

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, 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  int&& x = 50;
```

Here, `z` is an rvalue reference that binds directly to the temporary value 50. While this might seem trivial for primitive types like `int`, it becomes powerful when working with objects that manage resources (e.g., strings, vectors, or custom classes). By binding to an rvalue, you can extend the lifetime of the temporary object and potentially "move" its resources.

```
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  
9      int&& z = 30;  
10     func(z);  
11 }
```

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.

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`)

```

0 void noThrowFunction(std::vector<int>& vec, int value)
  ↳ noexcept {
1     vec.push_back(value); // Assumes `push_back` does not
  ↳ throw
2 }

```

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

```

0 void badFunction() {
1     int* ptr = new int(10);
2     throw std::runtime_error("Exception!");
3     delete ptr; // This line is never reached, memory leak!
4 }

```

With smart pointers, we can guarantee RAII

```

0 void goodFunction() {
1     std::unique_ptr<int> ptr = std::make_unique<int>(10);
2     throw std::runtime_error("Exception!");
3     // `ptr` is destroyed automatically, no memory leak.
4 }

```

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.

```
0  class ResourceManager {
1      FILE* file;
2  public:
3      ResourceManager() { file = nullptr; } // No resource
   ↪ acquired here
4
5      void openFile(const char* filename) {
6          file = fopen(filename, "r"); // Resource acquired
   ↪ separately
7      }
8
9      ~ResourceManager() {
10         if (file) fclose(file);
11     }
12 };
13
14 int main() {
15     ResourceManager rm;
16     // Forgot to call `openFile()`, now `rm` is in an invalid
   ↪ state
17 }
```

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.

2.6 The partition of memory

In C++, when a program runs, its memory is divided into distinct segments, each serving different purposes. These segments partition the memory block allocated to the program

2.6.1 Text segment (code segment)

This segment contains the executable code of the program. It is usually read-only to prevent accidental modification. The size of this segment is fixed once the program is loaded into memory.

2.6.2 Data Segment

Stores global and static variables.

- **Initialized Data Segment:** Stores global/static variables with explicit initial values.
- **Uninitialized Data Segment (BSS - Block Started by Symbol):** Stores global/static variables initialized to zero (default).

2.6.3 Stack

Used for function calls, local variables, and return addresses. Each function call pushes a new frame onto the stack, which is popped when the function returns. The stack grows downward (toward lower memory addresses).

It has a fixed size, set by the operating system or runtime environment.

Stack Overflow happens if it exceeds its allocated space (e.g., excessive recursion).

On linux the default stack size is typically 8mb, on windows it is typically 1mb.

Note that the *call stack* is essentially the stack in the context of program execution. However, the term "call stack" specifically refers to how the stack is used to manage function calls, while "stack" is a more general term for the data structure or memory region itself. Let me clarify the relationship between the two:

The call stack is a region of memory that operates as a Last-In-First-Out (LIFO) data structure. It is used to keep track of function calls, their parameters, local variables, and control flow. Each time a function is called, a new stack frame (or activation record) is created and pushed onto the call stack. When the function returns, its stack frame is popped off the stack.

The last function called is the first one to return, which aligns with the LIFO principle.

A special register (e.g., esp on x86 or rsp on x86-64) keeps track of the current top of the stack. It is adjusted as data is pushed onto or popped off the stack.

Another register (e.g., ebp on x86 or rbp on x86-64) is often used to reference the base of the current stack frame, making it easier to access local variables and function parameters.

When a function is called, the following happens:

- The return address (where the program should continue after the function returns) is pushed onto the stack.
- The function's parameters are pushed onto the stack.
- The stack pointer is adjusted to allocate space for local variables.

The function uses its stack frame to store local variables and intermediate results.

The frame pointer (if used) helps navigate the stack frame.

When the function finishes, its stack frame is popped off the stack:

- The stack pointer is restored to its previous position.
- The return address is used to jump back to the caller.

The size of a stack frame in C++ (or any other programming language) depends on several factors, including:

- The size of local variables declared in the function.
- The size of function parameters passed to the function.
- The size of the return address (typically the size of a memory pointer, e.g., 4 bytes on 32-bit systems or 8 bytes on 64-bit systems).
- The size of saved registers (if any) that need to be preserved across function calls.
- Alignment requirements (e.g., the stack frame might be padded to align data to specific boundaries for performance reasons).

2.6.4 Heap

Used for dynamic memory allocation (new, malloc). The program can allocate and free memory at runtime.

The heap grows upward (toward higher memory addresses). Memory leaks occur if allocated memory is not freed properly.

Unlike the stack, the heap does not have a fixed size limit, but it is constrained by system memory.

2.6.5 The free list

In C++, particularly in dynamic memory management, the free list is a data structure used to track free (unused) memory blocks in the heap. It is commonly implemented as a linked list of memory blocks that have been allocated and later freed but are available for reuse.

When a program requests memory (e.g., new or malloc), the allocator checks the free list first to find a suitable block.

If a sufficiently large free block is found, it is removed from the free list and returned to the caller.

If no suitable block is found, a new block is allocated from the heap.

When memory is freed (delete or free), the deallocated block is added to the free list.

2.6.6 Memory fragmentation

Memory fragmentation refers to inefficient memory allocation where free memory is divided into small, non-contiguous blocks, making it difficult to allocate large contiguous blocks even if there is enough total free memory. It occurs primarily in dynamic memory management when memory is allocated and freed in an unpredictable manner.

2.6.7 A programs execution

When a program is executed, the operating system (OS) finds a suitable slot in memory to load the program and manages its memory layout.

The OS reserves a contiguous block of memory for the program's execution. This includes space for the text segment (code), data segment (globals/statics), stack, and heap.

The OS determines a base address and loads the program's executable code into that location.

Once the program is loaded the text segment is placed first and has a fixed size.

The data segment follows, holding global/static variables.

The stack starts from a high memory address and grows downward as functions call each other.

The heap starts from a low memory address (above the data segment) and grows upward when dynamic allocations (new, malloc) occur.

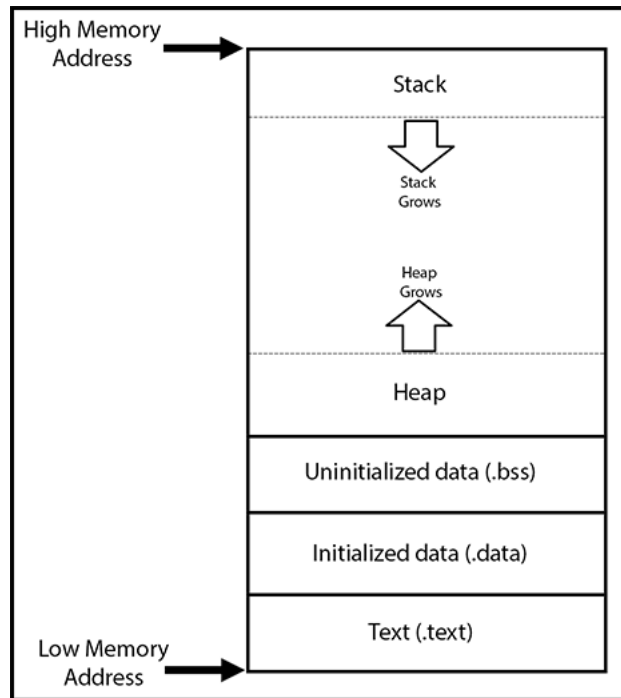
The OS does not preallocate a fixed size for the heap or stack but instead reserves an initial size for the stack and heap and expands or shrinks them dynamically based on need (within system constraints).

If the stack grows too much (e.g., deep recursion) or the heap expands too far, they may collide, causing a crash (segmentation fault).

Stack size is often limited by the OS (default is a few MB), but heap size depends on system RAM

If the stack needs more space, the OS allocates additional pages unless it hits the stack limit (causing a stack overflow).

If the heap needs more memory (malloc or new), the OS extends it using sbrk() or memory mapping (mmap).



The free space (or gap between the stack and heap) in a program's memory layout represents unallocated virtual memory that can be used by either the stack or heap as the program runs.

It acts as a buffer zone between the stack (growing downward) and the heap (growing upward).

2.6.8 What does it mean for memory to be "allocated"

When memory is "allocated," it means that a specific portion of the computer's memory space has been set aside and reserved for a particular program or data to use, essentially giving that program exclusive access to that memory area to store information during its execution; this is done by the operating system based on the program's needs

2.6.9 "Static allocation"

In C++ (and programming in general), statically allocated refers to memory that is allocated at compile time and remains fixed in size and location for the entire duration of the program's execution. This is in contrast to dynamic allocation, where memory is allocated and deallocated at runtime.

Static allocation means that the memory for a variable or object is reserved when the program is compiled, and it persists for the entire lifetime of the program.

The memory is allocated in one of the following regions

1. **Global/Static Memory:** For global variables and static variables.

Characteristics of Static Allocation

- **Fixed Size:** The size of the memory block is known at compile time and cannot change during runtime.
- **Lifetime:** The memory exists for the entire duration of the program (for global/static variables) or for the duration of the function call (for local variables).
- **Efficiency:** Statically allocated memory is fast to allocate and access because it is managed by the compiler and does not require runtime overhead (e.g., no need for new/delete or malloc/free).
- **Scope:** The visibility of the variable depends on where it is declared:
 - Global variables are accessible throughout the program.
 - Static variables inside a function are accessible only within that function but retain their value between calls.

2.6.10 "Automatic allocation"

Automatic allocation refers to memory that is allocated automatically for local variables when a function is called and deallocated automatically when the function exits.

This memory is allocated on the stack, which is a region of memory managed by the compiler and the runtime environment.

The term "automatic" comes from the fact that the allocation and deallocation are handled automatically by the system, without requiring explicit intervention from the programmer (e.g., no need for new/delete or malloc/free).

- **Lifetime:** The memory is allocated when a function is called and deallocated when the function returns.
- **Scope:** The variables are only accessible within the block (e.g., function or block) where they are declared.
- **Speed:** Allocation and deallocation are very fast because the stack is a simple LIFO (Last-In-First-Out) data structure.

Does the Compiler Assign Memory Locations at Compile Time? Yes and no. Here's the distinction:

The size of the stack frame (e.g., how much memory is needed for local variables and parameters) is determined at compile time. The layout of the stack frame (e.g., the order of variables and their offsets within the frame) is also determined at compile time.

The actual memory addresses on the stack are assigned at runtime when the function is called.

The stack pointer (a CPU register, e.g., esp on x86 or rsp on x86-64) keeps track of the current top of the stack and is adjusted dynamically as functions are called and return.

While stack memory is used for local variables, the actual allocation happens at runtime, not compile time.

When a function is called, the necessary stack space is allocated for its local variables.

The values of these variables are stored on the stack as the function executes.

When the function returns, the stack space is deallocated, effectively destroying the local variables.

2.6.11 "Dynamic allocation"

Dynamic memory allocation refers to memory that is explicitly allocated and deallocated at runtime by the programmer using `new`, `new[]`, `malloc()`, or similar functions. This memory is managed on the heap, which has a more flexible lifespan than stack memory.

2.6.12 Stack vs heap allocation

Stack memory is allocated at runtime, but it is not explicitly controlled by the programmer. Instead, the compiler and runtime manage it automatically.

Unlike heap memory, stack allocation is deterministic (it follows a Last In, First Out order) and does not require explicit deallocation.

Heap memory, on the other hand, allows arbitrary allocation and deallocation at any time, making it truly "dynamic."

2.6.13 Why new (dynamic allocation) is slow and stack allocation is fast

The `new` operator in C++ is considered slow compared to stack allocation for several reasons

`new` allocates memory from the heap, which is managed dynamically by the operating system. Heap memory is not contiguous like the stack, so the allocator needs to search for a large enough free block of memory. This requires bookkeeping (maintaining free lists, fragmentation handling, etc.), which adds computational overhead.

If the heap does not have enough free space, `new` might call `malloc()`, which, in turn, may invoke `sbrk()` or `mmap()` at the OS level.

System calls (context switches to the kernel) are expensive compared to simple pointer arithmetic used in stack allocation.

Stack memory is pre-allocated at program startup. Allocation is just moving the stack pointer (simple arithmetic, e.g., `esp -= size`). No system calls or bookkeeping are required.

Deallocation is implicit—when a function returns, all its local variables are freed automatically.

Most systems use a downward-growing stack, meaning the stack grows from higher memory addresses to lower ones.

The Stack Pointer (SP or ESP/RSP in x86/x86-64) keeps track of the top of the stack.

Pushing data decrements the stack pointer (moves it to a lower address).

Popping data increments the stack pointer (moves it to a higher address).

Yes, the stack pointer (SP, ESP in x86, RSP in x86-64) is used to keep track of where to allocate and deallocate memory on the stack.

2.6.14 Does new call malloc?

in most standard C++ implementations, `new` eventually calls `malloc()` (or an equivalent memory allocation function), but with additional functionality.

When you write:

```
0  int* ptr = new int(10);
```

The following steps happen

1. `operator new(size_t)` is called, which internally calls `malloc(size)`.
2. If `malloc()` succeeds, `operator new()` returns the allocated memory.
3. The constructor is called (for non-POD types).
4. A pointer to the object is returned.

For example, the default `operator new()` might look like this:

```
0  void* operator new(std::size_t size) {  
1      void* ptr = std::malloc(size); // Calls malloc internally  
2      if (!ptr) throw std::bad_alloc(); // Handle allocation  
    ↪ failure  
3      return ptr;  
4  }
```

Similarly, `new[]` calls `malloc()` but also stores extra metadata for array size.

2.6.15 Delete

In C++, when you use `delete` on a dynamically allocated object, it performs two main actions:

- Calls the object's destructor (if it has one).
- Frees the allocated memory using `operator delete` (which typically calls `free` under the hood but is not required to).

2.6.16 How does delete know how many bytes to relinquish?

The heap allocator (e.g., `malloc` internally in `operator new`) tracks allocations. This means when `delete` is called, the heap manager looks up the size using metadata stored alongside the allocated block.

2.6.17 Dangling pointer, memory leak, and resource leak

- **Dangling pointer:** A dangling pointer is a pointer that continues to reference a memory location that has been deallocated or freed. Accessing such a pointer can lead to undefined behavior, crashes, or security vulnerabilities.
- **Memory leak:** A memory leak occurs when dynamically allocated memory is not properly deallocated, causing unreachable memory to remain allocated until the program terminates. This can lead to increased memory consumption over time.
- **Resource leak:** A resource leak occurs when a program fails to release system resources such as file handles, network sockets, or database connections, leading to exhaustion of available system resources.

2.7 New in detail

When you use `new` to allocate memory for an object, the following happens:

1. **Memory allocation:** operator `new(size_t)` is called to allocate memory from the heap.
2. **Object construction:** The constructor of the object is called in the allocated memory.
3. **Returns a pointer:** A pointer to the newly created object is returned

If memory allocation fails (e.g., due to insufficient memory), `new` throws a `std::bad_alloc` exception (unless you're using the `nothrow` version of `new`).

2.7.1 Behind the scenes

Suppose we have a struct

```
0 struct point {  
1     int x{},y{};  
2 }
```

The following

```
0 point* p = new point();
```

Is roughly equivalent to

```
0 void* memory = operator new(sizeof(point));  
1 point* ptr = static_cast<point*>(memory);  
2 ptr = new (memory) point(); // or point{}
```

The first line This line allocates raw memory for an object of type `point` using the global operator `new`. This only allocates memory but does not construct the object.

The second line casts the raw memory to a pointer of type `point*`. However, the object is still not constructed at this point.

This line uses placement new to construct an object of type `point` in the already allocated memory. The placement new syntax is `new (memory) point()`, where `memory` is the pointer to the pre-allocated memory.

2.7.2 operator new

`operator new` is a low-level memory allocation function that allocates raw memory but does not invoke the constructor of an object. The syntax

```
0 void* name = operator new(size)
```

performs the following steps

1. It calls the global `operator new` function, which allocates *size* bytes of memory from the free store (heap).
2. The returned pointer (`memory`) is of type `void*` because `operator new` does not know the actual type of the object to be constructed.
3. The allocated memory is not yet initialized (i.e., no constructor is called)

2.7.3 placement new

Placement new is a special form of the `new` operator that constructs an object in a pre-allocated memory block. Instead of allocating memory from the heap, it initializes an object at a specific memory location.

```
0 T* name = new (location) Obj()
```

Does not allocate new memory—it reuses the given memory.

2.7.4 nothrow new

In C++, the `new` operator throws an exception (`std::bad_alloc`) if it fails to allocate memory. However, you can use `std::nothrow` to make `new` return `nullptr` instead of throwing an exception when memory allocation fails.

```
0 int* ptr = new (std::nothrow) int[1000000000]; // Try to
  ↳ allocate a large array
1
2 if (!ptr) {
3     std::cout << "Memory allocation failed!\n";
4 } else {
5     std::cout << "Memory allocation successful!\n";
6     delete[] ptr;
7 }
```

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
```

So we see, `auto` cannot implicitly give us references, we must use `auto&`

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

```

0 int x = 50;
1 auto&& y = x; // y is int&
2 auto&& z = 50; // z is int&&

```

4.5 Auto in function return types or lambda parameters

auto in a function return type or a lambda parameter implies template type deduction, not auto type deduction.

4.6 When can auto deduce undesired types?

Consider the code

```

0  std::vector<bool> foo() {
1      // ...
2  }
3
4  void bar(bool b) {
5      // ...
6  }
7
8  auto someBit = foo()[2]; // this is some pointer to a temp object
9
10 std::vector<bool> bits{ true, false, false, true };
11 bar(bits[2]); // Undefined behaviour

```

`std::vector<bool>` is a specialized version of `std::vector` that is optimized for space efficiency. Instead of storing each `bool` as a full byte, it typically stores each `bool` as a single bit.

This specialization means that `std::vector<bool>` does not behave exactly like other `std::vector` types. Specifically, accessing elements does not return a reference to a `bool` but rather a proxy object (of type `std::vector<bool>::reference`) that simulates a reference to a single bit.

When you access an element of a `std::vector<bool>` using operator[], it returns a temporary proxy object (`std::vector<bool>::reference`) that represents the bit at that position.

This proxy object can be implicitly converted to a `bool`, but it is not a direct reference to a `bool` in the vector.

Temporary objects (like the proxy object returned by `foo()[2]`) are destroyed at the end of the full expression in which they are created. This means that if you try to use the temporary object after the expression, you will be accessing a destroyed object, leading to undefined behavior.

Note: A proxy object is an intermediate object that stands in for another object. In the case of `std::vector<bool>`, accessing an element (e.g., `foo()[2]`) returns a proxy object of type `std::vector<bool>::reference`. This proxy object is not a direct reference to a `bool` but rather a lightweight object that provides access to a single bit in the underlying storage of the `std::vector<bool>`.

So what's happening here?

1. `foo()` returns a temporary `std::vector<bool>`.
2. `foo()[2]` creates a temporary proxy object (`std::vector<bool>::reference`) that refers to the bit at index 2 in the temporary vector.
3. The temporary proxy object is assigned to `x`.

However, the problem arises because The temporary `std::vector<bool>` returned by `foo()` is destroyed at the end of the full expression.

The proxy object (`std::vector<bool>::reference`) depends on the temporary vector for its existence. Once the vector is destroyed, the proxy object becomes invalid because it no longer has a valid vector to refer to.

The language rules specify that temporaries are destroyed at the end of the full expression unless they are bound to a const reference (which extends their lifetime to the lifetime of the reference).

Proxy objects like `std::vector<bool>::reference` are not regular objects but rather lightweight intermediaries. They are not designed to outlive the container they depend on.

To avoid the problem of the proxy object becoming invalid, you need to ensure that the `std::vector<bool>` outlives the proxy object. Here are two ways to do this:

1. Store the vector in a named variable

```
0 auto bits = foo(); // Store the vector in a named variable
1 auto x = bits[2]; // Access the bit from the persistent vector
```

Extract the value immediately, If you only need the value of the bit and not the proxy object, you can extract the value immediately:

```
0 bool x = foo()[2]; // Extract the value of the bit
```

Here, `x` is a `bool` (not a proxy object), so it does not depend on the temporary vector.

In such situations, it's better either not to use `auto` at all, or use the idiom that Meyers (effective modern c++) calls the explicitly typed initializer idiom.

```
0 auto x = static_cast<bool>(bits[2])
```

The idiom can also come in handy when you deal with proxy types.

4.7 What is the type of a lambda?

Consider the code

```
0 auto f = []() { return 12; }; // Type is class(lambda)
```

then the type of `f` is a lambda closure type. The compiler generates a unique unnamed class to represent the lambda expression.

It is a unique unnamed structure that overloads the function call operator. Every instance of a lambda introduces a new type.

When a lambda captures variables, it still generates a unique compiler-defined class, but with data members corresponding to the captured variables.

In the special case of a non-capturing lambda, the structure in addition has an implicit conversion to a function pointer.


```

0   auto f = []() ->int {
1       return 12;
2   };
3   int (*g)();
4   g=f; // OK
5
6   auto f = [&]() ->int {
7       return 12;
8   };
9   int (*g)();
10  g=f; // ERROR

```

4.8 What are the advantages of using auto?

auto variables must be initialized and as such, they are generally immune to type mismatches that can lead to portability or efficiency problems. auto can also make refactoring easier, and it typically requires less typing than explicitly specified types.

auto variables mainly can improve correctness, performance, maintainability, and robustness. It is also more convenient to type, but that's its least important advantage.

Consider declaring local variables `auto x = type{ expr };` when you do want to explicitly commit to a type. It is self-documenting to show that the code is explicitly requesting a conversion, and

it guarantees the variable will be initialized, in addition, it won't allow an accidental implicit narrowing conversion. Only when you do want explicit narrowing, use `()` instead of `{ }`.

4.8.1 Regarding local variables

Regarding local variables, if you use the `auto x = expr;` way of declaration, there are many advantages.

- It's guaranteed that your variable will be initialized. If you forgot, you'll get an error from the compiler.
- There are no temporary objects, implicit conversion, so it is more efficient.
- Using auto guarantees that you will use the correct type.
- In the case of maintenance, refactoring, there is no need to update the type.
- It is the simplest way to portably spell the implementationspecific type of arithmetic operations on built-in types. Those types might vary from platform to platform, and it also ensures that you cannot accidentally get lossy narrowing conversions.
- You can omit difficult to spell types, such as lambdas, iterators, etc.

4.8.2 Correctness

auto ensures that variables have the exact type returned by expressions, preventing unintended type conversions.

```
0 auto x = someFunction(); // Exact type of the return value
```

Complex template or iterator types are assigned correctly

```
0 std::unordered_map<std::string, int> myMap;  
1 auto it = myMap.begin(); // Correct type without manually  
  ↪ specifying
```

4.8.3 Performance

When used with references (auto&), auto can prevent unnecessary copies.

```
0 std::vector<int> vec = {1, 2, 3};  
1 for (auto& v : vec) { // Avoids copying  
2     v *= 2;  
3 }
```

And,

More Efficient Type Deduction: Instead of relying on implicit conversions, auto allows for direct initialization with the optimal type.

4.8.4 Maintainability

If the return type of a function changes, auto ensures code doesn't need manual updates.

```
0 auto result = computeValue(); // No need to change type if  
  ↪ computeValue() changes
```

4.8.5 Robustness

- **Ensures Type Safety in Loops:** Avoids iterator mismatches.

```

0  std::vector<std::string> words = {"hello", "world"};
1  for (auto it = words.begin(); it != words.end(); ++it) {
2      std::cout << *it << "\n";
3  }

```

- **Encourages Strong Typing with auto&&:** Useful in templates for perfect forwarding.

```

0  template <typename T>
1  void wrapper(T&& arg) {
2      process(std::forward<T>(arg)); // Maintains the original
   ↪  type (lvalue/rvalue)
3  }

```

4.9 Initializer_lists

Consider

```

0  auto myCollection = {1,2,3};

```

The type is `std::initializer_list<int>`. The `int` part is probably straightforward, and about `std::initializer_list`, well you just have to know that with `auto` type deduction, if you use braces, you have two options

If you put nothing between the curly braces, you'll get a compilation error as the compiler is unable to deduce '`std::initializer_list<auto>`' from '`<brace-enclosed initializer list>()`'. So you don't get an empty container, but a compilation error instead.

If you have at least one element between the braces, the type will be `std::initializer_list`.

If you wonder what this type is, you should know that it is a lightweight proxy object providing access to an array of objects of type `const T`. It is automatically constructed when:

- a braced-init-list is used to list-initialize an object, where the corresponding constructor accepts a `std::initializer_list` parameter
- a braced-init-list is used on the right side of an assignment or as a function call argument, and the corresponding assignment operator/function accepts a `std::initializer_list` parameter
- a braced-init-list is bound to `auto`, including in a ranged for loop

Initializer lists may be implemented with a pair of pointers or with a pointer and a length. Copying a `std::initializer_list` is considered a shallow copy as it doesn't copy the underlying objects.

4.10 decltype

4.11 Decltype vs auto

decltype deduces the exact type of an expression, including references and const qualifiers. It does not evaluate the expression, only inspects its type. If you pass a variable name without parentheses, it gives the declared type. If you pass an expression, the result depends on how the expression is formed.

```
0  int x = 20;
1  const int& y=x;
2  decltype(y) z = x; // Const int&
```

When decltype is applied to an expression inside parentheses, it deduces the type as a reference if the expression is an lvalue.

```
0  int x = 20;
1  decltype(x) y; // Int
2  decltype((x)) y = x; // int&
3
4  const int x = 10;
5  decltype(x) y = x; // const int
6  decltype((x)) z = x; // const int&
7
8  decltype(10) k; // int
9  decltype((10)) r; // int
```

auto does not preserve references unless explicitly specified. It removes const qualifiers unless the initializer is explicitly const.

Consider

```
0  struct A {
1      double x;
2  }
3
4  // a is a pointer to an A that is const
5  const A* a;
6
7  decltype(a->x) y // y is a double
8  decltype((a->x)) y = z // z is a const double& to y
```

When decltype is used without parentheses, it gives the declared type

```
0  decltype(a->x) y; // y is of type double
```

When `decltype` is applied to a parenthesized expression, it deduces the expression's type with reference qualifiers:

```
0  decltype((a->x)) z = y; // const double&
1  const decltype(a->x)& z = y; // Equivalent
```

Thus, `z` is a `double&`, but since `A` is a pointer to a constant objects, `(a->x)` is a reference to a constant

4.12 `decltype(auto)`

The `decltype(auto)` idiom was introduced in C++14. Like `auto`, it deduces a type from its initializer, but it performs the type deduction using the `decltype` rules.

in generic code, you have to be able to perfectly forward a return type without knowing whether you are dealing with a reference or a value. `decltype(auto)` gives you that ability:

```
0  template<class Fun, class... Args>
1  decltype(auto) foo(Fun fun, Args&&... args) {
2      return fun(std::forward(args)...);
3  }
```

If you wanted to achieve the same thing simply with `auto`, you would have to declare different overloads for the same example function above, one with the return type `auto` always deducing to a pure value, and one with `auto&` always deducing to a reference type.

Keyword static and its different uses

5.1 What does a static member variable in C++ mean?

A member variable that is declared static is allocated storage in the static storage area, only once during the program lifetime. Given that there is only one single copy of the variable for all objects, it's also called a class member.

When we declare a static member in the header file, we're telling the compiler about the existence of a static member variable, but we do not actually define it (it's pretty much like a forward declaration in that sense). Because static member variables are not part of class instances (they are treated similarly to global variables, and get initialized when the program starts), you must explicitly define the static member outside of the class, in the global scope.

```
0  class A {  
1      static MyType s_var;  
2  };  
3  MyType A::s_var = value;
```

Though there are a couple of exceptions. First, when the static member is a const integral type (which includes char and bool) or a const enum, the static member can be initialized inside the class definition:

```
0  class A {  
1      static const int s_var{42};  
2  };
```

If the member is not const, you either have to initialize it out-of-line, or you have to use the inline keyword

static constexpr members can be initialized inside the class definition starting from C++17 (no out-of-line initialization required)

It's been already mentioned but it's worth emphasizing that static member variables are created when the program starts and destroyed when the program ends and as such static members exist even if no objects of the class have been instantiated.

5.2 What does a static member function mean in C++?

static member functions can be used to work with static member variables in the class or to perform operations that do not require an instance of the class. Yet the actions performed in a static member function should conceptually, semantically be strongly related to the class. Some key points about static member functions.

- static member functions don't have the this pointer
- static member functions cannot be virtual
- static member functions cannot access non-static members
- The const and volatile qualifiers aren't available for static member functions

In practice, static member functions mean that you can call such functions without having the class instantiated (you can also call it on an instance).

```
0  struct S {  
1      static void print() {}  
2  };  
3  
4  S::print();  
5  S s;  
6  s.print();
```

As this pointer always holds the memory address of the current object and to call a static member you don't need an object at all, it cannot have a `this` pointer.

A virtual member is something that doesn't relate directly to any class, only to an instance. A "virtual function" is (by definition) a function that is dynamically linked, i.e. the right implementation is chosen at runtime depending on the dynamic type of a given object. Hence, if there is no object, there cannot be a virtual call.

Accessing a non-static member function requires that the object has been constructed but for static calls, we don't pass any instantiation of the class. It's not even guaranteed that any instance has been constructed.

Once again, the `const` and the `const volatile` keywords modify whether and how an object can be modified or not. As there is no object

5.3 What is the static initialization order fiasco?

The static initialization order fiasco is a subtle aspect of C++ that many don't know about, don't consider or misunderstand. It's hard to detect as the error often occurs before `main()` would be invoked.

Static or global variables in one translation unit are always initialized according to their definition order. On the other hand, there is no strict order for which translation unit is initialized first.

Let's suppose that you have a translation unit A with a static variable `sA`, which depends on static variable `sB` from translation unit B in order to get initialized. You have 50% chance to fail. This is the static initialization order fiasco.

```

0  // File1.cpp
1  #include <string>
2  std::string name = "Julia";
3
4  // File2.cpp
5  #include <string>
6  extern std::string message = "Hello, " + name;
7
8  int main() {cout << message;}
9  // g++ -o bin file2.cpp file1.cpp
10 // ./bin "Hello, "
11
12 // g++ -o bin file1.cpp file2.cpp
13 // ./bin "Hello, Julia"

```

5.4 How to solve the static initialization order fiasco?

Probably the simplest solution is to replace the variable in file1.cpp with a function. Observe

```

0  // file1.cpp
1  #include <string>
2  std::string foo() {
3      return std::string("Julia");
4  }
5
6  // file2.cpp
7  #include <string>
8  extern std::string foo();
9  std::string message = "Hello, " + foo();
10
11 int main() {cout << message;}
12 // g++ -o bin file2.cpp file1.cpp
13 // ./bin "Hello, Julia"
14
15 // g++ -o bin file1.cpp file2.cpp
16 // ./bin "Hello, Julia"

```

Starting from C++20, the static initialization order fiasco can be solved with the use of `constinit`. In this case, the static variable will be initialized at compile-time, before any linking.


```

0  // File1.cpp
1  #include <string>
2  constexpr std::string name = "Julia";
3
4  // File2.cpp
5  #include <string>
6  extern std::string message = "Hello, " + name;
7
8  int main() {cout << message;}
9  // g++ -o bin file2.cpp file1.cpp
10 // ./bin "Hello, Julia"
11
12 // g++ -o bin file1.cpp file2.cpp
13 // ./bin "Hello, Julia"

```

Polymorphism, inheritance and virtual functions

6.1 Rules for virtual functions

- They are always member functions
- They cannot be static
- They can be a friend of another class
- C++ does not contain virtual constructors but can have a virtual destructor

In fact, if you want to allow other classes to inherit from a given class, you should always make the destructor virtual, otherwise, you can easily have undefined behaviour

if you are using inheritance and intend to delete derived objects through a base class pointer, you should always make the base class destructor virtual, even if there are no other virtual functions.

If the base class destructor is not virtual and you delete a derived object through a base class pointer, only the base class destructor will be called, leading to undefined behavior (UB) due to a memory leak.

When you declare `print()` as virtual in the base class, it ensures that the function call is dynamically dispatched.

Even though `derived::print()` is private, it is still part of the vtable, and the lookup mechanism allows it to be called when accessed through a base class pointer

Since `b->print();` is being called via a `base*` pointer, the compiler only checks whether `base::print()` is accessible from the calling scope (which it is, because it's public).

The actual function executed (`derived::print()`) is resolved at runtime, at which point access specifiers do not matter.

Thus, the following would not work

```
0  struct base {
1      private:
2      virtual void print() const {cout << "Base" << endl;}
3  };
4
5  struct derived : base{
6      private:
7      void print() const override {cout << "Derived" << endl;}
8  };
9  base* b = new derived{};
10 b->print(); // Print is a private member of base
```

Notice that both `print` methods are now private