C++ interview questions and answers

🟢 Junior Level

Q1. What are the key features of object-oriented programming in C++?
A1. C++ supports all four OOP pillars: **Encapsulation** (data and methods wrapped inside a class with access control), **Inheritance** (code reuse and hierarchical relationships), **Polymorphism** (compile-time through function/operator overloading and run-time through virtual functions), and **Abstraction** (exposing only essential interfaces while hiding implementation). Together these make C++ powerful for large, modular systems.

Q2. Explain the differences between public, private, and protected access specifiers.
A2.

- **public**: members are accessible from anywhere the object is visible.
- **protected**: accessible within the class itself and its derived classes, but not from unrelated code.
- **private**: accessible only within the defining class.
  These specifiers enforce encapsulation and control the class's external interface.

Q3. Distinguish between function overloading and overriding.
A3.

- **Overloading**: same function name, different parameter lists, resolved at **compile time**. Example: `void print(int); void print(string);`.
- **Overriding**: a derived class provides its own implementation of a **virtual** base class function, resolved at **run time** through the v-table.

Q4. Compare abstract classes and interfaces in C++.
A4. C++ has no `interface` keyword; an "interface" is simply an **abstract class with only pure virtual functions** and no data members. Abstract classes can contain some implemented methods or data, making them more flexible.

Q5. Can an interface inherit from another interface in C++?
A5. Yes. Because interfaces are just abstract classes, one abstract class can inherit from another, allowing you to build layered or extended interfaces.

Q6. Define the `static` keyword and its significance.
A6.

- **Static data member**: one shared instance across all objects of the class.
- **Static member function**: callable without an object and cannot access non-static members directly.
- **Static local variable**: retains its value across function calls.
  It's crucial for memory management and for representing class-wide state.

Q7. Is it possible to override a static method?
A7. No. Static methods are not tied to a specific object and are resolved at **compile time**, so the concept of run-time overriding doesn't apply.

Q8. Explain polymorphism and inheritance in C++.
A8.

- **Inheritance** allows a derived class to reuse and extend base class functionality.
- **Polymorphism** allows the same interface to represent different underlying forms.
  In C++ it's achieved via **virtual functions** and base class pointers or references.

Q9. Can constructors be inherited in C++?
A9. Prior to C++11, no. Since C++11, you can use the `using` directive (e.g., `using Base::Base;`) to **inherit constructors**, but each is still explicitly invoked by the derived class when creating an object.

Q10. Discuss pass-by-reference and pass-by-value for objects.
A10.

- **By value**: makes a copy; changes in the function don't affect the caller but may incur performance cost.
- **By reference** (`T&` or `const T&`): no copy; allows the function to modify (or read without copying if `const`) the original object.

Q11. Compare == and `.equals` for string comparison.
A11. In C++, `std::string` overloads the == operator to compare **contents**. There is no `.equals` method—this is a Java construct. To compare pointers, use pointer equality, but for actual string text, == is correct.

Q12. Differentiate between local, instance, and class variables.
A12.

- **Local**: declared inside a block or function, lifetime ends when the block ends.
- **Instance (non-static data members)**: each object gets its own copy.
- **Class (static data members)**: single copy shared by all objects of that class.

🟠 Mid Level

Q13. What is reflection in C++?
A13. Standard C++ lacks built-in reflection like Java. Limited runtime type info (RTTI) is available through `typeid` and `dynamic_cast`. For richer reflection, developers use libraries such as Boost.TypeIndex or compile-time techniques with templates and macros.

Q14. Define dependency injection and name a few libraries.
A14. Dependency Injection is a design principle where an object's dependencies are supplied from the outside rather than created internally, improving testability and flexibility. In C++, it's often done manually via constructor or setter injection. Libraries like **Boost.DI** and **Fruit** can automate this.

Q15. Explain strong and weak references in C++.
A15. Managed with smart pointers:

- **Strong**: `std::shared_ptr` maintains ownership and increases the reference count.
- **Weak**: `std::weak_ptr` does not own the object and prevents cyclic references, letting memory be freed when only weak references remain.

Q16. Interpret the meaning of the synchronized keyword.
A16. C++ has no `synchronized` keyword. Thread safety is achieved using `std::mutex`, `std::lock_guard`, or higher-level abstractions like `std::scoped_lock` from `<mutex>`.

Q17. Can memory leaks occur in C++?
A17. Yes. If you allocate with `new`/`malloc` and never free the memory, it leaks. Modern C++ avoids this through RAII (Resource Acquisition Is Initialization) and smart pointers (`unique_ptr`, `shared_ptr`).

Q18. Is it necessary to set references to null in C++?
A18. No. References (`T&`) cannot be reseated or null. For pointers, setting to `nullptr` after deletion is good practice to avoid dangling pointers.

Q19. Discuss object instantiation vs. initialization.
A19. Instantiation is allocating storage for an object (creating it in memory). **Initialization** is giving it a defined starting state via constructors or initializer lists.

Q20. Under what conditions is a static block executed in C++?
A20. C++ doesn't have "static blocks," but it does have **static initialization**: global or namespace-scope objects and static class members are initialized before `main()` begins.

Q21. Why are generics used in C++?
A21. C++ uses **templates** to write type-independent code. Templates enable generic programming, allowing functions and classes to work with any data type while generating type-safe code at compile time.

Q22. Mention some design patterns you are familiar with. Which do you use?
A22. Common C++ patterns: **Singleton** (one instance), **Factory** (object creation encapsulation), **Observer** (event subscription), **Strategy** (algorithm selection at runtime), and **RAII** (automatic resource management). Choice depends on problem domain; RAII is used constantly for memory and resource safety.

Q23. Name some types of testing methodologies in C++.
A23. Unit testing (GoogleTest, Catch2), integration testing, system testing, regression testing, and performance profiling using tools like Valgrind or Google Benchmark.

🔴 Senior Level

Q24. Explain how `std::stoi` works.
A24. `std::stoi` converts a string to an `int`. It skips leading whitespace, handles optional sign characters, converts digits until a non-digit is found, and throws `std::invalid_argument` if no valid conversion exists or `std::out_of_range` if the value exceeds `int` limits.

Q25. What is the double-checked locking problem, and how can it be solved?
A25. In a singleton pattern, checking if an instance exists before and after acquiring a lock can fail because another thread may see a partially constructed object due to instruction reordering.

Solution in modern C++: use `std::call_once` with a `std::once_flag`, or use `std::atomic` with proper memory ordering to ensure full construction visibility.

Q26. How is performance influenced by using the same number in different types: int, double, and float?
A26. `int` operations are typically fastest and smallest in memory. `float` and `double` support fractions; `double` offers more precision but may require more CPU cycles and memory bandwidth, depending on the processor's floating-point unit.

Q27. What is a daemon thread in C++ terms?
A27. There is no formal "daemon thread," but you can create a background thread that runs detached (`std::thread t(func); t.detach();`) so it executes independently of the main thread.

Q28. Can a dead thread be restarted in C++?
A28. No. Once a `std::thread` has finished, it cannot be restarted. You must create a new thread object to run the task again.

✅ Preparation Tips

- Emphasize **modern C++ (C++17/20)** best practices—smart pointers, RAII, `std::thread`, and safe concurrency.
- If an interviewer asks a Java-specific term (e.g., `hashCode`, `Parcelable`), politely clarify that it's not part of standard C++ and offer the closest C++ equivalent