

C++ Daily Interview

Nathan Warner



Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

1	The different interview processes	2
1.1	The big tech style interview process	2
1.2	The shorter interview process	3
2	Background	4
2.1	Rvalue references	4
2.2	Universal References (T&& in templates)	4
2.3	Perfect forwarding	4
2.3.1	The Problem Without Perfect Forwarding	5
2.3.2	Perfect Forwarding with std::forward	5
3	Categories	6
4	Auto and type deduction	7
4.1	Auto with const	7
4.2	Const with auto and references	8
4.3	Const with auto and pointers	8
4.4	Universal references with auto	9

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, Hacker-rank, 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).

```
0 void func(int&& param) {
1     std::cout << "Rvalue reference received\n";
2 }
3
4 int main() {
5     int x = 10;
6     // func(x); // Error! x is an lvalue
7     func(20);   // OK, 20 is an rvalue
8 }
```

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.

```
0 template <typename T>
1 void func(T&& param) {
2     std::cout << "Universal reference received\n";
3 }
4
5 int main() {
6     int x = 10;
7     func(x); // param is int& (lvalue reference)
8     func(20); // param is int&& (rvalue reference)
9 }
```

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.

```
0 void func(int& x) { std::cout << "Lvalue reference\n"; }
1 void func(int&& x) { std::cout << "Rvalue reference\n"; }
2
3 template <typename T>
4 void wrapper(T x) { // Passes by value (unintended copy)
5     func(x); // Always an lvalue inside wrapper
6 }
7
8 int main() {
9     int a = 10;
10    wrapper(a); // Calls func(int&) (expected)
11    wrapper(20); // Calls func(int&) (unexpected, should be
    ↪ func(int&&))
12 }
```

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.

```
0 template <typename T>
1 void wrapper(T&& arg) {
2     func(std::forward<T>(arg)); // Perfectly forwards argument
3 }
```

- 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.

```
0 template <typename T>
1 void wrapper(T&& arg) {
2     func(arg); // Problem: 'arg' is always an lvalue inside
    ↪ wrapper
3 }
```

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.

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, polymorphism
- 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.

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

```
0 template <typename T>
1 void func(T t) {
2     std::cout << typeid(T).name() << "\n";
3 }
4
5 int main() {
6     func({1, 2, 3}); // Error: cannot deduce T from {1, 2, 3}
7 }
```

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

```
0 template <typename T>
1 void func(std::initializer_list<T> t) {
2     for (T n : t) {
3         std::cout << n << " ";
4     }
5 }
6
7 int main() {
8     func({1, 2, 3}); // Works, T = int
9 }
```

4.1 Auto with const

Consider

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

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

```
0 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.

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

```
0  const int x = 5;
1  auto& y = x;
2
3  y = 15;  // Error
```

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

```
0  int x = 5;
1  const auto& y = x;
2
3  y = 15;  // Error
```

4.3 Const with auto and pointers

First, recall

```
0  int x = 5;
1  int y = 10;
2
3  // Pointer is mutable, value is not
4  const int* ptr = &x;
5  int const* ptr = &x;  // Pointer is mutable, value is not
6  //
7  // // // Pointer is immutable, value is mutable
8  int* const ptr = &x;
9  int *const ptr = &x;
10
11 // Pointer and value are immutable
12 const int* const ptr = &x;
13 const int *const ptr = &x;
14 int const *const ptr = &x;
```

First, we discuss mutable pointers to `const` values

```

0 // Pointer is mutable, value is not
1 const int* ptr = &x;
2
3 // Pointer to constant int
4 auto ptr2 = ptr;
5 *ptr2 = 20; // Error
6 // Pointer is mutable
7 ptr2 = &y;
8
9 // Both value and pointer are immutable
10 const auto ptr2 = ptr;
11 *ptr2 = 20; // Error
12 ptr2 = &y // Error

```

Next, constant pointers to non-const data

```

0 // Pointer is immutable, value is mutable
1 int* const ptr = &x;
2
3 // Disregards const, ptr2 is non-const and value is non-const
4 auto ptr2 = ptr;
5 *ptr2 = 20;
6 ptr2 = &y;
7
8 // Pointer is const, value non-const
9 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