C++ Fundamentals

A Complete 9-Chapter Programming Course

Topics covered:

The C-compatible core & iostream · References & a real bool

Function overloading & name mangling · Classes, RAII & operator overloading

Inheritance & polymorphism · References vs. pointers · Namespaces

Exercises: 27 hands-on exercises with worked solutions

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

Course 1 of 3 · Intermediate and Advanced courses follow

Table of Contents

CHAPTER 1 OF 9

Getting Started

COURSE 1 · CH 1

Getting Started

Everything from the C track, still true — plus what changes the moment OOP gets layered on top

`c3-6` named this course directly: the next step once OOP is layered on top of everything the C track already covers. Most of what you already know still applies. This chapter is about the specific places it doesn't.

`g++` and `clang++`

The same underlying toolchains as `c1-1`'s `gcc` / `clang`, extended for C++. The situation around build tooling hasn't changed either — no official package manager or build system, exactly as before; `make`/CMake still fill that gap, and `c2-5`'s own Makefile material transfers directly, unchanged.

```
$ g++ --version
g++ (Ubuntu 13.2.0) 13.2.0
```

What's Source-Compatible With C

A large majority of valid C code compiles as valid C++, unchanged: variables, control flow, functions, arrays, pointers, and plain structs all carry over. Everything the C track covered about manual memory, bounds checking, and pointer discipline still applies at the same level as before — none of that gets replaced, only added to.

What's *Not* Compatible

A concrete, common example: in C, `malloc`'s `void *` return implicitly converts to any pointer type. In C++, it does **not** — an explicit cast is required.

```
// valid C, invalid C++:
int *p = malloc(10 * sizeof(int));

// required in C++:
int *p = (int *)malloc(10 * sizeof(int));
```

This reflects a broader theme worth noting immediately: C++'s type checking is stricter than C's in several places, not looser.

`iostream` vs. `stdio`

`cout` / `cin` / `cerr` are C++'s own I/O streams — using `<<` and `>>`, which are genuinely overloaded operators (Chapter 6 covers writing your own), not format-string parsing the way `printf` / `scanf` work. Both remain fully usable in C++ — `<cstdio>` still works — but `iostream` is the idiomatic choice.

```
#include <iostream>

std::cout << "Value: " << 42 << std::endl;
```

The Compile-Then-Link Model, Unchanged

`c1-1`'s compile-then-link model transfers completely unchanged — same two conceptual steps, same idea of object files and linking. The only visible difference is convention: C++ source files typically use a `.cpp` extension, and `g++` replaces `gcc`.

The Smallest C++ Program

```
#include <iostream>

int main() {
    std::cout << "Hello, C++!" << std::endl;
    return 0;
}
```

Compare directly against `c1-1`'s own hello world — the shape is identical; only the I/O mechanism changed.

CONCEPT	C	C++
Print a value	`printf("%d\n", x)`	`std::cout << x << std::endl;`
void* implicit conversion	`allowed`	`requires an explicit cast`
File extension convention	`.c`	`.cpp`
Compiler	`gcc / clang`	`g++ / clang++`

> **THE C TRACK'S DISCIPLINE CARRIES FORWARD, NOT SOMETHING TO RELEARN**

Since most valid C is also valid C++, the bounds-checking habits and manual-control mindset built across the entire C track apply directly here — this course adds to that foundation, it doesn't replace it.

A REAL "COMPILES IN C, DOESN'T COMPILE IN C++" GOTCHA

The `void *`-to-typed-pointer implicit conversion difference is a genuine, common trap when porting C code to C++ or mixing the two — code that compiled cleanly as C can fail to compile once treated as C++, specifically because C++'s type checking is stricter here.

Coding Challenges

CHALLENGE 1

Write a hello-world program using iostream that prints two separate lines with two separate std::cout statements, compile it with g++, and run it.

View solution

CHALLENGE 2

Take a small C program using malloc without an explicit cast on its return value, and compile it first as C (gcc), then attempt to compile the identical file as C++ (g++). Report what happens in each case.

View solution

CHALLENGE 3

Explain why the malloc void* conversion difference is evidence that C++ is, in at least this respect, MORE strictly typed than C — not simply "different" — and why that's a genuinely counter-intuitive fact given C++ is usually introduced as C "with more features."

View solution

CHAPTER 1 QUICK REFERENCE

- `g++` / `clang++` — same underlying toolchains as C's own compilers, extended
- Most valid C compiles as valid C++ unchanged — the manual-control discipline carries forward
- `void *` requires an explicit cast to a typed pointer in C++ — not implicit, unlike C
- `iostream`'s `cout` / `cin` use overloaded `<<` / `>>`, not format strings — `stdio` still works too
- The compile-then-link model and `c2-5`'s Makefile material transfer completely unchanged

- **Next chapter:** variables, references, and a genuine bool primitive from day one

CHAPTER 2 OF 9

Variables, References & Basic Types

COURSE 1 · CH 2

Variables, References & Basic Types

A real boolean from the start, and a second way to refer to a variable that isn't a pointer

`c1-3` spent real time on C's historical lack of a true boolean. This chapter opens by closing that gap for good — then introduces something the C track never had at all.

A Genuine `bool` From Day One

C had no boolean type until C99, and even then, `bool` is a macro over `_Bool` — genuinely just an integer in disguise. C++ has had a real, distinct `bool` primitive since its very first standard. `true` and `false` are genuine keyword literals of their own type — not macros standing in for `1` and `0`.

```
bool is_valid = true;
```

References — A New Concept Beyond Pointers

A reference is an **alias** for an existing variable — not a separate object with its own address the way a pointer is.

```
int x = 5;
int &ref = x;   // ref is another name for x, not a pointer to it
ref = 10;      // this changes x directly — no dereference needed
```

Two real, permanent constraints: a reference must be initialized the moment it's declared — there's no such thing as an uninitialized or null reference the way there's an uninitialized or null pointer — and once bound, it can never be rebound to refer to something else.

References vs. Pointers

CONCEPT	REFERENCE	POINTER
Syntax at use site	used exactly like the variable itself	requires * to dereference
Can it be null?	no	yes — NULL is valid
Can it be reassigned to something else?	no — bound permanently at initialization	yes — freely
Must be initialized immediately?	yes	no — can be declared, then assigned later

`cpp1-8` covers exactly when to reach for each in real code — this is just the first introduction.

`auto` Type Inference

`auto` lets the compiler deduce a variable's type from its initializer. C has no equivalent at all — every variable's type is always written explicitly. This is genuinely a point where C++ and Rust *agree* (Rust's `let` does the same thing), unlike most of the C track's own C-vs-Rust contrasts.

```
auto count = 5;      // deduced as int
auto name = "Alice"; // deduced as const char*
```

`const` in C++

Used far more pervasively and idiomatically in C++ than in C. A `const` variable's value cannot change after initialization — a full dedicated chapter on `const`-correctness comes later (`cpp2-8`); for now, just the basic qualifier.

```
const int max_size = 100; // cannot be reassigned
```

CONCEPT	C	C++
Boolean type	_Bool/bool since C99 — really an int	bool — a real, distinct primitive since the first standard
Alias for a variable	not available	references (&) — new in this course
Type inference	none — always explicit	auto — genuinely matches Rust's own let

USE AUTO WHERE IT HELPS, NOT EVERYWHERE

`auto` is genuinely useful for verbose or obvious types (iterator types especially, covered later) — but overusing it to the point where a reader can't tell a variable's type at a glance is a real, debated style tradeoff even within the C++ community.

A REFERENCE HAS NO "EMPTY" STATE — DECLARING ONE WITHOUT BINDING IT IS A COMPILE ERROR

Unlike a pointer, which C happily lets you declare and assign later (or leave uninitialized, a real risk of its own), a reference must be bound the instant it's declared. `int &ref;` with no initializer simply doesn't compile.

Coding Challenges

CHALLENGE 1

Declare an int variable, create a reference to it, modify the value through the reference, and print the original variable directly to confirm it changed — using cout, not printf.

View solution

CHALLENGE 2

Declare three variables using auto — an int literal, a double literal, and a string literal — and print each one's value along with, using typeid or a comment, what type auto actually deduced.

View solution

CHALLENGE 3

Explain why a reference cannot be null while a pointer can, tying your answer back to what a reference actually IS (an alias) rather than a separate object with its own address.

View solution

CHAPTER 2 QUICK REFERENCE

- `bool` / `true` / `false` — a genuine primitive and real literals, since C++'s first standard
- `int &ref = x;` — an alias for an existing variable, not a separate object
- References: no null state, no rebinding, must be initialized immediately — all unlike pointers
- `auto` — type inference from the initializer, genuinely matching Rust's own `let`
- `const` — used pervasively in idiomatic C++; full correctness rules come in `cpp2-8`

- **Next chapter:** functions and overloading — something C structurally cannot do at all

CHAPTER 3 OF 9

Functions & Overloading

COURSE 1 · CH 3

Functions & Overloading

Something plain C structurally cannot do — and the linker trick that makes it possible

`c1-5`'s functions had one name each, always. This chapter introduces the first genuinely new C++ capability that has no C equivalent at all — not a missing feature, a structural impossibility in C's own linking model.

Function Overloading

Multiple functions can share the same name, distinguished by their parameter types or count — the compiler picks the right one based on the actual arguments at each call site.

```
void print(int x)    { std::cout << "int: " << x << std::endl; }
void print(double x) { std::cout << "double: " << x << std::endl; }

print(5);      // calls print(int)
print(5.0);    // calls print(double)
```

C cannot do this at all. C compiles every function name directly to one plain, unmangled linker symbol — two functions named `print` in the same C program is simply a redefinition error, per `c2-5`'s own linking material.

Name Mangling, Briefly

C++ solves this by **encoding parameter types into the actual linker symbol name**. `print(int)` and `print(double)` become genuinely different symbols under the hood — something like `_Z5printi` and `_Z5printd`.

```
$ nm program.o | grep print
0000000000000000 T _Z5printi
0000000000000010 T _Z5printd
```

This directly extends `c2-5`'s own linker material: overloading isn't possible because the linker got smarter — it's possible because C++ generates *different names* for what looks, in source, like the same function name.

`extern "C"`

A real, practical need: calling a C library from C++, or being called from C code, requires plain, unmangled names. `extern "C"` tells the compiler to use C's own naming rules for specific declarations — this is exactly why standard C headers are internally wrapped in `extern "C"` when included from C++.

```
extern "C" {
    void legacy_c_function(int x);
}
```

Default Arguments

A genuinely new feature: parameters can have default values, letting a function be called with fewer arguments than it declares.

```
void greet(std::string name, std::string greeting = "Hello") {
    std::cout << greeting << ", " << name << std::endl;
}

greet("Alice");           // uses the default greeting
greet("Bob", "Hi");      // overrides it
```

C has no equivalent at all — the closest workaround is a variadic function, or several separately-named functions.

Overload Resolution Ambiguity

If a call could plausibly match more than one overload after implicit conversions, the compiler reports an **ambiguous call** error rather than guessing.

```
void print(int x);
void print(long x);

print(5L);   // fine — an exact match for print(long)
print('a');  // a char could convert to either int or long — genuinely ambiguous
```

CONCEPT	C	C++
Two functions, same name	**redefinition error**	**valid — overloading, resolved by parameter types**

CONCEPT	C	C++
Linker symbol names	plain function name	mangled — encodes parameter types
Fewer arguments than declared	not possible — variadic functions or multiple names only	default arguments

NAME MANGLING IS EXACTLY WHY EXTERN "C" EXISTS

Two of this chapter's own concepts connect directly: `extern "C"` is needed specifically because name mangling would otherwise make a C++-compiled function unreachable from C's own, unmangled linker expectations.

DEFAULT ARGUMENTS CAN ONLY BE OMITTED FROM THE RIGHT

Only trailing parameters can have defaults, and only trailing arguments can be skipped — a middle parameter can never be omitted while a later one is still provided. This is a real, common syntax constraint worth internalizing early.

Coding Challenges

CHALLENGE 1

Write two overloaded describe functions, one taking an int and one taking a std::string, each printing a different message. Call both from main and confirm the correct overload is selected each time.

View solution

CHALLENGE 2

Write a function power(int base, int exponent = 2) with a default argument, and call it once with both arguments and once with only the base, printing both results.

View solution

CHALLENGE 3

Explain precisely why C cannot support function overloading at all — tying your answer to how C's linker resolves symbol names — and why extern "C" is necessary when C++ code needs to interoperate with C.

View solution

CHAPTER 3 QUICK REFERENCE

- **Overloading** — same function name, different parameter types/count, resolved at the call site
- **Name mangling** — the compiler encodes parameter types into the real linker symbol, making overloading possible
- `extern "C"` — forces plain, unmangled C-style linkage for interop with C libraries/callers
- **Default arguments** — trailing parameters only; can't skip a middle one and provide a later one
- An ambiguous overload call is a compile error, never a silent guess
- **Next chapter:** classes and objects — the leap beyond plain structs

CHAPTER 4 OF 9

Classes & Objects

COURSE 1 · CH 4

Classes & Objects

The leap c2-1 explicitly foreshadowed — data and behavior, bundled together for the first time

`c2-1` named this chapter's whole subject directly: "one way C approximates method-like dispatch without a language feature for it at all." This chapter is that language feature.

From `struct` to `class`

`c2-1`'s C struct was pure data — no methods, ever; any function operating on it had to be written separately, taking a pointer as its first argument. A C++ **class** bundles data and methods together directly, inside the same definition, called with `object.method()` syntax instead of `function(&object)`.

A First Class

```
class Circle {
public:
    double radius;

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

Circle c;
c.radius = 5.0;
std::cout << c.area() << std::endl;
```

Compare directly against `c2-7`'s own hand-rolled `Circle` struct with a manually-wired function pointer. Same idea — the language now does the wiring for you.

`public` and `private` Access Control

A genuinely new concept: members can be hidden from outside code entirely. A `class`'s members are **private by default**; a `struct`'s are **public by default** — the *only* real difference between the two keywords in C++, worth naming explicitly since it's a common point of confusion.

```
class Account {
private:
    double balance; // inaccessible from outside the class

public:
    void deposit(double amount) { balance += amount; }
};
```

This is enforced by the compiler — unlike C, where a struct's members are always fully exposed, with zero protection of any kind.

Constructors

A special member function that runs automatically when an object is created — replacing the "manually call an init function" pattern C never even had a formal concept for. Multiple constructors are possible, reusing `cpp1-3`'s own overloading rules directly.

```
class Circle {
public:
    double radius;

    Circle(double r) { radius = r; } // runs automatically
};

Circle c(5.0); // constructed directly, no separate init call
```

Destructors

A special member function that runs automatically when an object is destroyed — a genuine preview of RAII, this course's own central chapter, coming next.

```
class Circle {
public:
    ~Circle() { std::cout << "Circle destroyed\n"; } // runs on scope exit
};
```

`this` — A Hidden Parameter

Inside a member function, `this` is a pointer to the object the method was called on — exactly the explicit `self` / `void *` parameter `c2-7` required passing by hand in plain C. The compiler now supplies it invisibly, every call.

CONCEPT	C (C2-1/C2-7)	C++ CLASS
Data + behavior	separate — struct, plus functions taking a pointer	bundled together directly
Hidden self/object parameter	passed explicitly, by hand	this — supplied automatically
Access control	none — every member always fully exposed	public/private, compiler-enforced
Initialization	a separate, manually-called init function, by convention	a constructor, run automatically

STRUCT AND CLASS ARE OTHERWISE IDENTICAL IN C++

Members, methods, and inheritance all work the same way on both — default access level is the only real difference. Many style guides reserve `struct` for pure-data types specifically as a signal to readers, purely by convention.

ACCESSING A PRIVATE MEMBER FROM OUTSIDE IS A COMPILE ERROR, NOT A WARNING

Unlike C, where every struct member is always reachable no matter what, attempting `account.balance` from outside `Account`'s own methods genuinely fails to compile — a real, enforced boundary, not a convention the compiler merely suggests.

Coding Challenges

CHALLENGE 1

Write a Rectangle class with private width and height members, a constructor taking both values, and a public area() method. Create an instance and print its area.

View solution

CHALLENGE 2

Add a destructor to the Rectangle class from Challenge 1 that prints a message when an instance is destroyed. Create an instance inside a nested scope (an inner {}) and observe when the destructor's message actually prints.

View solution

CHALLENGE 3

Explain precisely what the this pointer is standing in for, tying your answer directly back to c2-7's own hand-rolled vtable-style dispatch pattern and what parameter that pattern required the programmer to pass explicitly.

View solution

CHAPTER 4 QUICK REFERENCE

- A class bundles data and methods together — resolving what `c2-1` explicitly foreshadowed
- `class` — private by default; `struct` — public by default; otherwise identical
- A **constructor** runs automatically on creation; a **destructor** runs automatically on destruction — RAII's own preview
- `this` — the hidden object pointer, replacing `c2-7`'s explicit self/void* parameter
- Private-member access from outside the class is a genuine compile error, compiler-enforced
- **Next chapter:** Constructors, Destructors & RAII — C++'s own answer to C's manual malloc/free discipline

CHAPTER 5 OF 9

Constructors, Destructors & RAII

COURSE 1 · CH 5

Constructors, Destructors & RAII

C++'s own real answer to C's central safety tradeoff — invented roughly fifteen years before Rust's Drop

`c2-2` was the C track's own central chapter: manual `malloc` / `free`, with every allocation carrying real risk of a forgotten `free`. This chapter is C++'s answer — and, genuinely, it's the same answer Rust would later give.

RAII — Resource Acquisition Is Initialization

The core idea: tie a resource's lifetime directly to an object's lifetime. Acquire the resource in the constructor; release it in the destructor. When the object goes out of scope — through *any* exit path, including an early return or an exception — the destructor runs automatically, releasing the resource with zero remaining programmer effort at that point.

A Worked RAII Example — A Simple Owning Wrapper

```
class IntArray {
private:
    int *data;

public:
    IntArray(int size) {
        data = new int[size]; // acquire — in the constructor
    }

    ~IntArray() {
        delete[] data;        // release — in the destructor, always
    }
};
```

Compare this directly against `c2-2`'s own discipline: "every allocation needs a matching `free`, and the programmer must remember every single time." Here, allocate once in the constructor, and never have to remember to free again — the destructor handles it unconditionally, even during an exception unwind.

The Direct Rust Parallel

This is, genuinely, **exactly Rust's `Drop` trait** — invented roughly fifteen years earlier, via manual class-writing instead of a language-level trait system. Rust's own ownership model (`rust1-3`) and smart pointers (`rust2-2`) were historically compared to, and influenced by, this exact C++ idiom. A real, rare moment of genuine convergent evolution between the two languages, unlike most of the C track's own contrasts.

`new` and `delete`

C++'s own heap-allocation operators — a real, concrete improvement over C's `malloc`/`free`. `new` allocates *and* calls the constructor automatically; `delete` calls the destructor *and* frees the memory. Raw `malloc`/`free` know nothing about object lifecycles at all — just bytes.

```
Circle *c = new Circle(5.0); // allocates AND constructs
delete c;                    // destructs AND frees

int *arr = new int[10];   // array form — needs the matching array form below
delete[] arr;
```

Mismatching them — `new` with `delete[]`, or vice versa — is undefined behavior, the exact same category `c3-4` covered in full.

The Rule of Three, Previewed

A class managing a resource manually generally needs a destructor, a copy constructor, *and* a copy assignment operator together — or copies of the object risk a double-free exactly like `c2-3`'s own bug catalog. Full depth arrives in `cpp2-8`; this is just the forward pointer.

Exception Safety, Briefly

RAII's other major payoff, previewed ahead of `cpp2-6`'s own Exception Handling chapter: because destructors run on *any* scope exit — including an exception unwinding through the call stack — RAII-managed resources are cleaned up automatically even when an error is thrown mid-function. C's manual cleanup pattern has no equivalent mechanism at all; C has no exceptions and no automatic stack unwinding.

CONCEPT	C (C2-2)	C++ RAII	RUST
Resource release	manual free() — programmer must remember	automatic — destructor on scope exit	automatic — Drop on scope exit

CONCEPT	C (C2-2)	C++ RAII	RUST
Cleanup on an error mid-function	no mechanism — manual only	automatic — destructors run during unwinding	automatic — Drop runs even on panic unwind
Allocation + initialization	two separate manual steps (malloc, then init)	one step — new calls the constructor	one step — the constructor pattern is the norm

PREFER READY-MADE RAII WRAPPERS OVER HAND-ROLLED ONES

This chapter's `IntArray` is a teaching device. Course 2's smart pointers (`unique_ptr` / `shared_ptr`) are the standard, battle-tested RAII wrappers real code should reach for — reinventing them by hand is rarely the right call outside a learning exercise.

A CONSTRUCTOR THAT THROWS PARTWAY THROUGH SKIPS THAT OBJECT'S DESTRUCTOR

If a constructor throws before finishing, the object was never fully constructed — its destructor does *not* run for it, since construction never completed. A genuine, subtle edge case worth knowing exists, covered properly once `cpp2-6` introduces exceptions in full.

Coding Challenges

CHALLENGE 1

Write a class that acquires a resource (e.g. allocates an int array with new[]) in its constructor and releases it with delete[] in its destructor. Create an instance inside a nested scope and print a message from both the constructor and destructor to observe the order they run in.

View solution

CHALLENGE 2

Allocate an object with new, then intentionally never call delete on it. Explain, referencing c2-2's own material, what class of bug this is and why RAII alone doesn't protect against this specific mistake.

View solution

CHALLENGE 3

Explain precisely why RAII is considered the same underlying idea as Rust's Drop trait, and name the one concrete mechanical difference between how each language actually triggers the cleanup code.

View solution

CHAPTER 5 QUICK REFERENCE

- **RAII** — acquire in the constructor, release in the destructor, tied to object lifetime
- `new` / `delete` — allocate+construct / destruct+free in one step, unlike raw malloc/free
- `new[]` must be matched with `delete[]` — mismatching is undefined behavior
- Destructors run on *any* scope exit, including exception unwinding — C has no equivalent mechanism at all
- This is genuinely the same idea as Rust's `Drop`, roughly fifteen years earlier
- **Next chapter:** operator overloading — giving custom types +, ==, and <<

CHAPTER 6 OF 9

Operator Overloading

COURSE 1 · CH 6

Operator Overloading

The mechanism every single cout statement since Chapter 1 has secretly been using all along

Every `std::cout << ...` line since `cpp1-1` has quietly relied on a mechanism this chapter finally names directly.

Why Operator Overloading

A `Vector2D` class: adding two vectors should read naturally as `v1 + v2`, not `add(v1, v2)`. C has no equivalent mechanism at all — every "operation" on a struct must be a plainly-named function call, always.

Overloading `+`

```
class Vector2D {
public:
    double x, y;

    Vector2D operator+(const Vector2D &other) {
        return Vector2D{x + other.x, y + other.y};
    }
};

Vector2D v1{1, 2}, v2{3, 4};
Vector2D sum = v1 + v2; // reads naturally, calls operator+
```

Written as a member function, `this` supplies the left operand implicitly; `other` is the right one.

Overloading `==`

```
bool operator==(const Vector2D &other) {
    return x == other.x && y == other.y;
}
```

Returns a genuine `bool` — `cpp1-2`'s own real boolean primitive, put to direct use.

Overloading `<<` for Output

A real exception to the pattern so far: `operator<<` must be a **free function**, not a member — the left operand is `std::ostream&`, not the class itself, so it can't be written as a member of `Vector2D`. A `friend` declaration lets the free function reach the class's private members.

```
class Vector2D {
private:
    double x, y;
    friend std::ostream& operator<<(std::ostream& os, const Vector2D &v);
};

std::ostream& operator<<(std::ostream& os, const Vector2D &v) {
    os << "(" << v.x << ", " << v.y << ")";
    return os;
}
```

Which Operators Can/Can't Be Overloaded

Most can — arithmetic, comparison, stream, subscript `[]`, function-call `()`, and more. A small, fixed set genuinely cannot: `::`, `.`, `.*`, `?:`, `sizeof` — a real exclusion list, not an arbitrary restriction.

The Real Payoff — This *Is* What `cout <<` Already Uses

Here's the reveal: `std::cout << 5` is itself a call to `operator<<(ostream&, int)` — an overload of exactly this same operator, just for a built-in type instead of a custom one. Every `cout` statement since Chapter 1 has been using this mechanism the entire time.

CONCEPT	C	C++
Custom "addition" for a type	`a plainly-named function — add(a, b)`	`operator+ — a + b reads naturally`
Left operand isn't the class	`n/a — no operator concept at all`	`requires a free function, not a member`

PRESERVE THE OPERATOR'S NATURAL MEANING

Overload `+` for something genuinely addition-like — never for an unrelated operation just because the syntax happens to be convenient. A real style principle, sometimes called the principle of least astonishment.

OPERATOR<< CAN'T BE A MEMBER — A COMMON BEGINNER MISTAKE

Attempting to write `operator<<` as a member function of the class itself is a genuine, common early confusion — the left-hand operand is `ostream&`, which isn't the class being extended, so member-function syntax simply doesn't fit.

Coding Challenges

CHALLENGE 1

Write a Vector2D class with x and y members, overload operator+ as a member function, and print the sum of two vectors' x and y components after adding them.

View solution

CHALLENGE 2

Add operator== to the Vector2D class from Challenge 1, and test it against two vectors with identical components and two with different ones, printing the boolean result of each comparison.

View solution

CHALLENGE 3

Explain precisely why operator<< for a custom class cannot be written as a member function, tying your answer to which operand is on the left-hand side of the << expression.

View solution

CHAPTER 6 QUICK REFERENCE

- `operator+` / `operator==` — usually member functions, `this` as the implicit left operand
- `operator<<` — must be a free function, since the left operand is `ostream&`, not the class
- `friend` — lets a free function access a class's private members
- A fixed, small set of operators genuinely cannot be overloaded — `::`, `.`, `.*`, `?:`, `sizeof`
- `std::cout << 5` already uses this exact mechanism — an `operator<<` overload for `int`
- **Next chapter:** inheritance and polymorphism — making real the vtable mechanism `c2-7` built by hand

CHAPTER 7 OF 9

Inheritance & Polymorphism

COURSE 1 · CH 7

Inheritance & Polymorphism

c2-7's own hand-rolled function-pointer struct, made real by the language itself

`c2-7` ended by naming this chapter directly: what a struct of function pointers, wired up by hand, actually becomes once C++ does it for you. This is that reveal, in full.

Base and Derived Classes

```
class Shape {
public:
    double area() { return 0; }
};

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

Without `virtual` — The Problem

```
Shape *s = new Circle{5.0};
s->area(); // calls Shape::area() — NOT Circle::area(), even though s points at a Circle
```

A genuine, surprising gotcha: calling `area()` through a `Shape *` pointing at a `Circle` calls *Shape's* own `area()`, not `Circle's`. This is **static binding** — resolved at *compile time*, based on the pointer's declared type, not the actual object it points to.

`virtual` — Making Dispatch Real

```
class Shape {
public:
    virtual double area() { return 0; }
};

// now, with the same Shape* s pointing at a Circle:
s->area(); // correctly calls Circle::area()
```

Adding `virtual` changes everything: dispatch now resolves at **runtime**, based on the actual object's real type. This is exactly `c2-7`'s own hand-built mechanism, made real by the language.

The vtable, Named Directly

Under the hood, a class with virtual functions gets a hidden **vtable** — a table of function pointers, exactly `c2-7`'s own hand-rolled `Circle` struct with its own function-pointer member. Every object of such a class carries a hidden pointer to its class's vtable. Calling a virtual function through a base pointer looks up the real function through this table, at runtime — literally the same `shape->area(shape)` code `c2-7` wrote by hand, now generated automatically.

`override` and Pure Virtual Functions

`override` (C++11+) is a safety check confirming a derived function genuinely overrides a base virtual — catching typos or signature mismatches as compile errors instead of silently creating an unrelated function.

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

`= 0` marks a **pure virtual function** — the class becomes abstract, unable to be instantiated directly, requiring every derived class to provide its own implementation.

```
virtual double area() = 0; // Shape itself can no longer be instantiated
```

CONCEPT	C (C2-7)	C++
Dispatch table	hand-written struct of function pointers	vtable — generated automatically
Wiring a function pointer correctly	the programmer's own responsibility, unchecked	the compiler builds it, guaranteed correct
Forcing an implementation to exist	no mechanism — a NULL entry is UB when called	= 0 (pure virtual) — a compile-time requirement

GIVE A VIRTUAL BASE CLASS A VIRTUAL DESTRUCTOR

Any class with virtual functions should make its destructor virtual too — otherwise deleting a derived object through a base pointer skips the derived class's own destructor, a real and genuinely common source of leaks.

FORGETTING VIRTUAL IS A SILENT BUG, NOT A COMPILE ERROR

Code compiles fine without `virtual`, calls the wrong function, and gives no warning by default. C, by contrast, never pretends to have automatic dispatch at all — `c2-7`'s own struct-of-function-pointers approach fails loudly (a NULL call) rather than silently calling the wrong thing.

Coding Challenges

CHALLENGE 1

Write a Shape base class with a non-virtual area() returning 0, and a Circle derived class overriding it. Call area() through a Shape* pointing at a Circle and report what actually gets called.

View solution

CHALLENGE 2

Add virtual to Shape::area() from Challenge 1 (and override to Circle::area()), then run the identical Shape* call again. Report how the result changed and why.

View solution

CHALLENGE 3

Explain precisely what the vtable is, and why calling a virtual function through a base-class pointer is described as "the exact code c2-7 built by hand, generated automatically" — name the specific piece c2-7 wrote manually that the vtable now replaces.

View solution

CHAPTER 7 QUICK REFERENCE

- `class Derived : public Base { }` — inherits members and methods
- Without `virtual` — static binding, resolved by the pointer's declared type at compile time
- With `virtual` — dynamic binding, resolved by the actual object's type at runtime

- The **vtable** — a hidden, compiler-generated table of function pointers, exactly `c2-7`'s own hand-rolled mechanism
- `override` — compile-time safety check; `= 0` — pure virtual, makes a class abstract
- Always give a base class with virtual functions a virtual destructor
- **Next chapter:** references vs. pointers, in depth — when to reach for each

CHAPTER 8 OF 9

References vs. Pointers, In Depth

COURSE 1 · CH 8

References vs. Pointers, In Depth

C++ solves C's raw-pointer ergonomics problem — but not, on its own, C's safety problem

`cpp1-2` introduced references structurally. This chapter is the practical guidance — when to reach for each — and the honest comparison against what Rust's own references actually guarantee.

Recap: The Structural Differences

PROPERTY	REFERENCE	POINTER
Can be null	no	yes
Can be reassigned	no — bound permanently	yes
Must be initialized immediately	yes	no

When to Use a Reference

Primarily function parameters: passing a large object by reference avoids an expensive copy while still letting the function read or modify the caller's actual object. Return types too — but never return a reference to a local variable; that's the exact reference-shaped version of `c1-7`'s own dangling-pointer warning.

`const` Reference — Pass-By-Reference Without Copying, Safely

`const T&` is the real, idiomatic C++ default for passing anything nontrivial into a function — it avoids the copy plain pass-by-value would make, while `const` guarantees the function can't accidentally modify the caller's object. Genuinely the most common parameter-passing pattern in real C++ code.

```
void print_name(const std::string &name) {
    std::cout << name << std::endl; // no copy made, and name can't be modified here
}
```

When to Use a Pointer

- "No object" (null) is a genuinely valid state to represent
- The thing being pointed to needs to be reassigned to point elsewhere later
- Pointer arithmetic — iterating a raw array, exactly `c1-6` / `c1-7` 's own territory
- Ownership semantics via smart pointers — previewed for Course 2

Contrasted With Rust's Borrow Rules

Rust's `&` / `&mut` references are checked at *compile time* by the borrow checker for aliasing rules — you cannot have a mutable reference and any other reference to the same data alive simultaneously. C++ references have **none** of these guarantees enforced at all: a `const T&` and a plain `T&` to the same object can coexist freely, and nothing stops two non-`const` references from both mutating the same data unsafely — including from two different threads, tying directly to `cpp3-4` 's own concurrency chapter. This is a real, genuine gap this course's own framing has been building toward: C++ references solve C's raw-pointer *ergonomics* problem — no dereference syntax, no null, no rebinding — but not its *safety* problem.

CONCEPT	C++ REFERENCE	C++ POINTER	RUST REFERENCE
Aliasing rules enforced?	no	no	yes — compile-time borrow checking
Mutable + shared reference coexisting	allowed, unchecked	allowed, unchecked	compile error
Ergonomics vs. a raw pointer	better — no null, no dereference syntax	n/a	better, and safety-checked

"CONST REFERENCE UNLESS YOU HAVE A REASON NOT TO" IS CLOSE TO A UNIVERSAL DEFAULT

For any parameter type larger than a machine word, passing by `const T&` is the idiomatic starting point in real C++ — deviate only with a specific reason (needing to modify the caller's object, or genuinely wanting a cheap copy for a small type).

RETURNING A REFERENCE TO A LOCAL VARIABLE DANGLES — WITH NOTHING MARKING IT AS SUCH

The function's local storage is gone the instant it returns; the reference is left referring to memory that's already invalid, with no compiler error and no visible sign anything is wrong at the call site — exactly `c1-7` 's own dangling-pointer warning, now in reference form.

Coding Challenges

CHALLENGE 1

Write a function that takes a std::string by const reference and prints its length. Call it with a string literal and confirm no copy-related side effects are needed to make it work.

View solution

CHALLENGE 2

Write a function that deliberately returns a reference to one of its own local variables. Explain what happens when the caller tries to use the returned reference, tying your answer to c1-7's own dangling pointer material.

View solution

CHALLENGE 3

Explain precisely what Rust's borrow checker would reject that C++ freely allows, using a concrete scenario involving a mutable reference and a second reference to the same data — and why this specific gap is described as C++ solving pointer "ergonomics" but not "safety."

View solution

CHAPTER 8 QUICK REFERENCE — ALMOST COURSE 1 COMPLETE

- Use a reference for function parameters/return values tied to an object that already exists elsewhere
- `const T&` — the idiomatic default: avoids a copy, guarantees no mutation
- Use a pointer when null is valid, reassignment is needed, or for raw pointer arithmetic
- Never return a reference to a local variable — the exact dangling-pointer bug, in reference form
- C++ references have **no** compile-time aliasing guarantees — a real, genuine gap vs. Rust's borrow checker
- **Next chapter:** namespaces and the standard library tour — closing Course 1

CHAPTER 9 OF 9

Namespaces & the Standard Library Tour

COURSE 1 · CH 9 — COURSE FINALE

Namespaces & the Standard Library Tour

Why every example since Chapter 1 has written std:: — and a first map of what Course 2 actually builds

Every single example since `cpp1-1` has written `std::cout`, never just `cout`. This closing chapter finally explains why — and previews everything Course 2 is about to build on top of it.

The Global Namespace Problem

C has exactly one flat namespace for every function and variable name across an entire program. Two libraries both defining a function called `process` genuinely collide — a real, historical C problem with no built-in solution.

`std::` — The Standard Library's Own Namespace

This is why every example so far has written `std::cout`: C++'s entire standard library lives inside a namespace called `std`, specifically so it never collides with anything the programmer names themselves.

Declaring Your Own Namespace

```
namespace myapp {
    void process() { /* ... */ }
}

myapp::process(); // fully qualified — never collides with another library's own process()
```

`using namespace` — Convenient But Risky

`using namespace std;` brings every `std::` name into scope unqualified — genuinely convenient for small examples, but it reintroduces exactly the collision risk namespaces exist to prevent.

NEVER WRITE "USING NAMESPACE" IN A HEADER FILE

A header's `using namespace` directive affects *every single file* that includes it, transitively — a real, well-known C++ style rule, since it silently reintroduces global-namespace collision risk across an entire project, not just the one file that wrote it.

A Standard Library Tour, Previewing Course 2

A quick map of what's coming, not the full depth yet:

- **std::string** — a real, growable string type, unlike C's own raw `char` arrays
- **std::vector** — the direct answer to `c3-2`'s own "C has no standard containers" gap
- **std::map** — a built-in key-value structure, replacing `c3-2`'s own hand-rolled hash table

Course 2's own dedicated chapters cover each in full — this is just naming what's ahead.

Closing Course 1

From `cpp1-1`'s compile model through `cpp1-8`'s honest safety-gap admission, one throughline: C++ adds real capability on top of C — overloading, classes, RAII, polymorphism, safer references — without adding Rust's own compile-time safety guarantees. That tension is exactly what Course 3's concurrency and UB chapters make fully concrete. Course 2, starting next, is about the practical, everyday tools — templates, the STL, smart pointers, move semantics — built directly on this foundation.

CONCEPT	C	C++
Name collisions across libraries	`a real, unsolved problem — one flat namespace`	`namespaces — names grouped, no collision`
The standard library's own names	`global — printf, malloc, etc.`	`inside std:: — cout, string, vector`

QUALIFY NAMES EXPLICITLY IN REAL PROJECTS

Writing `std::` explicitly, rather than reaching for `using namespace std;` even in a source file, is the safer default in any codebase beyond a small standalone example — it costs a few extra characters and avoids the collision risk entirely.

Coding Challenges

CHALLENGE 1

Declare a namespace called geometry containing a function area(double radius) computing a circle's area. Call it fully qualified as geometry::area(...) and print the result.

View solution

CHALLENGE 2

Declare two separate namespaces, each containing a function named distance with a different implementation. Show that both can be called unambiguously when fully qualified, and explain what would happen if both were brought into scope with using namespace and called as plain distance(...).

View solution

CHALLENGE 3

Explain why "using namespace X;" in a header file is considered worse practice than the identical directive in a single source (.cpp) file, tying your answer to how #include actually works per c2-4's own material.

View solution

CHAPTER 9 QUICK REFERENCE — COURSE 1 COMPLETE

- C has one flat namespace for every name; C++ groups names into namespaces to avoid collisions
- `std::` — the entire standard library's own namespace, why every example writes it explicitly
- `using namespace` — convenient, but reintroduces collision risk; never in a header file
- Course 2 preview: `std::string`, `std::vector` (resolving `c3-2`'s container gap), `std::map`
- **Course 1 complete.** Course 2 builds templates, the STL, smart pointers, and move semantics directly on this foundation