Gorgo Continuing C++ – Answer Key

April 11, 2026

Contents

Chapter 1: Object-Oriented Programming

1. C++ does not make all functions virtual by default because virtual dispatch has a cost — each call goes through a vtable pointer, preventing inlining and adding indirection. Most functions do not need polymorphic behavior, so making them virtual would add overhead for no benefit. Requiring `virtual` to be explicit also communicates intent: it tells the reader that this function is meant to be overridden.

2. Output: `B`

`ptr` is a `A*` pointing to a `B` object. Since `who()` is virtual, the call resolves to `B::who()` at run time.

3. The destructor of `Base` is not virtual. When `delete b` is called on a `Base*` pointing to a `Derived` object, only `Base::~Base()` runs. `Derived::~Derived()` never runs, so `data_` is never deleted — a memory leak. Fix: make the destructor virtual: `virtual ~Base() { ... }`.

4. Use an abstract class when there is no sensible default behavior for the base class. `Shape::area()` returning 0.0 is meaningless — making it pure virtual forces every derived class to provide a real implementation. A regular base class with defaults is appropriate when most derived classes share the same behavior and only a few need to override.

5. Output:

```
Meow
...
```

`c.sound()` calls `Cat::sound()` which returns `"Meow"`. `Animal a = c;` slices the `Cat` down to an `Animal`, so `a.sound()` calls `Animal::sound()` which returns `"..."`.

6. Minimum `sizeof(C)` is 12 bytes. `A` has `int x` (4 bytes), `B` adds `int y` (4 bytes), `C` adds `int z` (4 bytes). With no padding: $4 + 4 + 4 = 12$.

7. `Square::area()` is missing the `override` keyword. Without `override`, if the base signature changes (e.g., becomes non-const), this would silently create a new function instead of overriding the virtual one. Adding `override` would cause a compiler error if the signatures don't match.

8. Output: `1 2 3 4`

Constructors run base-first: `Base()` prints `1`, then `Derived()` prints `2`. Destructors run derived-first: `~Derived()` prints `3`, then `~Base()` prints `4`.

9. Inheritance is clearly right for an "is-a" relationship where polymorphism is needed — e.g., `Circle` is a `Shape`, and you want to store different shapes in a container and call `area()` on each. Composition is better for "has-a" relationships — e.g., a `Car` has an `Engine`, but a `Car` is not an `Engine`. Making `Car` inherit from `Engine` would be nonsensical.

10. (Program exercise — no single answer. The program should define `MediaPlayer` with a pure virtual `play()`, at least two derived classes, store them in a `vector<unique_ptr<MediaPlayer>>`, and call `play()` polymorphically.)

Chapter 2: Templates

1. Templates (static polymorphism) resolve at compile time — no runtime overhead, but the types must be known at compile time, and you cannot store different template instantiations in the same container. Virtual functions (dynamic polymorphism) resolve at run time — they have vtable overhead but allow runtime flexibility (storing different types in one container). Templates are faster; virtual functions are more flexible.

2. Output:

```
7
Hola mundo
```

`add(3, 4)` instantiates `add<int>` returning 7. `add(string("Hola"), string(" mundo"))` instantiates `add<std::string>` which concatenates the strings.

3. The call `max_of(3, 4.5)` fails to compile because `T` cannot be deduced — `3` is `int` and `4.5` is `double`. The compiler cannot choose which type `T` should be. Fix: `max_of<double>(3, 4.5)` or use two template parameters.

4. Three instantiations: `identity<int>`, `identity<double>`, and `identity<std::string>`. The calls `identity(1)`, `identity(2)`, and `identity(42)` all use `identity<int>` — the same instantiation.

5. Output: `15`

The fold expression `(args + ...)` expands to `1 + (2 + (3 + (4 + 5)))` = 15.

6. Templates must be in headers because the compiler generates code for each instantiation at the point of use. If the definition is in a `.cpp` file, other translation units cannot see it, and the linker will report "undefined reference" errors for the instantiations they need.

7. CTAD deduces `T` as `const char*`, not `std::string`. `h.get()` returns `const char*`, which works for printing, but is likely not what the programmer intended — the pointer points to a string literal with static storage, so it is safe, but operations like concatenation or comparison would behave differently than with `std::string`. Add a deduction guide or explicitly write `Holder<std::string> h("Lose Yourself")`.

8. Output:

```
int
general
general
```

`describe(42)` matches the `int` specialization. `describe(3.14)` uses the general template (no `double` specialization). `describe("hello")` uses the general template (no `const char*` specialization).

9. Output: `5`

There are 5 arguments: `1` (int), `"two"` (const char*), `3.0` (double), `'4'` (char), `true` (bool).

10. (Program exercise — no single answer. The program should define `Pair<T,U>` with `first`, `second`, a constructor, and `print()`. A deduction guide `Pair(const char*, int) -> Pair<std::string, int>` enables the CTAD usage shown.)

Chapter 3: The Standard Template Library

1. `operator[]` inserts a default value because the alternative (throwing) would make the common pattern `map[key] = value` fail for new keys. You would need to call `insert` or check `find` before every assignment. The current design makes insertion and access share the same syntax, at the cost of accidental insertion on read.

2. Output: `a1b2c3`

`std::map` stores entries sorted by key. The keys are "a", "b", "c" in alphabetical order.

3. `scores["Charlie"]` inserts a default `int` (0) because "Charlie" does not exist in the map. After this line, the map has three entries and `charlie_score` is 0. This is a bug if the intent was to check whether Charlie has a score — use `find()` or `contains()` instead.

4. Output: `4`

The set contains unique values only: {1, 3, 4, 5}. Duplicates (1, 5, 3) are ignored.

5. $\log_2(1{,}000{,}000) \approx 20$. A lookup requires approximately 20 comparisons.

6. Use `std::map` when: (1) you need sorted iteration (e.g., printing entries in alphabetical order), or (2) you need range queries (e.g., "find all keys between A and M"). `std::unordered_map` cannot do either efficiently.

7. Output:

```
20
10
```

After pushing 10, 20, 30: stack is [10, 20, 30] (top = 30). After `pop()`: stack is [10, 20] (top = 20). Print `top()`: 20. After `pop()`: stack is [10] (top = 10). Print `top()`: 10.

8. The programmer expects `smallest` to be 5 (the smallest element), but `priority_queue` is a **max-heap** by default. `top()` returns 15 (the largest), not 5. Fix: use `std::priority_queue<int, std::vector<int>, std::greater<int>>` for a min-heap.

9. Output: `3`

`std::multiset` allows duplicates. The set contains {1, 2, 3, 3, 4, 5, 5, 5, 6, 9}. `count(5)` returns 3.

10. (Program exercise — no single answer. The program should read words in a loop, store counts in a map, and print sorted results.)

Chapter 4: Ranges, Algorithms, and Lambdas

1. The iterator version is kept for backward compatibility and because it supports subranges (sorting only part of a container). `std::ranges::sort` requires a full range, while `std::sort(it1, it2)` can sort any contiguous subrange. Removing the iterator version would break existing code.

2. Output: `8 5`

After sorting: {1, 2, 3, 5, 8, 9}. `find` returns an iterator to 8 (index 4). `*(it - 1)` is the element before 8, which is 5.

3. Output: `3`

`count_if` counts elements where `n % 2 != 0` (odd numbers). The odd numbers in {1, 2, 3, 4, 5} are 1, 3, 5 — count is 3.

4. `doubled` is empty (size 0), so `doubled.begin()` points nowhere. `std::transform` writes past the end of `doubled` — undefined behavior. Fix: create `doubled` with the right size: `std::vector<int> doubled(nums.size());`.

5. `x` = 33. `accumulate` starts with 10 and adds each element: 10 + 4 + 7 + 2 + 9 + 1 = 33.

6. Output: `15 30`

The lambda captures `factor` (3) by value. `multiply(5)` = 5 * 3 = 15. `multiply(10)` = 10 * 3 = 30.

7. The lambda captures `total` **by value** (`[total]`), not by reference. The lambda modifies its own copy of `total`, but the original `total` in `main` stays 0. Fix: capture by reference: `[&total]`.

8. Views are lazy, which means they avoid unnecessary work — if you only need the first few results from a large dataset, lazy evaluation skips processing the rest. Upfront processing would be better when you need all results and want to cache them, or when the transformation is expensive and you will iterate the results multiple times (views recompute each time).

9. Output: `4 5 6`

`filter` keeps elements > 3: {4, 5, 6, 7, 8}. `take(3)` takes the first 3: {4, 5, 6}.

10. (Program exercise — no single answer. The program should sort, filter >70, compute average, and find min/max using algorithms and/or views.)

Chapter 5: Enums, constexpr, and Compile-Time Programming

1. Requiring `static_cast` prevents accidental mixing of enums with integers. Without it, you could write `if (color == 2)` or `int x = color + 1`, which compiles but is often wrong. Scoped enums force you to be explicit about conversions, catching bugs like comparing a `Color` to a `TrafficLight` value.

2. Output: `3`

`Suit::Hearts = 0`, `Diamonds = 1`, `Clubs = 2`, `Spades = 3`. `static_cast<int>(Suit::Spades)` is 3.

3. `p == 2` does not compile because `enum class Priority` does not implicitly convert to `int`. Fix: `if (p == Priority::High)`.

4. `result` = 1024. `power(2, 10)` computes 2^10 = 1024.

5. `constexpr` functions *can* run at compile time but also work at run time. `consteval` functions *must* run at compile time — calling them with run-time values is a compile error. Use `constexpr` for functions that should work in both contexts. Use `consteval` when a function only makes sense at compile time (e.g., computing lookup table entries).

6. `compute(n)` fails to compile because `n` is read from `std::cin` at run time, but `consteval` requires compile-time arguments. Fix: change `consteval` to `constexpr`, or make `n` a `constexpr` value.

7. Output:

```
10
3.14
```

`check(5)`: `std::is_integral_v<int>` is true, so it prints `5 * 2 = 10`. `check(3.14)`: `std::is_integral_v<double>` is false, so it prints `3.14`.

8. `constinit` exists because `constexpr` makes a variable `const` (immutable). `constinit` guarantees compile-time initialization without making the variable immutable. This solves the static initialization order fiasco for globals that need to be modified after initialization.

9. No bug. The code is valid C++17. `get_pair()` returns a `StringPair`, and the structured binding unpacks it. `"Hola"` and `"mundo"` are implicitly converted to `std::string`.

10. (Program exercise — no single answer. The program should define `constexpr` Fahrenheit-to-Celsius, use `static_assert` for 212->100 and 32->0, define `enum class Season`, and print temperatures.)

Chapter 6: Advanced Strings

1. `string_view` is better because it works with both `std::string` and `const char*` without copies. `const std::string&` requires the caller to have a `std::string` — passing a `const char*` would create a temporary. The main risk of `string_view` is that it does not own the data, so if the underlying string is destroyed, the view becomes dangling.

2. Output:

```
aqui
4
```

`remove_prefix(6)` removes the first 6 characters ("Estoy"), leaving "aqui" with size 4.

3. `get_greeting()` returns a `string_view` to a local `std::string`. The string is destroyed when the function returns, and the view becomes a dangling pointer. Accessing it is undefined behavior. Fix: return `std::string` instead.

4. Output: `7`

`regex_search` finds the first match of `\d+` (one or more digits) in the text. The first sequence of digits is "7" (in "Track 7").

5. `std::stoi("0xFF", nullptr, 16)` returns 255. `0xFF` in base 16 is 15*16 + 15 = 255.

6. `from_chars`/`to_chars` use error codes instead of exceptions to avoid the overhead of exception handling. Setting up a try/catch block has a cost (even if no exception is thrown on some implementations), and throwing/catching is very expensive. For code that parses millions of values (like a data file or network protocol), this overhead matters.

7. `std::stoi` throws `std::invalid_argument` because "not a number" cannot be parsed as an integer. The program crashes with an unhandled exception. Fix: wrap in try/catch or validate the input first.

8. Output: `One More Time`

The regex captures three groups: `(.+)` for artist, `(.+)` for title, `(\d+)` for year. `m[2]` is the second capture group: "One More Time".

9. Alternatives include RE2 (Google's fast regex library), PCRE2 (Perl-compatible regular expressions), Hyperscan (Intel's high-performance regex engine), and CTRE (compile-time regular expressions in C++). These libraries are optimized for speed and can be orders of magnitude faster than `std::regex`.

10. (Program exercise — no single answer. The program should parse "Name:Score" entries, convert scores, and compute the average.)

Chapter 7: Utilities

1. A magic value like -1 or "" can be confused with a legitimate value. Is -1 an error or a valid negative score? Is "" an error or an empty name? `std::optional` explicitly distinguishes "no value" from "a value that happens to be zero/empty." It makes the intent clear and prevents bugs from accidentally using sentinel values as real data.

2. Output:

```
42
7
```

`opt` is empty, so `value_or(42)` returns 42. After `opt = 7`, `value_or(42)` returns the actual value: 7.

3. `*name` dereferences an empty optional — undefined behavior. Fix: check `if (name)` first, or use `name.value()` which throws `bad_optional_access`.

4. Output:

```
0
1
```

After `v = "changed"`, the active type is `std::string`. `holds_alternative<int>` is false (0), `holds_alternative<std::string>` is true (1).

5. Use `std::any` when the set of possible types is not known at compile time — for example, a plugin system where plugins can store arbitrary configuration values. The host application does not know what types the plugins will use, so `variant` (which requires listing all types) is not feasible.

6. `a = 10`, `b = 20`, `c = 30`, `d = 40`. `std::make_tuple(10, 20, 30, 40)` creates a tuple, and the structured binding unpacks each element in order.

7. `std::get<int>(v)` throws `std::bad_variant_access` because the active type is `double`, not `int`. Fix: use `std::get<double>(v)` or check with `holds_alternative<int>` first.

8. Output: `2006`

The structured binding `auto [title, year] = p;` creates **copies** of `p.first` and `p.second`. `year = 2007` modifies the copy, not `p.second`. `p.second` is still 2006.

9. Named structs have descriptive field names: `result.artist` is clearer than `std::get<0>(result)`. Tuples are convenient for quick returns of 2-3 values and do not require defining a new type. For larger returns or for types used across multiple functions, a named struct is more maintainable.

10. (Program exercise — no single answer. The program should parse "rgb(r,g,b)" and return `optional<tuple<int,int,int>>`.)

Chapter 8: Namespaces and the Preprocessor

1. `using namespace std;` in a header makes all of `std`'s names visible in every file that includes the header. For example, if a project also has a function called `count` or a class called `array`, they would collide with `std::count` or `std::array`, causing ambiguous name errors — or worse, silently calling the wrong function.

2. Output: `1 2`

`a::x` is 1, `a::b::x` is 2. The inner namespace `b` has its own `x` that shadows `a::x` within `a::b`.

3. `count` in file1.cpp is in an anonymous namespace, giving it internal linkage — it is only visible within file1.cpp. `extern int count` in file2.cpp tries to link to a global `count`, which does not exist. This causes a linker error.

4. Inline namespaces are useful for API versioning. A library can release v2 as an inline namespace so existing users get the new version by default, while users who need the old behavior can explicitly qualify `mylib::v1::function()`. This allows gradual migration without breaking existing code.

5. Output: `11`

`DOUBLE(3 + 4)` expands to `3 + 4 * 2` (text substitution), which is `3 + 8 = 11`, not `14`. Fix: `#define DOUBLE(x) ((x) * 2)` which expands to `((3 + 4) * 2) = 14`.

6. `using namespace std;` in a header file. This pollutes the namespace of every file that includes `utils.h`. Fix: use `std::string` explicitly in the header.

7. Modules compile faster (parsed once, not re-included everywhere), provide better isolation (macros do not leak), and make include order irrelevant. The industry has not fully adopted them because compiler and build system support is still maturing, the ecosystem has decades of header-based code, and migration requires significant effort.

8. Output:

```
42
42
```

Both `outer::value` and `outer::inner::value` refer to the same variable. `inner` is an inline namespace, so its contents are accessible directly from `outer`.

9. `MAX(3+1, 2+3)` expands to `((3+1) > (2+3) ? (3+1) : (2+3))` = `(4 > 5 ? 4 : 5)` = `5`. The parentheses in the macro protect against precedence issues, so this one works correctly.

10. (Program exercise — no single answer. The program should define `music::rock::top_song()` and `music::pop::top_song()`, use `using`, and include an anonymous namespace helper.)

Chapter 9: RAII and Resource Management

1. RAII is important because C++ uses deterministic destruction — destructors run at predictable times (scope exit), even during exception unwinding. This guarantees cleanup without explicit finally blocks. In Java/Python, try/finally requires the programmer to remember cleanup code, and forgetting it causes leaks. RAII makes leak-free code the default rather than something you must actively maintain.

2. If `process(fp)` throws, `fclose(fp)` is never called — the file handle leaks. Fix: wrap the file handle in an RAII class (like the `FileHandle` class in the chapter) or use `std::unique_ptr<FILE, ...>` with a custom deleter.

3. Move constructors should be `noexcept` because the standard library (e.g., `std::vector` during reallocation) falls back to copying if move might throw, to maintain the strong exception guarantee. A non-noexcept move constructor means `vector::push_back` copies instead of moves, which can be dramatically slower.

4. `Connection c2 = c1;` copies `c1`, giving both `c1` and `c2` their own connection. But the class has no user-defined copy constructor, so the compiler generates a memberwise copy. If `Connection` holds a handle or pointer, both objects will try to disconnect the same resource in their destructors — double cleanup. Fix: delete the copy constructor/assignment, or implement them properly.

5. `fclose` is called exactly once. `f1` is moved to `f2`, setting `f1`'s internal pointer to null. When both go out of scope, `f1`'s deleter sees null (no close), and `f2`'s deleter closes the file.

6. The basic guarantee ensures no leaks and valid state but allows partial changes — sufficient for most operations where the caller can handle an inconsistent state. The strong guarantee (rollback on failure) is needed when partial modifications would leave the system in a confusing state — e.g., a bank transfer that debits one account but fails to credit the other.

7. `ptr.release()` releases ownership (returns the raw pointer and sets the unique_ptr to null) but does **not** free the memory. The returned pointer is ignored, causing a memory leak. If the intent was to free the memory, simply let the `unique_ptr` go out of scope — it will call `delete[]` automatically.

8. Output: `C B A`

Scope guards are destroyed in reverse order of construction (LIFO, like all local variables). `g3` runs first ("C"), then `g2` ("B"), then `g1` ("A").

9. `unique_ptr` includes the deleter in its type to avoid overhead — the deleter is stored as part of the object (zero-size for stateless deleters like function pointers). `shared_ptr` stores the deleter in a separate control block (type-erased) so that `shared_ptr<T>` has a uniform type regardless of the deleter. This lets you store `shared_ptr`s with different deleters in the same container, at the cost of a heap allocation for the control block.

10. (Program exercise — no single answer. The program should use `unique_ptr<int, ...>` with a `free`-calling deleter for `malloc`'d memory.)

Chapter 10: Concurrency

1. A joinable thread owns a system thread. If the `std::thread` destructor runs while the thread is still joinable, the program calls `std::terminate` — a deliberate crash. This prevents the accidental situation where a thread continues running with references to destroyed local variables. You must explicitly choose: `join()` to wait, or `detach()` to let it run independently.

2. Data race on `total`. Both threads read and write `total` without synchronization. The result is undefined — it could be 100, 200, 300, or anything else. Fix: use `std::atomic<int>` for `total`, or protect it with a mutex.

3. `lock_guard` is designed for the common case: lock at construction, unlock at destruction, nothing else. Its simplicity prevents misuse. `unique_lock` adds `unlock()` and `lock()` because some patterns require it — condition variables need the lock to be released while waiting, and some algorithms need to unlock temporarily for non-critical work.

4. Output: `6`

Each thread increments `x` three times atomically. Two threads $\times$ 3 increments = 6. The exact interleaving varies, but the final value is always 6 because atomic operations are indivisible.

5. `thread_a` locks m1 then m2. `thread_b` locks m2 then m1. If thread_a locks m1 and thread_b locks m2 simultaneously, each waits for the other's lock — deadlock. Fix: lock in the same order in both threads, or use `std::scoped_lock lock(m1, m2)`.

6. Use `std::async` when you want a result from a background computation. It handles thread creation, exception propagation, and result retrieval automatically. Use `std::thread` when you need control over the thread's lifetime, want to detach it, or need to manage multiple threads in a thread pool.

7. `result` = 42. `std::launch::deferred` means the lambda runs lazily when `fut.get()` is called. 6 * 7 = 42.

8. If `some_condition()` returns true, the function returns with the mutex still locked — deadlock. `mtx.unlock()` is never called on that path. Fix: use `std::lock_guard<std::mutex> lock(mtx)` at the top of the function — it unlocks automatically on any exit path.

9. `std::atomic` is faster for simple operations (counter increment, flag toggle) because it uses hardware-level atomic instructions (like `lock xadd` on x86) with no context switches. A mutex involves kernel calls when contended, which is much more expensive. Prefer a mutex when you need to protect multiple variables or a complex data structure — atomics only protect individual values.

10. (Program exercise — no single answer. The program should split a vector into quarters, use `async` to sum each quarter, collect results with `future::get()`, and print the total.)

Chapter 11: The Filesystem Library

1. `+` on strings just concatenates characters. `"/home/user" + "music"` would produce `"/home/usermusic"` — no separator. The `/` operator is overloaded to insert the correct path separator, giving `"/home/user/music"`. It also handles edge cases like trailing separators.

2. Output:

```
"report"
".pdf"
"docs"
```

`stem()` is the filename without extension: "report". `extension()` includes the dot: ".pdf". `parent_path().filename()` is the last component of the parent: "docs".

3. `file_size("maybe_missing.txt")` throws `filesystem_error` if the file does not exist. Fix: check `fs::exists()` first, or use the `error_code` overload: `auto size = fs::file_size("maybe_missing.txt", ec);`

4. `directory_iterator` lists entries in a single directory (non-recursive). `recursive_directory_iterator` walks the entire directory tree, descending into subdirectories. Use `directory_iterator` when you only care about immediate children. Use `recursive_directory_iterator` when you need to find files anywhere in a tree (e.g., finding all `.cpp` files in a project).

5. Output: `"/home/user/music/track.mp3"`

`lexically_normal()` resolves `.` (current directory) and `..` (parent directory), simplifying the path.

6. Three directories: `b`, `c`, and `d`. `/a` already exists, so `create_directories` creates `b` inside `a`, `c` inside `b`, and `d` inside `c`.

7. `create_directory` only creates one level. If `/new_project` does not exist, creating `/new_project/src/main` fails because the parent `/new_project/src` does not exist. Fix: use `fs::create_directories("/new_project/src/main")`.

8. `remove_all` returns the count so you can verify the operation — e.g., if you expect to remove a directory with 100 files and the count is 3, something is wrong. It is also useful for logging and diagnostics.

9. The second `copy_file` throws `filesystem_error` because `b.txt` already exists and the default option is `copy_options::none` (fail if destination exists). Fix: use `fs::copy_file("a.txt", "b.txt", fs::copy_options::overwrite_existing)`.

10. (Program exercise — no single answer. The program should take a directory argument, use `recursive_directory_iterator`, and count files, directories, and total bytes.)

Chapter 12: Best Practices and Common Idioms

1. Starting with `const` is safer because you can only forget to add mutability, not accidentally add it. If everything is const by default, the compiler catches any attempt to modify something that should not change. Going the other way (adding `const` later), you might miss variables that should have been const, allowing bugs where values are accidentally modified.

2. Follow the Rule of Zero when your class uses only standard library types (string, vector, smart pointers) that manage themselves. Follow the Rule of Five when your class directly manages a resource (raw pointer, file descriptor, etc.). The decision: if you are writing a destructor, you need all five. If you are not, you likely need zero.

3. `Data b = a;` uses the compiler-generated copy constructor, which copies the raw pointer `ptr_`. Now `a.ptr_` and `b.ptr_` point to the same memory. When `a` and `b` are destroyed, both destructors call `delete[] ptr_` on the same address — double free (undefined behavior). Fix: define a copy constructor that allocates new memory and copies the contents, or delete the copy constructor and use smart pointers.

4. CRTP (static): resolved at compile time, no vtable overhead, the compiler can inline calls. But you cannot store different CRTP-derived types in the same container. Virtual functions (dynamic): resolved at run time, vtable overhead, but you can store `Base*` pointers to any derived type in a single container. Choose CRTP for performance-critical code where types are known at compile time. Choose virtual functions when you need run-time flexibility.

5. Output: `45.14`

`auto x = 42` deduces `int`. `auto y = 3.14` deduces `double`. `auto z = x + y`: `int + double` promotes to `double`, so `z` is `45.14`.

6. `unique_ptr` needs the complete type (including the destructor) to destroy the object. In the header, `Impl` is only forward-declared (incomplete type). If the compiler generates the destructor in the header, it cannot call `Impl`'s destructor. You get a compile error about an incomplete type. Defining `~Playlist() = default;` in the `.cpp` file (where `Impl` is complete) solves this.

7. `add_to_vector` takes the vector by **value**, not by reference. It modifies a copy, not the original. `data.size()` is still 3 after the call. Fix: `void add_to_vector(std::vector<int>& v, int x)`.

8. Comments explaining "what" are redundant because the code already says what it does. `// increment counter` above `counter++` adds no information. Comments explaining "why" provide context the code cannot: `// skip the first element because it's a header row` explains intent. Good: `// Use stable_sort to preserve relative order of equal-rated songs`. Bad: `// Sort the songs`.

9. Zero. Both `std::vector<std::string>` and `std::unique_ptr<Widget>` manage their own resources. The compiler-generated destructor, copy (if applicable), and move operations all do the right thing. This is the Rule of Zero. (Note: `unique_ptr` makes the class non-copyable by default, which is probably correct.)

10. (Program exercise — no single answer. The program should define `Printable<Derived>` with CRTP, `Song` and `Album` derived classes with `to_string()`, and demonstrate `print()`.)

Appendix A: Build Systems and Tooling

1. CMake generates platform-specific build files (Makefiles, Visual Studio projects, Ninja files). The actual building is done by those tools (make, msbuild, ninja). CMake is a meta-build system — it describes *what* to build, and the native build tool determines *how*.

2.

```
cmake_minimum_required(VERSION 3.20)
project(AudioApp LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
add_executable(audioapp main.cpp audio.cpp)
target_include_directories(audioapp PRIVATE ${CMAKE_SOURCE_DIR})
target_compile_options(audioapp PRIVATE -Wall -Wextra -pedantic)
```

3. Adding warnings to an existing project often produces hundreds of warnings from code that already works. Developers either disable the warnings (defeating the purpose) or spend days fixing them. Starting with warnings from day one means you fix each warning as it appears — a small incremental cost instead of a large batch.

4. AddressSanitizer catches buffer overflows. Use `-fsanitize=address -g`.

5. ASan and TSan both instrument memory accesses but in incompatible ways — they would interfere with each other's tracking. To test for both, run your test suite twice: once with `-fsanitize=address` and once with `-fsanitize=thread`.

6.

```
break main
run
next
next
next
print count
continue
```

7. `-O0`: no optimization, fastest compile, best for debugging (variables are not optimized away). `-O2`: standard optimization, good balance of speed and compilation time, used for most release builds. `-O3`: aggressive optimization, may increase binary size with auto-vectorization and function inlining; use when maximum performance matters and you have tested thoroughly.

8. Missing `PRIVATE` (or `PUBLIC`/`INTERFACE`) keyword in `target_link_libraries`. Also, `fmt` should be `fmt::fmt` if using FetchContent. Corrected: `target_link_libraries(myapp PRIVATE fmt::fmt)`.

9. Sanitizers detect bugs that tests alone miss because many bugs (buffer overflows, data races, use-after-free) produce correct-looking output most of the time. A test might pass 99% of the time and only fail under specific memory layouts or thread scheduling. Sanitizers instrument every access and catch the bug deterministically, regardless of whether the test output looks correct.

10. (Program exercise — no single answer. The program should have CMakeLists.txt, main.cpp, math_utils.cpp/h with factorial, build with CMake, and run with ASan.)

Appendix B: Testing

1. Shared state means one test can affect another — a test that modifies a global variable can cause the next test to fail (or pass incorrectly). This makes failures order-dependent and non-reproducible, which makes debugging much harder. Independent tests can run in any order and even in parallel.

2.

```
TEST(PalindromeTest, Racecar)
{
    EXPECT_TRUE(is_palindrome("racecar"));
}

TEST(PalindromeTest, Hello)
{
    EXPECT_FALSE(is_palindrome("hello"));
}

TEST(PalindromeTest, EmptyString)
{
    EXPECT_TRUE(is_palindrome(""));
}
```

3. `EXPECT_EQ` reports the failure and continues running the rest of the test — useful when multiple independent checks are in one test. `ASSERT_EQ` stops the test immediately — necessary when the rest of the test would crash or be meaningless if this check fails (e.g., checking that a pointer is non-null before dereferencing it).

4. The test assumes `SetUp()` adds songs in a specific order: "Hey Ya!", "Toxic", "Crazy". If `SetUp()` changes the order (or if the fixture is modified later), `EXPECT_EQ(playlist[0], "Toxic")` breaks. The fix: either sort the expected values, or test more robustly (e.g., check that the erased element is no longer present).

5. TDD is most useful when requirements are clear and testable — e.g., writing a parser, a math library, or fixing a reported bug (write a test that reproduces it first). TDD is less useful for exploratory or prototype code where the API and behavior are still being discovered — you would constantly rewrite tests as the design evolves.

6.

```
TEST_CASE("clamp restricts values to range", "[clamp]")
{
    SECTION("value below range returns lo")
    {
        REQUIRE(clamp(-5, 0, 100) == 0);
    }

    SECTION("value in range returns value")
    {
        REQUIRE(clamp(50, 0, 100) == 50);
    }

    SECTION("value above range returns hi")
    {
        REQUIRE(clamp(200, 0, 100) == 100);
    }
}
```

7. Without virtual functions, the compiler resolves calls at compile time — there is no mechanism to intercept or replace behavior. Google Mock's `MOCK_METHOD` works by overriding virtual functions. If the function is not virtual, the mock class cannot override it, and the test will call the real implementation instead of the mock.

8. The test checks three properties of a default-constructed `std::string`: that `empty()` is true, `size()` is 0, and it compares equal to `""`. These are redundant (if one is true, the others must be), but together they document the expected behavior of an empty string.

9. Writing tests after the code often leads to tests that only cover the "happy path" — the cases the developer already knows work. Writing tests alongside the code forces you to think about edge cases and failure modes

as you write, producing better test coverage. It also means the code is designed to be testable from the start.

10.

```
class MockLogger : public Logger {
public:
    MOCK_METHOD(void, log, (const std::string& message), (override));
    MOCK_METHOD(int, message_count, (), (const, override));
};

class App {
public:
    App(Logger& logger) : logger_(logger) {}
    void run() { logger_.log("Starting"); }
private:
    Logger& logger_;
};

TEST(AppTest, RunLogsStarting)
{
    MockLogger mock;
    EXPECT_CALL(mock, log("Starting")).Times(1);

    App app(mock);
    app.run();
}
```