C++ Daily Interview

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

The different interview processes

1.1 The big tech style interview process

One of them, the more usual one nowadays is very competitive and has several rounds. It starts with an initial screening call that is often conducted by a non-technical person, a recruiter. However, I've heard and seen cases where even the first contacts were made by engineers so that they can see earlier whether you'd be a good fit for the team.

This first round might have been preceded by a 0th round with some takeaway exercise. The idea behind is that if you cannot prove a certain level of expertise, they don't want to waste their time on the applicant. Sometimes this exercise is way too long and many of us would outright reject such homework. If it's more reasonable, you can complete it in an hour or two.

After the screening, there is a technical round that is usually quite broad and there is not a lot of time to go deep on the different topics. It's almost sure that you'll get an easy or medium-level Leetcode-style coding exercise. For those, you must be able to reason about algorithmic complexities and it's also very useful if you are more than familiar with the standard library of your language. Apart from a coding exercise, expect more general questions about your chosen language.

By understanding your language deeper - something this book helps with - you'll have a better chance to reach the next and usually final round of interviews, the so-called on-site. Even if it's online, it might still be called on-site and it's a series of interviews you have to complete in one day or sometimes spanned over one day.

It typically has 3 or 4 different types of interviews.

- Behavioural interviews focusing on your soft skills
- A system design interview where you get a quite vague task to design a system. You have to clarify what the requirements are and you have to come up with the high-level architecture and dig deeper into certain parts
- There are different kinds of coding interviews

    1. **Coding exercises**: You won't be able to solve coding exercises with what you learn in this book, but you'll be able to avoid some pitfalls with a deeper understanding of the language. Daily C++ Interview helps you to achieve that understanding. In addition, you must practice on pages like Leetcode, Hackerrank, Codingame, etc
    2.
    3. **Debug interview**: You receive a piece of code and you have to find the bugs. Sometimes this can be called a code review interview. It's still about finding bugs. Personally, I find it a bit deeper than a simple coding exercise. In a coding interview, you are supposed to talk about design flaws, code smells, and testability. If you know C++ well enough, if you try to answer some of the questions of Daily C++ Interview on a day-to-day basis, you'll have a much better chance to recognize bugs, smells, flaws and pass the interview.

1.2 The shorter interview process

Certain companies try to compete for talent by shortening their interview cycles and making a decision as fast as possible. Often they promise a decision in less than 10 days. Usually, they don't offer so competitive packages - but even that is not always true - so they try to compete on something else.

A shorter decision cycle obviously means fewer interviews. Sometimes this approach is combined with the lack of coding interviews - at least for senior engineers. The idea behind is that many engineers despise the idea of implementing those coding exercises. They find it irrelevant and even derogative to implement a linked list. Instead, they will ask questions that help evaluate how deep you understand the given programming language.

Background

2.1 Rvalue references

An rvalue reference is a reference that can only bind to rvalues (temporary values).

```
void func(int&& param) {
    std::cout << "Rvalue reference received\n";
}

int main() {
    int x = 10;
    // func(x);  //  Error! x is an lvalue
    func(20);    //  OK, 20 is an rvalue
}
```

Rvalue references allow efficient resource transfers via std::move. They do not accept lvalues.

2.2 Universal References (T&& in templates)

A universal reference is when T&& appears in a template parameter and depends on type deduction.

```
template <typename T>
void func(T&& param) {
    std::cout << "Universal reference received\n";
}

int main() {
    int x = 10;
    func(x);   //  param is int& (lvalue reference)
    func(20);  //  param is int&& (rvalue reference)
}
```

Universal references can bind to both lvalues and rvalues. Their behavior depends on type deduction. Used primarily for perfect forwarding

2.3 Perfect forwarding

Perfect forwarding is a technique that allows a function to pass its arguments to another function while preserving their original value category (i.e., whether they are lvalues or rvalues).

This is crucial when writing generic functions that should forward arguments efficiently, without unnecessary copies or moves.

2.3.1 The Problem Without Perfect Forwarding

Without perfect forwarding, lvalues and rvalues can be unintentionally converted, leading to performance issues.

```
void func(int& x) { std::cout << "Lvalue reference\n"; }
void func(int&& x) { std::cout << "Rvalue reference\n"; }

template <typename T>
void wrapper(T x) {  // Passes by value (unintended copy)
    func(x); // Always an lvalue inside wrapper
}

int main() {
    int a = 10;
    wrapper(a);  // Calls func(int&) (expected)
    wrapper(20); // Calls func(int&) (unexpected, should be
    ↪ func(int&&))
}
```

The problem is that T x always treats x as an lvalue inside wrapper, even if it was originally an rvalue. This causes an unnecessary copy and prevents func(int&&) from being called.

2.3.2 Perfect Forwarding with std::forward

Perfect forwarding ensures that arguments retain their original value category.

```
template <typename T>
void wrapper(T&& arg) {
    func(std::forward<T>(arg));  // Perfectly forwards argument
}
```

- If arg is an lvalue, T deduces as T&, so std::forward<T>(arg) behaves like an lvalue reference.
- If arg is an rvalue, T deduces as T, so std::forward<T>(arg) behaves like an rvalue reference.

If you do not use std::forward in a universal reference (T&& in a template), the argument loses its rvalue status and is always treated as an lvalue inside the function. This can lead to incorrect function overload resolution and unnecessary copies/moves.

```
template <typename T>
void wrapper(T&& arg) {
    func(arg);  // Problem: 'arg' is always an lvalue inside
    ↪ wrapper
}
```

Even though arg is declared as T&&, it becomes an lvalue when used inside wrapper.

This means that even if you pass an rvalue, it will be treated as an lvalue.

2.4 Exception safety: four types

Exception safety in C++ refers to the guarantees a function or piece of code provides regarding its behavior when exceptions are thrown. Exception safety ensures that objects remain in a valid state and that resources (such as memory, file handles, and locks) are properly managed even in the presence of exceptions.

Exception safety is typically categorized into four levels:

1. **No Guarantee (Unsafe)**: The function provides no exception safety. If an exception is thrown, the program may enter an invalid state, leak resources, or cause undefined behavior.

```
0  void unsafeFunction(std::vector<int>& vec, int value) {
1      vec.push_back(value); // If push_back throws, `vec` may
   ↪  be in an inconsistent state.
2      doSomething(); // May not run if push_back fails.
3  }
```

2. **Basic Guarantee**: If an exception is thrown, no resources leak, and all objects remain in a valid (but possibly modified) state.

```
0  void basicGuarantee(std::vector<int>& vec, int value) {
1      try {
2          vec.push_back(value); // If this throws, `vec` is
   ↪  still valid.
3      } catch (...) {
4          // Exception handling ensures no resource leaks.
5      }
6  }
```

3. **Strong Guarantee**: Either the function succeeds completely or has no effect (strong exception safety). This is often achieved using copy-and-swap techniques or transactions.

```
0  void strongGuarantee(std::vector<int>& vec, int value) {
1      std::vector<int> temp(vec); // Copy to temporary
2      temp.push_back(value); // Modify the copy
3      vec.swap(temp); // Commit the change safely
4  }
```

   If push_back throws, vec remains unchanged.

4. **No-Throw Guarantee (Exception Neutrality)**: The function is guaranteed not to throw exceptions. Achieved by using only non-throwing operations (noexcept)

```cpp
void noThrowFunction(std::vector<int>& vec, int value)
    noexcept {
    vec.push_back(value); // Assumes `push_back` does not
    throw
}
```

2.5 Resource Acquisition Is Initialization (RAII)

Resource Acquisition Is Initialization (RAII) is a programming technique in C++ where resource management (such as memory, file handles, or locks) is tied to the lifetime of objects. This ensures that resources are acquired in a constructor and released in a destructor, providing exception safety and preventing resource leaks.

Smart pointers are an example of this

```cpp
void badFunction() {
    int* ptr = new int(10);
    throw std::runtime_error("Exception!");
    delete ptr;  // This line is never reached, memory leak!
}
```

With smart pointers, we can guarantee RAII

```cpp
void goodFunction() {
    std::unique_ptr<int> ptr = std::make_unique<int>(10);
    throw std::runtime_error("Exception!");
    // `ptr` is destroyed automatically, no memory leak.
}
```

The name Resource Acquisition Is Initialization (RAII) comes from the idea that acquiring a resource (e.g., memory, file, mutex, etc.) should happen at the same time as object initialization—specifically, in the constructor of a class.

- Resource Acquisition → The act of obtaining or allocating a resource (e.g., opening a file, allocating memory).
- Is Initialization → This acquisition happens during the initialization phase of an object, meaning inside its constructor.

C++ only guarantees deterministic destruction for objects with automatic (stack) storage duration. That is, objects created on the stack are automatically destroyed when they go out of scope.

However, RAII extends this guarantee to dynamically allocated resources by using smart pointers and custom RAII classes.

If resource acquisition does not happen during the initialization phase (i.e., inside the constructor in C++), several issues can arise, primarily resource leaks, partially initialized objects, and lack of exception safety.

If a class does not acquire its resource in the constructor, but instead in a separate function, the object may exist in an invalid state.

```
class ResourceManager {
    FILE* file;
public:
    ResourceManager() { file = nullptr; } // No resource acquired here

    void openFile(const char* filename) {
        file = fopen(filename, "r");  // Resource acquired separately
    }

    ~ResourceManager() {
        if (file) fclose(file);
    }
};

int main() {
    ResourceManager rm;
    // Forgot to call `openFile()`, now `rm` is in an invalid state
}
```

The object exists in an invalid state unless openFile() is called manually. A user of the class might forget to call openFile(), leading to runtime errors.

Categories

The questions in this book fall into the following categories

- Auto and type deduction
- Keyword static and its different uses
- Polymorphism, inheritance, and virtual functions
- Lambda functions
- How to use const
- Best practices in modern c++
- Smart pointers
- References, universal references
- C++20
- Special function and the rules of how many
- OOP, inheritance, poylmorphism
- Observable behaviors
- The STL
- Misc

Auto and type deduction

auto type deduction is usually the same as template type deduction, but auto type deduction assumes that a braced initializer represents a std::initializer_list, and template type deduction doesn't hold such premises.

```
auto x = {1, 2, 3}; // auto deduces std::initializer_list<int>
```

```
template <typename T>
void func(T t) {
    std::cout << typeid(T).name() << "\n";
}

int main() {
    func({1, 2, 3}); //  Error: cannot deduce T from {1, 2, 3}
}
```

However, if we explicitly specify std::initializer_list<T>, it works

```
template <typename T>
void func(std::initializer_list<T> t) {
    for (T n : t) {
        std::cout << n << " ";
    }
}

int main() {
    func({1, 2, 3}); //  Works, T = int
}
```

4.1 Auto with const

Consider

```
const int x = 10;
auto y = x;   // y is `int`, not `const int`
```

We must const qualify y if we require it to maintain constness

```
const auto y = x; // y is int and const
```

4.2 Const with auto and references

Using auto& ensures that the deduced type keeps the const qualifier when referencing a const object.

```
const int a = 42;
auto& b = a;  // `b` is `const int&`
```

```
const int x = 5;
auto& y = x;

y = 15; // Error
```

If x were not const, we could also qualify y to be const, therefore not allowing modification of x through y

```
int x = 5;
const auto& y = x;

y = 15; // Error
```

4.3 Const with auto and pointers

First, recall

```
int x = 5;
int y = 10;

// Pointer is mutable, value is not
const int* ptr = &x;
int const* ptr = &x; // Pointer is mutable, value is not
//
// // // Pointer is immutable, value is mutable
int* const ptr = &x;
int *const ptr = &x;

// Pointer and value are immutable
const int* const ptr = &x;
const int *const ptr = &x;
int const *const ptr = &x;
```

First, we discuss mutable pointers to const values

```
// Pointer is mutable, value is not
const int* ptr = &x;

// Pointer to constant int
auto ptr2 = ptr;
*ptr2 = 20; // Error
// Pointer is mutable
ptr2 = &y;

// Both value and pointer are immutable
const auto ptr2 = ptr;
*ptr2 = 20;  // Error
ptr2 = &y // Error
```

Next, constant pointers to non-const data

```
// Pointer is immutable, value is mutable
int* const ptr = &x;

// Disregards const, ptr2 is non-const and value is non-const
auto ptr2 = ptr;
*ptr2 = 20;
ptr2 = &y;

// Pointer is const, value non-const
const auto ptr2 = ptr;
```

Similarly for const pointers to const data, using auto will maintain constness for the data, but not for the pointer. Therefore, in this case, we must also qualify ptr2 with const.

4.4 rvalue references with auto