The C++ Interview Companion

VOLUME ONE

MODERN C++ — C++11 THROUGH C++23

The C++ Interview Companion

Complete, Runnable Answers to the Questions That Actually Get Asked

VOLUME ONE

Prepare with it, Interview with it

Jeganathan Swaminathan

TEKTUTOR

Hosur, India

The C++ Interview Companion — Volume 1
Complete, Runnable Answers to the Questions That Actually Get Asked

Published by TekTutor, Hosur, Tamil Nadu, India.
jegan@tektutor.org

First Edition

About the code and the standard. This book covers modern C++ through C++23. Language and library features are explained across the standards in which they appear, and each version-specific feature is tagged in the text with the standard that introduced it.

The programs in Volume 1 target C++23 and were compiled and run with g++ 16.1, each producing zero warnings under `-std=c++23 -Wall -Wextra` before publication. A small number of examples use threading and additionally require `-pthread`, as noted in their build lines. Every listing was verified against its printed output by an automated test harness on g++ 16.1 before publication.

Disclaimer. The information and code in this book are provided on an "as is" basis, without warranty of any kind, express or implied. While every effort has been made to ensure accuracy, the author assumes no responsibility for errors or omissions, or for damages resulting from the use of the information or code contained herein. Compilers, libraries, and language standards evolve; verify behavior in your own environment before relying on it in production.

---

to my wife Meenakshi,

sons Nitesh and Sriram

Contents

Preface

why this book exists, and how to get the most from it

The best way to understand a piece of C++ code is to watch it run, break it on purpose, and see what happens. That is how this whole book works. Every answer comes with a small, complete program you can compile and change yourself, so you are never taking my word for anything. That habit matters most in an interview. Memorized facts have a way of falling apart the moment someone asks you to write the code or explain why it is true, but understanding you have seen with your own eyes stays with you, and that is the kind this book is built to give you.

I have sat on both sides of the interview table. For twelve years I built and shipped software in C++, as an individual contributor writing applications, then as someone who led teams, held technical leadership roles, and worked as an architect. I have hired engineers and served on interview panels, and I have sat in the candidate's seat and answered the questions myself. For the fourteen years since, I have taught this language full time, to more than five thousand software professionals across the world, and I consult for software companies who need help with the code they are shipping. That is twenty-six years in this industry, spent first writing the code and then explaining it, and never entirely stopping either.

I mention that number for a reason. When you teach the same language to five thousand engineers, you no longer have to guess which parts are hard, because you have seen them struggle with the same things again and again. The same questions come up, the same explanations need fixing, and the same gaps show up when the pressure of an interview is on. The questions in this book are not the ones I find interesting. They are the ones that kept returning, year after year, in room after room. The consulting work keeps that honest: it puts me back inside real codebases, under real deadlines, so what you read here reflects the C++ that companies actually ship, not a tidy classroom version of it. This book comes out of all of it: the code I wrote and reviewed, the candidates I evaluated, the answers I gave when I was the one being evaluated, and the engineers I have taught since. What is collected here is what separates someone who has read about C++ from someone who understands it. I wrote the book I wished I could hand to every candidate.

01 Who this book is for

This book is ideal for readers who fall into three groups. The first group is the candidates preparing for a C++ interview, at any level from a first job to a senior role. The second group is the interviewers who want a reliable, precise source for the questions worth asking and the answers worth hearing. The third group is the engineers who simply want to understand C++ more deeply, and find that a sharp question opens up a topic faster than a chapter full of theory or philosophy. All three groups need the same thing: an explanation that is correct, stays current, and is backed by code that actually compiles on the latest compiler.

You should already know basic C++ syntax: how to write a function, a class, a loop. You do not need to be an expert. The book starts from object-oriented foundations and builds steadily, so a competent beginner can read it front to back, while an experienced engineer can jump to the parts that matter for a specific interview. Where a topic depends on an earlier one, a cross-reference points back rather than repeating the material.

02 What makes this book different

> **THE CORE PROMISE**
>
> Every code example in this book is a complete, compilable, runnable program. Each one was compiled with `g++ -std=c++23 -Wall -Wextra` and produced zero warnings before it reached the page, and each shows the exact command to build it and the output it produces. There are no fragments that "would work if you filled in the rest," and no code that only looks correct.

That promise shapes everything. When a topic involves undefined behavior, the dangerous form is shown only as a comment and the live code demonstrates the safe pattern, so nothing you copy from this book can quietly break. When an answer turns on the exact wording of a rule, the example is built to make that rule visible in its output. The goal is that you never have to take my word for granted: you can run it and see it for yourself.

The style is deliberately conversational. Each chapter opens with the blunt question a real person would ask, then gives a direct answer first and the explanation second, the way you would want it under interview pressure. Recurring callouts flag the things that matter most: an *Interviewer's Angle* box explains what a strong answer sounds like and what distinguishes it from a weak one; a *New term* box defines each acronym the first time it

appears; and short notes catch the mistakes that trip people up. A *From the other side of the table* box appears now and then, sharing what I have learned from years of interviewing and being interviewed, for both the candidate and the interviewer. The book is carefully crafted for readers whose first language need not be English: it avoids hard-to-understand idioms and uses simple vocabulary and simple sentences.

03 How the book is organized

Volume 1 moves from foundations to modern practice across fifty-five parts. It begins with object-oriented concepts and the differences between C and C++, then works through object construction and lifetime, control flow and functions, polymorphism and virtual functions, inheritance and class relationships, and the rules of access, declarations, and namespaces. From there it covers data types and memory layout, the hardware-facing corners of the language, concurrency foundations, the memory bugs that smart pointers solve, and templates and concepts.

The later parts turn from language mechanics to judgment: the SOLID principles, design patterns and the modern C++ idioms that often replace them, code smells and refactoring, then the computer-science foundations of data structures and algorithm analysis. Volume 1 closes with the C++17 features that reshaped everyday code, and a rapid-fire round of the comparison questions interviewers love. The code throughout is written for and compiles under C++23.

Volume 2 goes where the day-to-day work happens: the standard library in depth, concurrency and coroutines, file and network I/O, testing, logging, debugging, and performance tuning. It covers C++20 and C++23 in full and closes with a look ahead to C++26. Together the two volumes take you from the language itself to everything you build on top of it.

Questions are numbered by chapter, as **Chapter.Question**, so 51.5 is the fifth question in Chapter 51. Adding a question to one chapter never renumbers another. Each answer gives the direct rule first, then the explanation, then the runnable example, and points to related questions where they help.

04 How to use this book

If you are preparing for an interview, read the direct answer and run the example for each question in the parts that match the role. Do not just read the code; compile it, change a line, and see what breaks. The understanding that survives an interview comes from having watched the behavior yourself, not from having read about it. When a question has

an *Interviewer's Angle* box, study it: it is the difference between an answer that is correct and one that is convincing.

If you are an interviewer, use the questions as a calibrated set and the answers as a scoring guide. The *Interviewer's Angle* notes describe what to listen for, so you can tell a memorized answer from a real one.

> **RUNNING THE CODE**
>
> The examples target C++23 and were verified with g++ 16.1 on Linux. To build any example, save it to a `.cpp` file and run the command shown beneath it, for instance `g++ -std=c++23 -Wall -Wextra example.cpp -o demo && ./demo`. A small number of examples use threading and need `-pthread`; those show the flag in their build line. Any recent compiler with C++23 support will work.

05 A note on correctness

C++ is a large language with many common mistakes to make, and it keeps changing. I have worked hard to make every statement in this book accurate and every program correct, and the fact that all of them compile cleanly is part of that effort. Still, compilers differ, standards evolve, and errors survive even careful review. Where a feature is specific to a standard version, the text says so. If you find a mistake, it is mine, and I would rather know than not. You can reach me at jegan@tektutor.org.

My thanks go to the colleagues I built software with in my engineering years, who taught me how C++ behaves when real systems depend on it, and to the many engineers in my training rooms since, whose questions, confusions, and hard-earned insights shaped every part of this book. You taught me which explanations work and which only sound good. This book is, in a real sense, the answer I have been refining with you all along.

Jeganathan Swaminathan
Founder, TekTutor
Hosur, Tamil Nadu, India
https://www.tektutor.org
jegan@tektutor.org

Chapter - 1

Thinking in Objects

object-oriented programming, without the hand-waving

You already think in objects. Your phone is an object. It has state (battery level, brightness, the apps installed) and it has behavior (make a call, take a photo, ring). C++ lets you build software the same way you already divide up the world in your mind. This chapter is about that shift, from writing steps to modeling things. Once it clicks, a huge amount of C++ stops feeling like syntax and starts feeling like design.

Every program in this chapter is complete and runnable. Copy it into a `.cpp` file and build it with the command shown under each one. All examples here compile with any C++23 compiler (g++, clang++, or MSVC).

SOMEBODY'S GOTTA ASK

"I've been writing working code for years without any of this object stuff. Why should I care about OOP?"

Fair question. You can absolutely write working code without it. But as programs grow, the code that isn't organized around objects tends to turn into a mess where every part depends on every other part. OOP is a way to draw walls: each object minds its own data and exposes a small set of things it can do. Those walls are what keep a 200,000-line codebase from becoming an unmanageable mess.

1.1 What does object-oriented programming actually mean, beyond the buzzwords?

> **Short answer:** OOP means you build your program out of *objects*, self-contained bundles of data (state) and the functions that operate on that data (behavior), instead of out of free-floating functions that pass data around between them.

Here's the mental switch. In procedural code, you have data over *here* and functions over *there*, and you pass the data into the functions:

LISTING 1.1

```
// Procedural style: data in a struct,
// behavior in free functions.
#include <iostream>

struct Phone {
    int  battery;
    bool ringing;
};

// behavior lives apart from the data, passed in by reference
void chargePhone(Phone& p, int amount) { p.battery += amount; }
void ringPhone(Phone& p)               { p.ringing = true; }

int main() {
    Phone p{50, false};    // data here
    chargePhone(p, 30);    // behavior operates from outside
    ringPhone(p);
    std::cout << "battery=" << p.battery
              << " ringing=" << std::boolalpha
              << p.ringing << "\n";
    return 0;
}
```

```
$ g++ -std=c++23 -Wall -Wextra c1_q1_proc.cpp -o demo && ./demo
```

```
battery=80 ringing=true
```

In object-oriented code, the data and the behavior live *together*. The phone knows how to charge itself:

LISTING 1.2

```
// Object-oriented style: data and behavior together in a class.
#include <iostream>

class Mobile {
public:
    // behavior lives WITH the data
    void charge(int amount) { battery_ += amount; }
    void ring()             { ringing_ = true; }
    void status() const {
        std::cout << "battery=" << battery_
```

```
                  << " ringing=" << std::boolalpha
                  << ringing_ << "\n";
    }
private:
    int  battery_ = 50;    // in-class initializer (C++11)
    bool ringing_ = false;
};

int main() {
    Mobile myPhone;
    myPhone.charge(30);    // "phone, charge yourself by 30"
    myPhone.ring();
    myPhone.status();
    return 0;
}
```

```
$ g++ -std=c++23 -Wall -Wextra c1_q1_oo.cpp -o demo && ./demo
```

```
battery=80 ringing=true
```

BRAIN POWER

Read `myPhone.charge(30)` out loud. Now read `chargePhone(myPhone, 30)`. Which one sounds like you're telling the phone what to do, and which sounds like you're operating on it from the outside? That difference in feel is the whole idea.

That's it. That's the core of OOP. Everything else, encapsulation, abstraction, inheritance, polymorphism, is a refinement of this one move: **put the data and the behavior that belongs to it in the same box, and let the box protect itself.**

INTERVIEWER'S ANGLE

Opening with "what is OOP" is rarely about the definition, it's a warm-up that sets the interview's depth. A textbook recital ("encapsulation, inheritance, polymorphism, abstraction") tells the interviewer to keep digging for real understanding. Tie the idea to a concrete benefit, keeping a large codebase from turning into an unmanageable mess, and you signal experience, which often earns you harder, more interesting questions instead of a checklist.

1.2 What are the four pillars of OOP, and why does each one matter in real code?

> **Short answer:** Encapsulation, abstraction, inheritance, and polymorphism. They aren't just words to memorize for an interview. Each one solves a real problem that shows up in every codebase, and each is a tool for solving it.

Let's ground all four in one product line you already know: mobile phones.

Encapsulation

"Who's allowed to touch this data?"

Your `Mobile` hides `battery_` behind `private`. Nobody can set it to -999 from outside. The object guards its own state.

Abstraction

"What does it do, not how?"

You call `myPhone.ring()`. You don't care whether that fires a speaker driver or a haptic motor. The *what* is exposed; the *how* is hidden.

Inheritance

"Is this a special kind of that?"

An `iPhone17` is a kind of `Mobile`. It reuses everything a phone does, then adds or changes what makes it an iPhone.

Polymorphism

"Same command, different behavior?"

You tell any phone to `ring()`. The iPhone plays its marimba, the Nothing Phone plays its glyph tone. One command, many behaviors.

WATCH IT

People love to recite the four pillars in interviews and then write code that uses none of them. The pillars only count when they change how your code is *structured*. We'll come back to each one with real `Mobile` code in later chapters, so you see them work, not just hear their names.

1.3 What is the difference between a class and an object?

> **Short answer:** A class is the blueprint. An object is a thing built from that blueprint. One class, many objects.

The Nothing Phone (3a) Pro has a design spec: dimensions, materials, what buttons it has, what it does when you press them. That spec is the **class**. The actual phone in your pocket, with *your* battery level and *your* photos, is an **object**. The factory produces millions of objects from one spec.

LISTING 1.3

```
// One class (the blueprint) makes many objects,
// each with its own independent data.
#include <iostream>

class Mobile {
public:
    void charge(int amount) { battery_ += amount; }
    int  getBattery() const { return battery_; }
private:
    int battery_ = 50;     // in-class initializer (C++11)
};

int main() {
    Mobile jegansPhone;    // object #1, its own battery_
    Mobile sriramsPhone;   // object #2, a DIFFERENT battery_

    jegansPhone.charge(40);   // only object #1 changes

    std::cout << "jegan's phone:  "
              << jegansPhone.getBattery()  << "\n";
    std::cout << "sriram's phone: "
              << sriramsPhone.getBattery() << "\n";
    return 0;
}
```

```
$ g++ -std=c++23 -Wall -Wextra c1_q3.cpp -o demo && ./demo
```

```
jegan's phone:  90
sriram's phone: 50
```

SHARPEN YOUR PENCIL

If `Mobile` is a class and `jegansPhone` is an object, what is `iPhone17`? (Hint: it's a *blueprint that borrows from another blueprint*.) We'll confirm the answer when inheritance arrives in Chapter 11.

1.4 What is encapsulation, and how is it different from just making data private?

> **Short answer:** Making data `private` is the *mechanism*. Encapsulation is the *goal*: keeping an object in a valid state by controlling every way its data can change.

Marking a field `private` and then adding a `getBattery()` and `setBattery()` that do nothing but read and write it isn't encapsulation. You've just added ceremony. Real encapsulation means the object *enforces its own rules*.

LISTING 1.4

```
// Real encapsulation: the object enforces its own
// invariant (battery stays between 0 and 100).
#include <algorithm>
#include <iostream>

class Mobile {
public:
    // charge() protects the rule instead of
    // blindly writing the field
    void charge(int amount) {
        if (amount < 0) return;      // reject nonsense
        battery_ = std::min(battery_ + amount, 100);
    }
    int getBattery() const { return battery_; }
private:
    int battery_ = 50;
};

int main() {
    Mobile phone;
    phone.charge(70);      // 50+70 would be 120; capped at 100
    std::cout << "after charging 70: "
              << phone.getBattery() << "\n";
    phone.charge(-5);      // rejected
    std::cout << "after charging -5: "
```

```
                << phone.getBattery() << "\n";
    return 0;
}
```

```
$ g++ -std=c++23 -Wall -Wextra c1_q4.cpp -o demo && ./demo
```

```
after charging 70: 100
after charging -5: 100
```

> **NEW TERM**
>
> **Invariant**: a rule about an object's data that must always hold true, for as long as the object exists. "A `Mobile`'s battery is always between 0 and 100" is an invariant. Encapsulation is how an object guarantees its invariants no matter who calls it. This word comes back often, class design, exception safety, and container internals all depend on it.

If any outside code could set the battery to 250, you couldn't trust that number anywhere in your program. That's why the object, and only the object, gets to change its own data.

> **INTERVIEWER'S ANGLE**
>
> The encapsulation question separates people who see it as "make fields private" from those who understand it as protecting invariants. If your answer mentions that the object guarantees its own valid state, not just that data is hidden, you signal design maturity. Interviewers often follow up by asking for an example where a public setter breaks encapsulation; having one ready (a setter that skips validation) shows you've thought past the textbook.

1.5 What is abstraction, and how is it different from encapsulation?

> **Short answer:** Encapsulation hides the *data*. Abstraction hides the *complexity*. They work together but they answer different questions.

People mix these up constantly, so here's the clean split, using the same phone:

Encapsulation asks	Abstraction asks
"Who can see and change `battery_`?"	"What does the user of a `Mobile` need to know about?"
Answer: nobody, except through `charge()`.	Answer: they call `ring()`, and never learn about speaker drivers, haptics, or ringtone files.
It's about *access control*.	It's about *presenting a simple surface over a complex machine.*

SOMEBODY'S GOTTA ASK

"So is a public method with a good name abstraction, and a private field encapsulation?"

Close. The `public` interface, your method names and what they promise, is the abstraction: the simple surface. The `private` internals, the data and the complicated internal steps, are what encapsulation hides. Same wall, two purposes: one hides data, the other hides complexity.

1.6 Is C++ a pure object-oriented language?

> **Short answer:** No, and that's deliberate. C++ chose to be multi-paradigm so you can use objects where they help and plain functions where they don't.

In a "pure" OO language like Smalltalk, *everything* is an object, even a number. C++ deliberately doesn't force that. You can write a plain function that isn't attached to any class:

LISTING 1.5

```
// C++ is multi-paradigm: a plain function need not
// belong to any class.
#include <iostream>

int add(int a, int b) { return a + b; }   // no object

int main() {
```

```
    std::cout << "add(2, 3) = " << add(2, 3) << "\n";
    return 0;
}
```

```
$ g++ -std=c++23 -Wall -Wextra c1_q6.cpp -o demo && ./demo
```

```
add(2, 3) = 5
```

C++ is **multi-paradigm**. It supports object-oriented code, plain procedural code, generic (template) code, and functional-style code, and lets you mix them in one program. A `Mobile` class sitting next to a free `add()` function next to a `std::sort` template call is normal, idiomatic C++.

> **WHY IT'S A STRENGTH**
>
> Forcing everything into a class leads to pointless wrapper classes, such as a `MathHelper` class whose only job is to hold a static `add()`. C++ lets a function just be a function. Use objects when you're modeling a *thing* with state and behavior. Use a plain function when you just need to compute something.

1.7 When should I *not* use OOP in C++?

> **Short answer:** When there's no state to protect and no "thing" to model. If a task is just "take these inputs, compute this output," a free function beats a class every time.

Reach for a plain function, not an object, when:

- The operation is **stateless**, it computes a result and keeps nothing. `int gcd(int, int)` doesn't need to be a class.
- You'd be writing a class with **no data members** and a pile of static methods. That's a namespace wearing a class costume; use an actual `namespace`.
- You're in a **hot performance path** where a simple array of numbers and a tight loop will beat a graph of small objects chasing pointers across memory. (We'll dig into *why* in the performance chapters, it's about the cache.)
- The problem is genuinely a **sequence of steps**, like a short script, and inventing objects would only add ceremony.

BRAIN POWER

You're asked to write code that resizes 10,000 photos. Do you model each photo as a rich object with behavior, or write a function that takes a buffer of pixels and returns a resized buffer? There's no single right answer, it depends on what else the program does with photos. Keep this in mind; the tension between "model it as an object" and "just process the data" runs through this whole book.

INTERVIEWER'S ANGLE

"When would you not use OOP" tests maturity. Junior candidates defend OOP as always correct; stronger ones name real cases, a tight numeric loop, a small script, a data-oriented hot path. Showing you know the tool's limits signals you choose designs deliberately rather than by habit. The interviewer wants to hear "it depends, and here's when," not a blanket defense of OOP.

Points to Remember

- **OOP** means building programs from **objects** that bundle state and behavior together, instead of free-floating functions.
- A **class** is the blueprint; an **object** is a thing built from it. One class makes many objects, each with its own data.
- **Encapsulation** hides the data and lets the object protect its own **invariants** (like "battery is 0 to 100").
- **Abstraction** hides the complexity, showing a simple surface (`ring()`) over a complex machine.
- **Inheritance** models "is a special kind of" (`iPhone17` is a `Mobile`). **Polymorphism** lets one command (`ring()`) do different things per type.
- C++ is **multi-paradigm**, not purely OO. Use objects to model things with state; use plain functions for stateless computation.

INTERVIEW SPOTLIGHT

"Difference between a class and an object?" is asked in nearly every entry-level C++ interview. Answer it clearly in one line: *a class is the blueprint, an object is an instance built from that blueprint, each object has its own copy of the data.* Then, to give a stronger answer, add: *the class exists at compile time; objects exist at runtime.*

Coming up in Chapter 2

We've been treating C++ as an object language. But it grew out of C, and it still contains most of C inside it. Chapter 2 tackles the questions every C++ engineer gets asked: what can C++ do that C can't, is C++ really a superset of C, and why does `sizeof('a')` give a different answer in each? Bring your `Mobile` class, and we'll see what it compiles down to.

Chapter - 2

C in Its Bones

what C++ kept, what it changed, and the traps in between

C++ grew out of C, and it never fully left home. Most of C still lives inside C++: the same loops, the same pointers, the same way memory works. That inheritance is a gift and a trap. A gift, because everything you know about C still applies. A trap, because the two languages disagree in small, sharp ways that show up in interviews and in real porting work. This chapter walks the border between them.

The programs in this chapter are complete and runnable. Where C and C++ differ, the build commands under each example show exactly what compiles and what doesn't.

SOMEBODY'S GOTTA ASK

"If C++ came from C, can't I just compile my C code with a C++ compiler and be done?"

Mostly, but not quite, and that small gap is exactly what causes problems. A lot of C compiles cleanly as C++. Then one line, a `malloc` without a cast, a variable named `class`, a character constant whose size you assumed, stops the build or quietly changes meaning. Knowing where the two languages differ is what this chapter is for.

2.1 What can C++ do that C cannot?

> **Short answer:** C++ adds whole tools C never had: classes with access control, templates, references, function overloading, namespaces, exceptions, operator overloading, and RAII. C gives you a fast portable assembler with structs. C++ gives you all of that plus the tools to build large systems without losing control.

Take one small example: an object that cleans up after itself. In C you'd allocate memory, use it, and remember to free it, and one forgotten `free()` is a leak. In C++ the object frees itself when its scope ends. Watch the destructor run automatically:

LISTING 2.1

```
// RAII: the object frees its resource automatically
// when its scope ends.
#include <iostream>

class Mobile {
public:
    Mobile()  { std::cout << "Mobile constructed\n"; }
    ~Mobile() {
        std::cout << "Mobile destroyed (cleanup here)\n";
    }
    void charge(int amount) { battery_ += amount; }
    int  getBattery() const { return battery_; }
private:
    int battery_ = 50;
};

int main() {
    std::cout << "entering scope\n";
    {
        Mobile phone;          // constructed here
        phone.charge(30);
        std::cout << "battery = " << phone.getBattery()
                  << "\n";
    }                          // destroyed here
    std::cout << "left scope\n";
    return 0;
}
```

```
$ g++ -std=c++23 -Wall -Wextra c2_q8_raii.cpp -o demo && ./demo
```

```
entering scope
Mobile constructed
battery = 80
Mobile destroyed (cleanup here)
left scope
```

That automatic cleanup is the single most important idea in C++, and it has a name:

NEW TERM

RAII stands for *Resource Acquisition Is Initialization*. It's the C++ idea that a resource, memory, a file, a network socket, a lock, is owned by an object. The object grabs the resource when it's created and releases it automatically when it goes out of scope. You never write the cleanup call, so you can never forget it. RAII comes up constantly in this book; if you remember one idea from C++, make it this one.

THE BIG FIVE C++ ADDS

Classes (data + behavior + access control), **templates** (write once, work for any type), **references** (aliases without pointer syntax), **overloading** (same name, different parameter lists), and **RAII** (resources tied to object lifetime). Each one exists to solve a problem that C makes you solve by hand, every time.

2.2 Is C++ a superset of C?

Short answer: No, not strictly. C++ is *almost* a superset, most C programs compile as C++, but there are real cases where valid C is invalid C++, or means something different. Saying "C++ is a superset of C" will get you corrected in an interview.

Think of it as heavy overlap, not containment:

Valid in C	What happens in C++
`int* p = malloc(n);`	Error. C++ won't implicitly convert `void*` to `int*`. (See question 11.)
`int new = 5;`	Error. `new` is a keyword in C++.
`sizeof('a')` is `sizeof(int)`	It's `1`. Same code, different answer. (See question 12.)
`char* s = "hi";`	Error since C++11. String literals are `const char[]`.

So the honest phrasing is: **C++ was designed to be highly compatible with C, but it is not a strict superset.**

2.3 What C code will fail to compile as C++?

> **Short answer:** The usual suspects are implicit `void*` conversions, C keywords used as identifiers, missing casts, and assigning a string literal to a non-const `char*`.

Here's a small C file that a C compiler accepts and a C++ compiler rejects, line by line:

LISTING 2.2

```
/* This file compiles as C but FAILS as C++.
   Each flagged line is a reason why. */
#include <stdlib.h>

int main(void) {
    /* (1) void* -> int* is implicit in C,
           but an ERROR in C++ */
    int* p = malloc(4);

    /* (2) 'new' is a C++ keyword; fine as a C name */
    int new = 5;

    (void)p;   // silences the unused-variable warning (void)new;
    return 0;
}
```

```
$ gcc -std=c17 c2_q10.c -o demo # compiles as C
$ g++ -std=c++23 c2_q10.c -o demo # FAILS as C++:
```

`error: invalid conversion from 'void*' to 'int*'` · `error: expected unqualified-id before 'new'`

SHARPEN YOUR PENCIL

In **Listing 2.2**, three of the four flagged lines are hard compile errors in C++. One of them is not an error at all; it simply *means something different* in C++ than it does in C. Which flagged line is it, and what changed? (The answer is at the end of this chapter.)

2.4 Why is `void*` implicitly convertible in C but not in C++?

> **Short answer:** Because C++ takes type safety more seriously. C lets a `void*` turn into any object pointer with no cast; C++ makes you ask for that conversion explicitly, so you can't do it by accident.

LISTING 2.3

```cpp
// C++ requires an explicit cast from void*; C does not.
// Standard: concepts (C++20)
#include <cstdlib>
#include <iostream>

int main() {
    // int* a = malloc(sizeof(int));
    //   ^ ERROR in C++: void* -> int* is not implicit

    // explicit cast makes it valid C++:
    int* b = static_cast<int*>(std::malloc(sizeof(int)));
    if (!b) return 1;
    *b = 42;
    std::cout << "value through cast pointer: "
              << *b << "\n";
    std::free(b);
    return 0;
}
```

```
$ g++ -std=c++23 -Wall -Wextra c2_q11.cpp -o demo && ./demo
```

```
value through cast pointer: 42
```

WATCH IT

This is the number-one reason C code fails to build as C++, and the number-one porting headache. The compiler is protecting you: an unchecked `void*` conversion is how you accidentally read an `int` as if it were a `double`. In new C++ code, reach for `new`, containers, or smart pointers and the question disappears.

2.5 Why does `sizeof('a')` differ between C and C++?

> **Short answer:** Because a character constant has a different type in each language. In C, `'a'` is an `int`, so `sizeof('a')` is usually 4. In C++, `'a'` is a `char`, so `sizeof('a')` is exactly 1.

LISTING 2.4

```cpp
// In C++, a character literal has type char, so
// sizeof('a') is 1. (In C, 'a' is int, so it's 4.)
#include <iostream>

int main() {
    std::cout << "sizeof('a')  = " << sizeof('a')
              << "  (char in C++)\n";
    std::cout << "sizeof(char) = " << sizeof(char)
              << "  (1 in both C and C++)\n";
    std::cout << "sizeof(int)  = " << sizeof(int)
              << "\n";
    return 0;
}
```

```
$ g++ -std=c++23 -Wall -Wextra c2_q12.cpp -o demo && ./demo
```

```
sizeof('a')  = 1  (char in C++)
sizeof(char) = 1  (1 in both C and C++)
sizeof(int)  = 4
```

INTERVIEW SPOTLIGHT

"What does `sizeof('a')` return?" is a favorite trap. The winning answer names both languages: *in C it's `sizeof(int)` because character literals are `int`; in C++ it's 1 because they're `char`*. Note that `sizeof(char)` is 1 in *both* languages by definition, it's only the bare literal that differs.

2.6 Can I mix C and C++ in one project? What does `extern "C"` do?

> **Short answer:** Yes, mixing them is routine. `extern "C"` is the bridge. It tells the C++ compiler to give a function C-style linkage (no name mangling), so C and C++ code can call each other and link cleanly.

C++ mangles function names to support overloading, `ring(int)` and `ring()` become different symbols. C doesn't mangle. So a C++ caller looking for a C function must be told "this one uses C rules":

LISTING 2.5

```
/* A C header, made safe to include from BOTH C and C++. */
#ifndef MODEM_H
#define MODEM_H

#ifdef __cplusplus
extern "C" {
#endif

/* plain C linkage, no name mangling */
void reset_modem(int port);

#ifdef __cplusplus
}
#endif

#endif /* MODEM_H */
```

The `__cplusplus` guard means the same header works in a C build (where `extern "C"` would be a syntax error) and a C++ build (where it's needed).

> **INTERVIEWER'S ANGLE**
>
> Questions about mixing C and C++ probe whether you've worked on real systems with legacy or OS-level code, not just greenfield C++. Mentioning the __cplusplus guard and name mangling shows you've actually debugged a link error, not just read about extern "C". If you have a real debugging experience with a missing extern "C", a sentence about it here lands better than more theory.

2.7 Why do I get link errors when calling C functions from C++?

> **Short answer:** Name mangling. If you declare a C function without `extern "C"`, the C++ compiler mangles the name the caller looks for, but the C object file exported the plain, unmangled name. The linker can't match them, and you get "undefined reference."

LISTING 2.6

```
// WRONG: declaring the C function without extern "C".
// C++ mangles the name, so the linker cannot find the
// unmangled C symbol, and linking fails.
void reset_modem(int);   // C++ mangles: _Z11reset_modemi

int main() {
    reset_modem(0);
    return 0;
}
```

```
$ g++ -c c2_q14_bad.cpp -o bad.o && g++ bad.o modem.o -o demo
```

Link error: `undefined reference to `reset_modem(int)'` — the C++ name was mangled, so it never matched the C symbol.

LISTING 2.7

```
/* Compiled as C. Exports the unmangled symbol reset_modem. */
#include "modem.h"
#include <stdio.h>

void reset_modem(int port) {
    printf("modem on port %d reset\n", port);
}
```

LISTING 2.8

```
// C++ program calling a C function. The extern "C"
// in modem.h is what makes this link.
#include "modem.h"

int main() {
```

```
    reset_modem(0);   // links: header used extern "C"
    reset_modem(1);
}
```

```
$ gcc -std=c17 -c modem.c -o modem.o
$ g++ -std=c++23 -c c2_q13_main.cpp -o main.o
$ g++ modem.o main.o -o demo && ./demo
```

```
modem on port 0 reset
modem on port 1 reset
```

BRAIN POWER

The fix is almost always in the *header*, not the call site. If a C library ships a header that already has the `__cplusplus` + `extern "C"` guard from question 13, you get correct linkage for free. When you hit this error, your first move is to check whether that guard is missing.

2.8 Should I still write C-style code in a C++ codebase? When is it justified?

> **Short answer:** Sometimes, yes, but it should be a deliberate choice, not a habit. Default to C++ idioms. Drop to C style only when something concrete demands it.

Reach for C-style code when:

- You're **talking to a C API**, the OS, a driver, or a hardware register that speaks C. The boundary is C, so you meet it in C.
- You need a **stable C ABI** (Application Binary Interface, the low-level contract for how compiled code calls across a boundary), for a plugin, a shared library, or anything loaded across compiler boundaries. C++ has no standard ABI; C does.
- You're on **bare metal or a tiny embedded target** with no C++ runtime, no exceptions, no heap.
- A measured **hot loop** genuinely runs faster on a flat array than on a graph of objects. Measure first; don't assume.

Everywhere else, prefer the C++ tools: **`std::vector` over raw arrays, `std::string` over `char*`, smart pointers over `malloc` / `free`, and RAII over manual cleanup.**

They're safer, and they cost you nothing at runtime when used well.

> **THE HONEST RULE**
>
> "Write C++ by default, write C on purpose." Using `char*` and `malloc` only because that is how you learned them is not a good reason. Using them because you must cross a boundary into C code is a good reason. What separates the two cases is your intent.

Points to Remember

- C++ adds **classes, templates, references, overloading, namespaces, exceptions, and RAII** on top of C.
- C++ is **not a strict superset of C**. It's highly compatible, but valid C can fail or change meaning in C++.
- **`void*` converts implicitly in C** but needs an explicit cast in C++, the most common porting error.
- **`sizeof('a')` is `sizeof(int)` in C, but `1` in C++**, because character literals are `int` in C and `char` in C++.
- **`extern "C"`** disables name mangling so C and C++ can link. Missing it causes "undefined reference" link errors.
- Write **C++ by default, C on purpose**: drop to C style only for C APIs, a stable ABI, bare-metal targets, or a measured hot path.

> **ANSWER TO "SHARPEN YOUR PENCIL"**
>
> The answer is flagged line **(4)** of **Listing 2.2**, the declaration `int f();`. It is not an error in either language, but it *means* different things. In C, `int f();` declares a function that takes an *unspecified* number of arguments. In C++, the empty parentheses mean the function takes *no* arguments, exactly like writing `int f(void);` in C.

Coming up in Chapter 3

You've seen how a C++ object cleans up after itself at the end of a scope. But how does it get *set up* in the first place? Before we can talk about constructors, we need to talk about initialization itself, and C++ has more ways to initialize a variable than any sane language should. Chapter 3 sorts them out: braces versus parentheses, why `{}` blocks narrowing, what `std::initializer_list` does to your constructors, and the traps that make `Mobile m{};` and `Mobile m();` mean completely different things.