3, cpp2-7
const-correctness throughout	cpp2-8

What's Still Out of Scope, Honestly

No multiple inheritance was needed — `cpp3-1`'s own material simply wasn't necessary here, a single hierarchy sufficed, itself a realistic outcome. No concurrency (`cpp3-4`) — this is a single-threaded tool. No hand-written Rule of Five anywhere — the entire project follows the Rule of Zero throughout, a deliberate demonstration that modern C++ rarely needs it. C++20 concepts (`cpp3-2`) weren't used either, kept to plain templates for broader compiler compatibility.

> **THE THROUGHLINE, RESTATED**

Every piece of this capstone is a direct, unmodified application of a chapter's own material — nothing new was introduced here, matching the same pattern the C track's own capstone (`c3-6`) established.

REAL CAPABILITY, NOT RUST'S COMPILE-TIME GUARANTEES

This capstone is genuinely modern, safe-by-convention C++ — RAII everywhere, no raw `new` / `delete`, no slicing, exception-safe construction. But nothing here is *enforced* by the compiler the way Rust's ownership and borrow rules would be. The discipline is real; it's still discipline, not a guarantee.

Closing the Course & Track

From `cpp1-1` 's compile model to a real, polymorphic, exception-safe, template-based, RAII-everywhere program — every chapter added one piece of real capability C never had, while this course's own repeated Rust comparisons kept one honest question in view: what does the language *enforce*, versus what does it merely *make possible*? The genuine convergence points along the way — RAII/ `Drop` , monomorphization, `unique_ptr` / `Box` , `shared_ptr` / `Rc` , lambdas/closures — show two languages solving similar problems from different starting philosophies. That comparison, sustained across 44 chapters spanning both tracks, is the actual lesson.

Coding Challenges

CHALLENGE 1

Identify which specific chapter each of the following pieces of the shape inventory came from: (a) the Repository template, (b) the exception thrown from Circle's constructor, (c) sorting shapes by area.

View solution

CHALLENGE 2

Explain why storing shapes as std::vector (by value) instead of std::vector> would break this capstone's own design, tying your answer directly to cpp3-5's own object-slicing material.

View solution

CHALLENGE 3

Across the entire C++ track, name the single idea that recurs most often, and explain in your own words why understanding it deeply matters more than memorizing any individual keyword or syntax rule.

📄 View solution

CHAPTER 6 QUICK REFERENCE — COURSE & TRACK COMPLETE

- The shape inventory combines: polymorphism (cpp1-7), RAII (cpp1-5), templates (cpp2-1), smart pointers (cpp2-4), move semantics (cpp2-5), exceptions (cpp2-6), lambdas (cpp2-7), const-correctness (cpp2-8), and avoids slicing (cpp3-5)
- Still out of scope, honestly: multiple inheritance, concurrency, hand-written Rule of Five, C++20 concepts — none were needed here
- Real capability, not compiler-enforced guarantees — the discipline is genuine, but still discipline, not Rust's kind of guarantee
- **Course 3 complete** — C++ Advanced, 6 chapters
- **Full C++ track complete** — Fundamentals + Intermediate + Advanced, 23 chapters across 3 courses