C++ Intermediate

A Complete 8-Chapter Programming Course

Topics covered:

Templates & monomorphization · The STL — containers, iterators & algorithms

Smart pointers & move semantics · Exception handling & lambdas

const-correctness & the Rule of Zero

Exercises: 24 hands-on exercises with worked solutions

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

Course 2 of 3 · the Advanced course follows

Table of Contents

CHAPTER 1 OF 8

Templates

COURSE 2 · CH 1

Templates

The exact strategy Rust's own generics use — another rare point of real convergence

Course 1 built the OOP foundation. Course 2 starts with the everyday tool that foundation — and the rest of the standard library — is actually built on top of.

The Problem Templates Solve

Writing the same logic separately for every type — a `max_int` and a separate `max_double` — is genuinely repetitive. C has no generic mechanism beyond `void *` plus manual casting (losing type safety entirely) or macros (`c2-4`'s own text-substitution trap, applied to "generic" code with none of the type checking a real function would provide).

Function Templates

```
template<typename T>
T max_value(T a, T b) {
    return (a > b) ? a : b;
}

max_value(3, 7);       // T deduced as int
max_value(2.5, 1.1);   // T deduced as double
```

Compile-Time Monomorphization

The real mechanism: the compiler generates a separate, fully concrete function for *each distinct type actually used*. Calling `max_value` with `int` and again with `double` produces two genuinely distinct compiled functions, each fully type-checked and optimized exactly as if hand-written for that specific type — not a single generic, runtime-dispatched function. Zero runtime overhead, at the cost of larger compiled binary size. This is a **direct parallel to `rust2-3`'s own generics chapter** — Rust uses the identical monomorphization strategy, another rare point of real convergence between the two languages, alongside `cpp1-5`'s own RAII/`Drop` parallel.

Class Templates

```
template<typename T>
class Box {
public:
    T value;
    Box(T v) : value(v) {}
};

Box<int> b(42);
```

This is exactly the shape `cpp2-2`'s own STL containers use — `std::vector<int>` genuinely *is* a template instantiation, not a special built-in construct.

Template Type Deduction

In most cases the compiler infers `T` from the arguments automatically — `max_value(3, 7)` works with no need to write `max_value<int>(3, 7)` explicitly, though the explicit form is always allowed.

A Compile Error You'll See — No Matching Function

If `T` doesn't support an operation the template body actually uses (e.g. calling `max_value` on a type with no `>` operator defined), the error surfaces at the *instantiation site*, often with a genuinely confusing, deeply nested message — a well-known rough edge of pre-C++20 templates. C++20 concepts improve this substantially; full depth is deferred to `cpp3-2`'s own "Templates, Deeper" chapter.

CONCEPT	C	C++ TEMPLATES	RUST GENERICS
Generic code	`void*` + manual casting, or macros — no type safety	templates — fully type-checked	generics — fully type-checked
Compilation strategy	n/a	monomorphization — one concrete copy per type used	monomorphization — the same strategy
Runtime overhead	n/a	none	none

> **TEMPLATES ARE ENTIRELY A COMPILE-TIME CONCEPT**
>
> There's no such thing as a "template object" existing at runtime — only the fully concrete, monomorphized functions and classes the compiler actually generates from each distinct instantiation.

TEMPLATE CODE USUALLY LIVES IN HEADERS, NOT SEPARATE .CPP FILES

A genuine, structural exception to `cpp1-3` / `c2-5` 's own header/definition-separation convention: the compiler needs to see a template's *full definition* at every point it's instantiated, so template code is typically written entirely in the header itself, rather than declared in a header and defined in a matching `.cpp` file.

Coding Challenges

CHALLENGE 1

Write a function template min_value(T a, T b) returning the smaller of two values, and call it with two ints and two doubles, printing both results.

View solution

CHALLENGE 2

Write a class template Pair holding two values of the same type, with a method sum() that adds them together. Instantiate it once with int and once with double, printing both results.

View solution

CHALLENGE 3

Explain precisely what monomorphization means, and why calling max_value with both int and double arguments in the same program results in genuinely two separate compiled functions rather than one generic function handling both cases at runtime.

View solution

CHAPTER 1 QUICK REFERENCE

- `template<typename T>` — declares a function or class template
- **Monomorphization** — one fully concrete, type-checked copy generated per distinct type actually used, zero runtime overhead
- A direct parallel to Rust's own generics — the identical compilation strategy
- `std::vector<int>` is literally a class template instantiation, not a special built-in
- Template errors surface at the instantiation site — often confusing pre-C++20; concepts (`cpp3-2`) improve this
- Template definitions usually live entirely in headers, not split across a .cpp file

- **Next chapter:** the STL's containers — finally resolving C's own "no standard containers" gap

CHAPTER 2 OF 8

The STL — Containers

COURSE 2 · CH 2

The STL — Containers

c3-2's own "C has no standard containers" gap, finally resolved

`c3-2` built a linked list and a hash table entirely by hand, because C offers nothing. This chapter is the direct payoff: the same shapes, already written, tested, and optimized, as class templates.

`std::vector`

```
#include <vector>

std::vector<int> nums;
nums.push_back(10);
nums.push_back(20);
std::cout << nums[0] << /* 10 */ std::endl;
```

A growable array. `c3-2`'s own linked list required a manual `malloc` / `free` pair per node; `std::vector` manages its own memory internally via RAII — the caller never calls `free` at all.

`std::vector` Under the Hood

Contiguous memory, not a linked list — growing occasionally reallocates a larger block and moves every element over. Despite this, `push_back` is **amortized O(1)** — a real, important performance characteristic: individual calls are occasionally more expensive (during a reallocation), but averaged over many calls, the cost per call stays constant.

`std::map`

```
#include <map>

std::map<std::string, int> ages;
ages["Alice"] = 30;
std::cout << ages["Alice"] << std::endl;
```

A direct resolution of `c3-2`'s own hand-built hash table. `std::map` specifically is typically a balanced tree internally, *not* a hash table — giving O(log n) operations and genuine key ordering as a side benefit. `std::unordered_map` is the hash-table-based sibling — closer to `c3-2`'s own separate-chaining structure — giving O(1) average operations but no ordering.

`std::string`

A real, growable string type — contrasted directly with `c1-8`'s own `char` arrays: no manual null-termination tracking, no manual sizing, and `.length()` / `.size()` are O(1), unlike `c1-8`'s own O(n) `strlen` scan. Concatenation uses `+`, via exactly `cpp1-6`'s own operator overloading mechanism.

```
std::string greeting = "Hello, " + name + "!";
```

RAII All the Way Down

Every one of these containers is itself an RAII wrapper — exactly `cpp1-5`'s own mechanism. A `vector`'s destructor frees every element's memory automatically when the vector goes out of scope. Nothing here is new magic — it's `cpp1-5`'s own idea, already written for you by the standard library.

CONCEPT	C (C3-2)	C++ STL
Growable list	**a hand-built linked list, manual free per node**	**std::vector — RAII-managed, contiguous memory**
Key-value store	**a hand-built hash table, separate chaining**	**std::map (ordered) / std::unordered_map (hash-based)**
String length	**strlen — O(n), scans for '\0'**	**.length() — O(1), stored directly**

STD::VECTOR IS THE DEFAULT CONTAINER CHOICE

Reach for `std::vector` unless there's a specific reason for something else — `std::map` for real key lookups, `std::unordered_map` when ordering genuinely doesn't matter and average-case speed does. This matches the real-world idiomatic default in C++.

A PUSH_BACK THAT TRIGGERS REALLOCATION INVALIDATES EXISTING ITERATORS/POINTERS

An operation that causes `std::vector` to reallocate moves every element to a new block of memory — any iterator, pointer, or reference into the old memory is left dangling. Continuing to use one afterward is undefined behavior, a genuinely real and easy-to-hit trap. Iterators get full treatment in `cpp2-3`.

Coding Challenges

CHALLENGE 1

Create a std::vector, push_back five values onto it, and print each one using a plain indexed loop with operator[].

View solution

CHALLENGE 2

Create a std::map storing three people's ages, then look up and print one specific person's age by name.

View solution

CHALLENGE 3

Explain why std::vector's push_back is described as "amortized O(1)" rather than simply "O(1)," and what specifically happens during the occasional more expensive call that the amortized average accounts for.

View solution

CHAPTER 2 QUICK REFERENCE

- `std::vector<T>` — a growable, RAII-managed, contiguous array; resolves `c3-2`'s linked-list gap
- `std::map<K, V>` — ordered, tree-based; `std::unordered_map` — hash-based, closer to `c3-2`'s own structure
- `std::string` — O(1) length, no manual null-termination, `+` for concatenation via operator overloading
- Every STL container is itself an RAII wrapper — `cpp1-5`'s own idea, pre-written
- Reallocation-triggering operations invalidate existing iterators/pointers/references — a real, common trap
- **Next chapter:** iterators and algorithms — std::sort, std::find, and generic operations over any container

CHAPTER 3 OF 8

The STL — Iterators & Algorithms

COURSE 2 · CH 3

The STL — Iterators & Algorithms

Templates, applied one level up — algorithms that don't know or care which container they're given

`cpp2-2`'s containers gave C++ real, standard data structures. This chapter is what makes them interchangeable: one `std::sort`, working identically across every container that offers iterators.

What an Iterator Is

A generalized pointer-like object. `begin()` points at the first element; `end()` points *one past* the last — never the last element itself, the "half-open range" convention `[begin, end)`. Dereferencing (`*it`) gives the element; `++it` advances.

```
std::vector<int> nums = {10, 20, 30};
for (auto it = nums.begin(); it != nums.end(); ++it) {
    std::cout << *it << std::endl;
}
```

Iterating With Iterators

`cpp1-2`'s own range-based `for (auto x : container)` is genuinely **sugar** for exactly this iterator pattern, generated automatically underneath.

Why Iterators Exist — Generic Algorithms Over Any Container

The real payoff: `std::sort`/`std::find`/and friends don't know or care whether they're operating on a `vector`, a `list`, or a raw array — they only need `begin()`/`end()` iterators satisfying certain requirements. This is `cpp2-1`'s templates applied *one level up*, at the algorithm level rather than the container level — `std::sort` is itself a function template.

`std::sort`

```
std::sort(nums.begin(), nums.end());
```

Sorts a range in place. `c3-2`'s own hand-built structures never included a general sorting algorithm at all — writing one in C means implementing the actual algorithm by hand, every time.

`std::find`

```
auto it = std::find(nums.begin(), nums.end(), 20);
if (it != nums.end()) {
    std::cout << "Found!" << std::endl;
}
```

A linear search returning an iterator to the found element, or `end()` if not found — the `end()`-as-sentinel pattern is idiomatic and used constantly across the STL.

Iterator Invalidation, Revisited

`cpp2-2`'s own warn-box flagged reallocation invalidating iterators — with iterators now properly introduced, the fuller picture: erasing or inserting into a `vector` can invalidate iterators at or after the modification point too, not just growth-triggered reallocation.

CONCEPT	C (C3-2)	C++ STL
Traversal	hand-written loop, container-specific	iterators — the same syntax across any container
Sorting	implement the algorithm by hand	std::sort — one call, any container
Searching	hand-written traversal loop	std::find — returns an iterator, end() as "not found"

PREFER A STANDARD ALGORITHM OVER HAND-WRITING THE EQUIVALENT LOOP

`std::sort` / `std::find` and the rest of the STL's algorithm library are genuinely less error-prone than a hand-written loop, and usually at least as fast — real implementations are heavily optimized.

USING AN INVALID OR MISMATCHED ITERATOR IS UNDEFINED BEHAVIOR, UNCAUGHT BY THE COMPILER

Using an iterator from one container against a different container's `begin()` / `end()`, or one that's already been invalidated, compiles cleanly and is undefined behavior at runtime — the type system doesn't tie an iterator to "the container it's still safely valid for," the way Rust's borrow checker would.

Coding Challenges

CHALLENGE 1

Create a std::vector with values in unsorted order, sort it with std::sort, and print the result using an explicit iterator-based loop (not a range-based for).

View solution

CHALLENGE 2

Use std::find to search a std::vector for a value that exists and a value that doesn't, printing "Found" or "Not found" for each using the end() sentinel check.

View solution

CHALLENGE 3

Explain why the range-based for loop (for (auto x : container)) is described as "sugar" over the explicit iterator loop, and rewrite one range-based for loop yourself as its equivalent explicit iterator-based form.

View solution

CHAPTER 3 QUICK REFERENCE

- `begin()` / `end()` — the half-open range `[begin, end)`; `end()` points one past the last element
- Range-based `for` is sugar over the exact same iterator pattern, generated automatically
- Algorithms like `std::sort` / `std::find` work identically across any container offering iterators — templates, one level up
- `std::find` returns `end()` as its "not found" sentinel — check `it != container.end()`
- Erasing/inserting/reallocating can invalidate existing iterators — using one afterward is undefined behavior, uncaught by the compiler
- **Next chapter:** smart pointers — unique_ptr and shared_ptr, a modern RAII alternative to raw malloc/free

CHAPTER 4 OF 8

Smart Pointers

COURSE 2 · CH 4

Smart Pointers

The chapter cpp1-5's own tip-box pointed toward — and where C++'s and Rust's ownership models genuinely converge

`cpp1-5`'s own Challenge 2 built a leak on purpose: a `Circle*` allocated with `new`, never deleted, because RAII only protects objects whose own lifetime is tied to a scope — and a raw pointer variable's scope has nothing to do with the heap object it points at. This chapter is the real fix.

The Gap Raw Pointers Leave Open

A raw pointer going out of scope destroys only the pointer itself, never the object it points to. Smart pointers close this gap by wrapping the raw pointer *inside* an RAII-managed object — exactly `cpp1-5`'s own idea, applied specifically to ownership.

`unique_ptr` — Sole Ownership

```
#include <memory>

std::unique_ptr<Circle> c = std::make_unique<Circle>(5.0);
// c is automatically deleted when it goes out of scope — no explicit delete, ever
```

Exactly one `unique_ptr` owns a given object at a time. When it's destroyed, it automatically deletes the object it owns — genuinely just `cpp1-5`'s own `IntArray` wrapper, generalized and already written for you.

`unique_ptr` Cannot Be Copied

A real, deliberate constraint: copying a `unique_ptr` would mean two owners both believing they're responsible for deleting the same object — exactly the double-free class from `c2-3`'s own catalog. The compiler enforces this directly; attempting to copy one is a **compile error**, not a runtime risk.

```
std::unique_ptr<Circle> c2 = c; // compile error — unique_ptr cannot be copied
std::unique_ptr<Circle> c2 = std::move(c); // fine — ownership transfers; c is now empty
```

Ownership can be *transferred* via `std::move` — a direct preview of `cpp2-5`'s own Move Semantics chapter. After the move, the source `unique_ptr` becomes empty.

`shared_ptr` — Shared Ownership via Reference Counting

```
std::shared_ptr<Circle> a = std::make_shared<Circle>(5.0);
std::shared_ptr<Circle> b = a; // fine — both now share ownership; reference count is 2
```

Multiple `shared_ptr`s can jointly own the same object, tracked by a hidden reference count. The object is deleted only when the *last* owning `shared_ptr` is destroyed.

The Real Comparison to Rust

`unique_ptr` is essentially Rust's `Box<T>` — sole ownership, moved not copied, deterministic destruction. `shared_ptr` is essentially Rust's `Rc<T>` — reference-counted shared ownership, the identical strategy. This is genuinely another point of real convergence, alongside `cpp1-5`'s RAII/`Drop` parallel and `cpp2-1`'s monomorphization parallel — C++'s and Rust's own ownership *models* visibly converge here.

A Genuine Gap Even Here — No Compile-Time Enforcement

The honest caveat: nothing stops a programmer from *also* keeping a raw pointer to the object a `unique_ptr` owns, and using that raw pointer after the `unique_ptr` is destroyed — a genuine use-after-free, still fully possible. Smart pointers eliminate the **forgetful** class of memory bug — nobody forgets to call `delete` anymore — not the **deliberate-misuse** class. Rust's borrow checker closes both.

CONCEPT	RAW POINTER	UNIQUE_PTR	SHARED_PTR	RUST
Ownership	none — just an address	sole owner	shared, ref-counted	Box<T> / Rc<T>
Copy behavior	freely copyable	compile error — must move	copyable — increments the count	move / Rc::clone
Compile-time misuse prevention	none	prevents double-ownership only	prevents double-ownership only	prevents use-after-free entirely

> **UNIQUE_PTR IS THE DEFAULT; SHARED_PTR IS THE EXCEPTION**

Reach for `unique_ptr` first — reserve `shared_ptr` for cases where genuine shared ownership is actually needed, since reference counting carries real runtime overhead `unique_ptr` doesn't have at all.

A SHARED_PTR REFERENCE CYCLE LEAKS, DESPITE BEING "SMART"

Two objects each holding a `shared_ptr` to the other never reach a zero reference count between them — a genuine memory leak, smart pointers notwithstanding. `std::weak_ptr` exists specifically to break such cycles, a non-owning reference that doesn't contribute to the count.

Coding Challenges

CHALLENGE 1

Rewrite cpp1-5's own Challenge 2 (a Circle allocated with new, never deleted) using std::unique_ptr instead, adding a destructor message to Circle to confirm it's now genuinely destroyed automatically.

View solution

CHALLENGE 2

Create two shared_ptrs jointly owning the same object, print the reference count after each is created (use_count()), then let one go out of scope and print the count again to observe it decrease.

View solution

CHALLENGE 3

Explain precisely why smart pointers are described as eliminating the "forgetful" class of memory bug but not the "deliberate misuse" class, giving a concrete scenario where a raw pointer kept alongside a unique_ptr still produces a genuine use-after-free.

View solution

CHAPTER 4 QUICK REFERENCE

- `std::unique_ptr<T>` — sole ownership, cannot be copied, transferred only via `std::move`
- `std::shared_ptr<T>` — reference-counted shared ownership, deleted when the last owner is destroyed
- **unique_ptr ≈ Rust's Box<T>; shared_ptr ≈ Rust's Rc<T>** — a real convergence in ownership model
- Smart pointers eliminate forgotten-`delete` bugs, not deliberate raw-pointer misuse — Rust's borrow checker closes both

- `std::weak_ptr` — a non-owning reference, the escape hatch for `shared_ptr` reference cycles
- **Next chapter:** move semantics and rvalue references — the mechanism `std::move` already previewed here

CHAPTER 5 OF 8

Move Semantics & Rvalue References

COURSE 2 · CH 5

Move Semantics & Rvalue References

std::move doesn't move anything — it's a cast. The real mechanism, revealed.

`cpp2-4` used `std::move` to transfer a `unique_ptr`'s ownership without explaining why it worked. This chapter is that explanation, in full.

The Problem — Unnecessary Copies

Returning a large object like a `vector` by value historically meant a full, expensive deep copy. Before C++11, this was a genuine performance problem, pushing programmers toward awkward workarounds — passing output parameters by reference instead of simply returning by value.

`lvalue`s and `rvalue`s

A necessary vocabulary: an **lvalue** has a name and persistent identity — an ordinary variable. An **rvalue** is a temporary with no persistent identity — a literal, or a function's return value. `T&` binds only to lvalues; `T&&` (a new syntax — a **rvalue reference**) binds only to rvalues. This distinction is exactly what lets the compiler choose between a copy and a move, at compile time, based on which kind of value is actually being used.

Move Constructors and Move Assignment

A class can define a **move constructor**, taking an rvalue reference. Instead of copying the source object's resources, it *steals* them — pointer/handle fields are copied over, then explicitly nulled out in the source, so the source's own destructor doesn't also try to free the same resource. This directly avoids `c2-3`'s own double-free bug class, by deliberate design.

```
class Buffer {
public:
    int *data;

    Buffer(Buffer &&other) { // move constructor
        data = other.data;
        other.data = nullptr; // prevents other's destructor from double-freeing
    }
};
```

`std::move` — Casting an lvalue to an rvalue

The real mechanism: `std::move` doesn't actually move anything by itself — it's purely a **cast**, converting an lvalue (which would normally bind to `T&`, triggering a copy) into an rvalue reference (`T&&`), so the move constructor gets selected instead of the copy constructor. A genuinely important, often-misunderstood fact: `std::move` performs no action of its own at all.

```
Buffer a;
Buffer b = std::move(a); // casts a to an rvalue reference — selects the move constructor
```

After a Move, the Source Is in a "Valid But Unspecified" State

A precise, real rule: a moved-from object is guaranteed to still be safely destructible and (usually) safely reassignable — but its actual *value* is unspecified. Reading it for anything else is a logic bug, though **not** undefined behavior the way `c1-7`'s dangling pointer would be — a genuinely important, precise distinction.

Contrasted With Rust's Own Move-By-Default Semantics

Rust performs this exact optimization **automatically and by default**, per `rust1-3`'s own ownership model — passing or returning a value moves it unless the type is `Copy`, with no special syntax needed and no possibility of forgetting to move efficiently. C++ requires explicitly opting in via `std::move`, or relying on the compiler's own automatic Return Value Optimization in specific cases — another point where Rust simply enforces, by default, a discipline C++ makes available but optional.

CONCEPT	C++	RUST
Moving instead of copying	**opt-in — std::move, or compiler RVO**	**automatic and default — per rust1-3's ownership model**
Forgetting to move efficiently	**genuinely possible — an unnecessary copy happens silently**	**not possible — the compiler always moves by default**

CONCEPT	C++	RUST
Using a moved-from value	a logic bug (valid but unspecified), not UB	a compile error — the compiler tracks it

MODERN COMPILERS OFTEN APPLY RETURN VALUE OPTIMIZATION AUTOMATICALLY

A function returning a local object by value frequently doesn't need an explicit `std::move` at all — the compiler elides the copy/move entirely in many cases. Worth knowing so as not to sprinkle `std::move` defensively where it isn't needed.

USING AN OBJECT'S ORIGINAL VALUE AFTER MOVING FROM IT IS A REAL LOGIC BUG

Calling `std::move(a)` and then continuing to read `a`'s value afterward (rather than only destroying or reassigning it) silently reads whatever unspecified state the object was left in. Not undefined behavior — but definitely wrong.

Coding Challenges

CHALLENGE 1

Write a class with a raw pointer member and a move constructor that steals the pointer and nulls out the source. Move-construct one instance from another, then print a message confirming the source's pointer is now nullptr.

View solution

CHALLENGE 2

Move a std::string into another variable with std::move, then print the moved-from string's value. Explain what you observe and why reading it afterward is a logic bug rather than undefined behavior.

View solution

CHALLENGE 3

Explain precisely why std::move is described as "performing no action of its own" — what does it actually do at the type-system level, and what causes the move constructor to actually run afterward?

View solution

CHAPTER 5 QUICK REFERENCE

- **lvalue** — has a name/identity; **rvalue** — a temporary with none
- `T&&` — an rvalue reference, binds only to rvalues, enables move constructors/assignment
- A move constructor steals resources and nulls the source, avoiding a double-free
- `std::move` is purely a cast to an rvalue reference — it moves nothing by itself
- A moved-from object is valid but unspecified — safe to destroy/reassign, a logic bug (not UB) to read otherwise
- Rust moves automatically by default; C++ requires opting in via `std::move` or relying on RVO
- **Next chapter:** exception handling — try/catch/throw, and RAII's role in exception safety

CHAPTER 6 OF 8

Exception Handling

COURSE 2 · CH 6

Exception Handling

cpp1-5's own exception safety preview, shown in full — and the strongest guarantee still belongs to Rust

`cpp1-5` previewed exception safety briefly. This chapter delivers it in full — and closes with the honest, three-way comparison this whole course has been building toward.

C's Total Absence of Error Propagation

Worth naming directly, for the first time: C has no exceptions, no `Result` type, nothing beyond a manual return code or `errno` that the caller must remember to check, every single time, with nothing enforcing that check at all.

`throw`, `try`, `catch`

```
try {
    throw std::runtime_error("something went wrong");
} catch (const std::exception &e) {
    std::cout << "Caught: " << e.what() << std::endl;
}
```

Stack Unwinding

When an exception is thrown, the stack **unwinds**: every function between the throw site and the matching `catch` exits, and every local object along the way has its destructor called, in reverse order of construction. This is exactly `cpp1-5`'s own "Exception Safety, Briefly" preview, now shown in full.

RAII's Role in Exception Safety

The real payoff: because destructors run during unwinding, RAII-managed resources — smart pointers, containers, file handles wrapped in RAII types — are cleaned up automatically even when an exception is

thrown mid-function, with zero additional code from the programmer. A raw resource acquired with `new` / `malloc` and manually freed at a function's end **leaks** if an exception is thrown before that manual free line is ever reached — a genuine, concrete argument *for* using RAII wrappers pervasively, not just a style preference.

Standard Exception Types

`std::exception` is the base type, with `std::runtime_error`, `std::out_of_range`, and others derived from it. Catch by reference (`const std::exception&`) to avoid slicing — a real forward pointer to `cpp3-1`'s own object-slicing material.

Exceptions vs. Return Codes vs. Rust's `Result<T, E>`

Three real, genuinely different guarantees. C's manual return codes/`errno`: nothing enforces checking them — silently ignorable. C++ exceptions: an uncaught exception terminates the program rather than silently continuing with a wrong value — genuinely stronger than C — but *which* exceptions a function might throw is invisible in its own signature, a real, honest downside. Rust's `Result<T, E>`: the possibility of failure is part of the function's own *type*, and the compiler forces the caller to at least acknowledge it — via `?`, `.unwrap()`, or a `match` — genuinely the strongest of the three.

CONCEPT	C	C++	RUST
Failure signaling	manual return code / errno	exceptions — throw/catch	Result<T, E>
Ignoring failure silently	fully possible — nothing enforces a check	uncaught exception terminates the program	compile-time forced acknowledgment
Visible in the function's own type	no	no — throw specifications aren't part of the signature	yes — Result<T, E> is the return type itself

RAII ISN'T JUST CONVENIENT — IT'S THE ACTUAL EXCEPTION-SAFETY MECHANISM

Prefer RAII wrappers specifically *because* of exception safety, not merely to avoid writing manual cleanup code — a concrete, practical reason beyond convenience.

A DESTRUCTOR THAT THROWS DURING UNWINDING CAN CRASH THE WHOLE PROGRAM

If a destructor itself throws while the stack is already unwinding from another exception, the result is a call to `std::terminate` — the entire program crashes outright. Destructors should essentially never throw; a real, important C++ idiom.

Coding Challenges

CHALLENGE 1

Write a function that throws a std::runtime_error if a passed-in int is negative, and a try/catch block in main that calls it with a negative value, printing the caught exception's message via e.what().

View solution

CHALLENGE 2

Write a class with a destructor that prints a message, create an instance inside a try block, throw an exception after creating it, and catch the exception in main. Observe when the destructor's message actually prints relative to the catch block.

View solution

CHALLENGE 3

Explain precisely why Rust's Result<T, E> is described as the strongest of the three failure-signaling guarantees compared in this chapter, tying your answer to what specifically is and isn't visible in a function's own type signature in each language.

View solution

CHAPTER 6 QUICK REFERENCE

- `throw` / `try` / `catch` — C++'s error-propagation mechanism, entirely absent from C
- **Stack unwinding** — every local object's destructor runs, in reverse order, between the throw and the catch
- RAII resources clean up automatically during unwinding — a raw resource with manual cleanup leaks on an exception
- Catch by `const std::exception&` to avoid slicing (`cpp3-1`)
- C: silently ignorable failure. C++: terminates if uncaught, but invisible in the type signature. Rust: forced compile-time acknowledgment via `Result<T, E>`
- Destructors should never throw — doing so during unwinding calls `std::terminate`
- **Next chapter:** lambda expressions — closures in C++, contrasted with Rust's own

CHAPTER 7 OF 8

Lambda Expressions

COURSE 2 · CH 7

Lambda Expressions

cpp1-6's own operator() mechanism, generated automatically by the compiler

`cpp2-3`'s `std::sort` works with any callable — this chapter introduces the concise way real C++ code writes one inline, and reveals it's built from a mechanism already covered.

Lambda Syntax

```
std::vector<int> nums = {5, 2, 8, 1};
std::sort(nums.begin(), nums.end(), [](int a, int b) {
    return a > b; // descending order
});
```

`[capture](params) { body }` — a lambda passed directly as `std::sort`'s custom comparator, exactly `cpp2-3`'s own algorithm-plus-callable pattern.

Capture Lists

The real new concept: how a lambda accesses variables from its surrounding scope.

- `[]` — captures nothing
- `[=]` — captures everything used, by value (a copy)
- `[&]` — captures everything used, by reference
- `[x]` — captures just `x`, by value
- `[&x]` — captures just `x`, by reference

What a Lambda Actually Is — A Class With `operator()`

The real mechanism, connecting directly back to `cpp1-6`: a lambda is compiler-generated syntactic sugar for an anonymous class, with captured variables as member fields and an overloaded `operator()` containing the lambda's body. Not a new language feature at its core — `cpp1-6`'s own mechanism, applied automatically.

Capturing by value copies into those member fields at the point the lambda is *created*, not when it's called — a real, sometimes-surprising timing detail.

```
// [x](int y) { return x + y; } is roughly sugar for:
class __lambda {
    int x; // captured member field, copied at creation time
public:
    __lambda(int x_) : x(x_) {}
    int operator()(int y) { return x + y; }
};
```

`std::function`

A type-erased wrapper capable of holding *any* callable matching a given signature — a lambda, a function pointer (`c2-7`'s own material), or a class with `operator()`. Useful because every lambda genuinely has its own unique, compiler-generated type, even when two lambdas look identical in source — `std::function` gives them a common type to be stored or passed around as.

```
std::function<int(int, int)> op = [](int a, int b) { return a + b; };
```

Contrasted With Rust's Own Closures

Rust closures work via the same underlying idea — the compiler generates a hidden struct capturing referenced variables, implementing one of the `Fn` / `FnMut` / `FnOnce` traits, genuinely comparable to `operator()`. The real difference: Rust's own capture rules are *inferred automatically* from how the closure body actually uses each variable (by reference, mutable reference, or move), rather than requiring an explicit capture-list syntax the way C++ does. A genuine ergonomic difference — but the underlying "anonymous callable object" mechanism is, once again, a real point of convergence, alongside RAII/`Drop` and monomorphization.

CONCEPT	C++ LAMBDA	RUST CLOSURE
Underlying mechanism	an anonymous class with operator()	a hidden struct implementing Fn/FnMut/FnOnce
Capture specification	explicit — [x], [&x], [=], [&]	inferred automatically from usage
Each closure's type	unique, unnameable — needs std::function or auto	unique, unnameable — needs a generic bound or Box<dyn Fn>

> **CAPTURE ONLY WHAT'S NEEDED, BY THE NARROWEST MEANS**

Prefer `[x]` over `[=]`, and `[&x]` over `[&]` — both a performance consideration (avoiding unnecessary copies) and a safety consideration (fewer captured references means fewer chances of a dangling reference).

A REFERENCE-CAPTURING LAMBDA THAT OUTLIVES ITS CAPTURED VARIABLE DANGLES

A lambda capturing by reference (`[&]` or `[&x]`) that's stored and called after the enclosing function has returned holds a dangling reference — exactly `cpp1-8`'s own dangling-reference warning, reached through a lambda's capture instead of an explicit reference variable.

Coding Challenges

CHALLENGE 1

Use std::sort with a lambda comparator to sort a std::vector in descending order, then print the result.

View solution

CHALLENGE 2

Write a lambda that captures a local int by value and one that captures the same variable by reference. After creating both lambdas, change the original variable's value, then call both lambdas and explain why they produce different results.

View solution

CHALLENGE 3

Explain precisely why a lambda is described as "not a new language feature at its core," tying your answer directly back to cpp1-6's own operator overloading material and what a lambda actually compiles down to.

View solution

CHAPTER 7 QUICK REFERENCE

- `[capture](params) { body }` — lambda syntax; used directly with `cpp2-3`'s own algorithms
- `[]` / `[=]` / `[&]` / `[x]` / `[&x]` — capture nothing, everything by value, everything by reference, or specific variables
- A lambda is compiler-generated sugar for a class with `operator()` — `cpp1-6`'s own mechanism, automated
- By-value captures copy at lambda *creation* time, not call time
- `std::function` — a type-erased wrapper for any callable, since every lambda's real type is unique and unnameable
- Reference-capturing lambdas that outlive their captured variables dangle — the same bug class as `cpp1-8`

- **Next chapter:** const-correctness and modern C++ style — closing Course 2

CHAPTER 8 OF 8

const-Correctness & Modern C++ Style

COURSE 2 · CH 8 — COURSE FINALE

const-Correctness & Modern C++ Style

The Rule of Five exists to be understood — the Rule of Zero exists to be practiced

Course 2 built templates, containers, smart pointers, move semantics, exceptions, and lambdas. This closing chapter ties them together into the actual practice of writing idiomatic modern C++.

`const`-Correctness, In Full

`cpp1-2` introduced `const` briefly; `cpp1-8` covered `const T&` parameters. The remaining piece: a **const member function** promises not to modify the object, enforced by the compiler.

```
double area() const { return 3.14159 * radius * radius; } // promises not to modify *this
```

This is a real, propagating discipline — a `const` object can only call `const`-qualified member functions on itself.

The Rule of Three

Revisiting `cpp1-5`'s own preview directly: if a class defines *any one* of a destructor, a copy constructor, or a copy assignment operator, it almost certainly needs all three. It's managing a resource manually, and the compiler's own default-generated versions of the other two would perform a shallow copy — two objects both believing they own the same resource, leading straight to `c2-3`'s own double-free bug class.

The Rule of Five

C++11 extended it: the move constructor and move assignment operator (`cpp2-5`'s own material) join the same rule. Writing any one of the five is a real signal you likely need to think through all five.

The Rule of Zero

The modern, actually-recommended alternative: **don't manage raw resources directly at all**. Use RAII wrappers — smart pointers (`cpp2-4`), STL containers (`cpp2-2`) — for every member, and let the compiler's default-generated destructor, copy, and move operations work correctly automatically, since each member's own RAII type already does the right thing on its own. A genuine, real best practice: the Rule of Five exists to be *understood*; the Rule of Zero exists to be *practiced*. Most well-written modern C++ classes need none of the five written by hand.

```
class Owner {
    std::unique_ptr<Resource> res; // RAII member — no destructor/copy/move needed by hand at all
};
```

`auto` and Range-Based `for` Loops, Revisited

`cpp1-2`'s `auto` and `cpp2-3`'s range-based `for`, named together now as part of "modern C++ style" — alongside the Rule of Zero, smart pointers, and RAII, this is what a well-written modern codebase actually looks like, in sharp contrast to older, pre-C++11 "C with Classes" style still using raw `new`/`delete` and hand-written Rule-of-Three classes everywhere.

Closing Course 2

Templates → STL containers → iterators/algorithms → smart pointers → move semantics → exceptions → lambdas — now const-correctness and the Rule of Zero tie the whole course together into a genuine practice, not just a list of features. Course 3 goes deeper still: multiple inheritance, templates in more depth, the Rule of Five's full mechanics, concurrency, C++-specific undefined behavior, and a real capstone.

CONCEPT	OLD "C WITH CLASSES"	MODERN C++
Resource management	raw new/delete, hand-written Rule of Three/Five	RAII wrappers, Rule of Zero
Containers	hand-rolled or raw arrays	std::vector/std::map and the rest of the STL
Type declarations	always spelled out explicitly	auto where it aids readability

WRITING A DESTRUCTOR BY HAND? STOP AND ASK WHY

In modern C++, hand-writing a destructor is a real signal worth pausing on — could an existing RAII type (a smart pointer, a container) hold that resource instead? The Rule of Zero as a practical, everyday habit, not just a rule to recite.

FOLLOWING NEITHER RULE LEAVES A GENUINE DOUBLE-FREE TRAP

A class manually managing a raw pointer member, but relying on the compiler's default-generated copy constructor (following neither the Rule of Five nor the Rule of Zero), gets a shallow copy — two objects both pointing at the same heap memory, both believing they own it, leading straight to a double-free exactly like `c2-3`'s own bug catalog.

Coding Challenges

CHALLENGE 1

Write a class with a const member function that computes and returns a value from the object's own data without modifying it. Then attempt to call a non-const member function on a const instance of that class, and report the compile error.

View solution

CHALLENGE 2

Write a class that follows the Rule of Zero by holding a std::unique_ptr as its only member, with no hand-written destructor, copy, or move operations at all. Demonstrate that moving an instance still works correctly.

View solution

CHALLENGE 3

Write a class managing a raw new'd int* member with a hand-written destructor, but deliberately no copy constructor or copy assignment operator. Copy an instance of it, let both copies go out of scope, and explain precisely what goes wrong.

View solution

CHAPTER 8 QUICK REFERENCE — COURSE 2 COMPLETE

- **const member functions** — a compiler-enforced promise not to modify the object
- **Rule of Three** — destructor, copy constructor, copy assignment: write one, likely need all three
- **Rule of Five** — the Rule of Three plus move constructor and move assignment (C++11)
- **Rule of Zero** — the modern practice: hold RAII members only, let the compiler generate everything correctly
- Modern C++ style: RAII, smart pointers, the STL, `auto`, range-based `for` — vs. older raw new/delete "C with Classes"

- **Course 2 complete.** Course 3 covers multiple inheritance, deeper templates, the Rule of Five in full, modern concurrency, C++-specific UB, and a real capstone