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).

```
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.

```
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

    int&& z = 30;
    func(z);
}
```

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.

```
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.

```
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.

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

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

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

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.

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

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.

```
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.

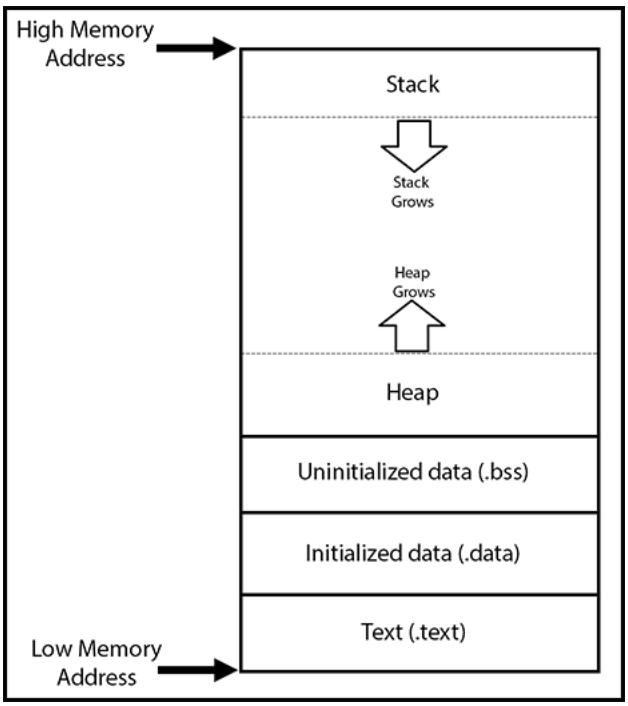
The OS does not preallocate a fixed size for the heap or stack but instead reserves an initial size for the stack and heap and expands or shrinks them dynamically based on need (within system constraints).

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:

```
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:

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

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

```
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.

```
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.

```
int* ptr = new (std::nothrow) int[1000000000]; // Try to
↪  allocate a large array

if (!ptr) {
    std::cout << "Memory allocation failed!\n";
} else {
    std::cout << "Memory allocation successful!\n";
    delete[] ptr;
}
```

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
```

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

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

```
int x = 50;
auto&& y = x; // y is int&
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

```cpp
std::vector<bool> foo() {
    // ...
}

void bar(bool b) {
    // ...
}

auto someBit = foo()[2]; // this is some pointer to a temp object

std::vector<bool> bits{ true, false, false, true };
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

```
auto bits = foo(); // Store the vector in a named variable
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:

```
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 the explicitly typed initializer idiom.

```
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

```
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.

```
 auto f = []() ->int {
    return 12;
};
int (*g)();
g=f; // OK

auto f = [&]() ->int {
    return 12;
};
int (*g)();
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.

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

Complex template or iterator types are assigned correctly

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

4.8.3 Performance

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

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

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.

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

4.8.5 Robustness

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

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

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

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

4.9 Intializer_lists

Consider

```
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.

```
int x = 20;
const int& y=x;
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.

```
int x = 20;
decltype(x) y; // Int
decltype((x)) y = x; // int&

const int x = 10;
decltype(x) y = x; // const int
decltype((x)) z = x; // const int&

decltype(10) k; // int
decltype((10)) r; // int
```

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

Consider

```
struct A {
    double x;
}

// a is a pointer to an A that is const
const A* a;

decltype(a->x) y // y is a double
decltype((a->x)) y = z // z is a const double& to y
```

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

```
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:

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

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

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.

```
class A {
    static MyType s_var;
};
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:

```
class A {
    static const int s_var{42};
};
```

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, a 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 implementaion 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.

```cpp
// File1.cpp
#include <string>
std::string name = "Julia";

// File2.cpp
#include <string>
extern std::string message = "Hello, " + name;

int main() {cout << message;}
// g++ -o bin file2.cpp file1.cpp
// ./bin "Hello, "

// g++ -o bin file1.cpp file2.cpp
// ./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

```cpp
// file1.cpp
#include <string>
std::string foo() {
    return std::string("Julia");
}

// file2.cpp
#include <string>
extern std::string foo();
std::string message = "Hello, " + foo();

int main() {cout << message;}
// g++ -o bin file2.cpp file1.cpp
// ./bin "Hello, Julia"

// g++ -o bin file1.cpp file2.cpp
// ./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.

```cpp
// File1.cpp
#include <string>
constinit std::string name = "Julia";

// File2.cpp
#include <string>
extern std::string message = "Hello, " + name;

int main() {cout << message;}
// g++ -o bin file2.cpp file1.cpp
// ./bin "Hello, Julia"

// g++ -o bin file1.cpp file2.cpp
// ./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

6.2 The diamond problem and its solution

The diamond problem in C++ occurs in multiple inheritance when a class inherits from two base classes that both derive from a common ancestor, leading to duplicate instances of the ancestor and ambiguity in member access. It is resolved using virtual inheritance, ensuring only one shared instance of the common base class exists.

Virtual inheritance is a C++ technique that ensures only one copy of a base class's member variables is inherited by grandchild derived classes. Without virtual inheritance, if two classes B and C inherit from class A, and class D inherits from both B and C, then D will contain two copies of A's member variables: one via B, and one via C. These will be accessible independently, using scope resolution

Instead, if classes B and C inherit virtually from class A, then objects of class D will contain only one set of the member variables from class A.

In practice, virtual base classes are most suitable when the classes that derive from the virtual base, and especially the virtual base itself, are pure abstract classes. This means the classes above the "join class" (the one at the bottom) have very little if any data.

Consider the following class hierarchy to represent the diamond problem, though not with pure abstracts.

```
0  struct Person {
1      virtual ~Person() = default;
2      virtual void speak() {}
3  };
4  struct Student: Person {
5      virtual void learn() {}
6  };
7
8  struct Worker: Person {
9      virtual void work() {}
10 }; // A teaching assistant is both a worker and a student
11 struct TeachingAssistant: Student, Worker {};
12
13 int main() {
14     TeachingAssistant aTeachingAssistant;
15     aTeachingAssistant.Student::speak();
16     // or
17     aTeachingAssistant.Worker::speak();
18
19     // And we can also do
20     Person& p = static_cast<Student&>(aTeachingAssistant);
21     p.speak();
22 }
```

a call to aTeachingAssistant.speak() is ambiguous because there are two Person (indirect) base classes in TeachingAssistant, so any TeachingAssistant object has two different Person base class subobjects. So an attempt to directly bind a reference to the Person subobject of a TeachingAssistant object would fail, since the binding is inherently ambiguous:

To disambiguate, one would have to explicitly convert aTeachingAssistant to either base class subobject:

If we introduce virtual to our inheritance as such, our problems disappear.

```
struct Person {
    virtual ~Person() = default;
    virtual void speak() {}
};
struct Student: virtual Person {
    virtual void learn() {}
};

struct Worker: virtual Person {
    virtual void work() {}
}; // A teaching assistant is both a worker and a student
struct TeachingAssistant: Student, Worker {};

TeachingAssistant aTeachingAssistant;
aTeachingAssistant.speak();

Person* p = new TeachingAssistant{};
p->speak();
```

The Person portion of TeachingAssistant::Worker is now the same Person instance as the one used by TeachingAssistant::Student, which is to say that a TeachingAssistant has only one, shared, Person instance in its representation and so a call to TeachingAssistant::speak is unambiguous. Additionally, a direct cast from TeachingAssistant to Person is also unambiguous, now that there exists only one Person instance which TeachingAssistant could be converted to.

This can be done through vtable pointers. Without going into details, the object size increases by two pointers, but there is only one Person object behind and no ambiguity.

You must use the virtual keyword in the middle level of the diamond. Using it at the bottom doesn't help.

```
cout << sizeof(TeachingAssistant); // 16 (two pointers)
```

6.3 Should we always use virtual inheritance? If yes, why? If not, why not?

The answer is definitely no, we shouldn't use virtual inheritance all the time. According to an idiomatic answer, one of the key C++ characteristics is that you should only pay for what you use. And if you don't need to solve the problems addressed by virtual inheritance, you should rather not pay for it.

Virtual inheritance is almost never needed. It addresses the diamond inheritance problem that we saw just yesterday. It can only happen if you have multiple inheritance, and in case you can avoid it, you don't have the problem to solve. In fact, many languages don't even have this feature

Virtual inheritance causes troubles with object initialization and copying. Since it is the "most derived" class that is responsible for these operations, it has to be familiar with all the intimate details of the structure of base classes. Due to this, a more complex dependency appears between the classes, which complicates the project structure

Troubles with type conversions may also be a source of bugs

In virtual inheritance, the most derived class is responsible for initializing the virtual base class. This creates tight coupling between the most derived class and all the base classes. The most derived class must be aware of every virtual base class and explicitly initialize them. If not handled properly, virtual base classes may be default-initialized, leading to unintended behaviors or even undefined behavior in some cases.

```
class A {
    public:
    int x;
    A(int val) : x(val) {}
};

class B : virtual public A {
    public:
    B() : A(0) {}  // A(0) does nothing because A is virtual
};

class C : virtual public A {
    public:
    C() : A(0) {}  // A(0) does nothing because A is virtual
};

class D : public B, public C {
    public:
    D() : A(42) {}  // Only the most derived class initializes A
};

int main() {
    D d;
    std::cout << d.x << std::endl;  // Correctly prints 42
}
```

The initialization order is as follows

When D d; is executed:

1. Virtual base A is constructed first (via D's constructor).
2. B is constructed (but does not initialize A).
3. C is constructed (but does not initialize A).
4. Finally, D's constructor runs.

Since A is virtually inherited, it is not initialized by B or C at all. The most derived class (D) is responsible for initializing A. If B and C were allowed to initialize A, there would be ambiguity

Note: If you try to instantiate a derived class that doesn't explicitly initialize the virtual base [1], it may lead to default initialization or even compilation errors if the base has no default constructor.

```
class D : public B, public C {
public:
    D() {}  // Error: constructor for D must explicitly
    ↪  initialize the base class A
};
```

- Virtual inheritance breaks the hierarchical structure of the class relationships because it forces the most derived class to handle the construction of the virtual base.
- This means that a base class (e.g., B or C in the example above) cannot assume that the virtual base (A) has been properly initialized.

6.3.1 Issues with Type Conversions

```
class A {
public:
    virtual void foo() {}
};

class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {};

int main() {
    D d;
    A* a = &d;  // Works fine
    B* b = &d;  // Works fine
    C* c = &d;  // Works fine

    B* b2 = dynamic_cast<B*>(a);  // Requires dynamic_cast due
    ↪  to virtual base
    C* c2 = dynamic_cast<C*>(b);  // Also requires dynamic_cast

    return 0;
}
```

Here, since A is virtually inherited, converting from A* to B* or C* requires a runtime check (dynamic$_c ast) instead of a simple implicit cast.$

This increases runtime overhead and makes it harder to predict the behavior of type conversions.

[1] When we talk about a "virtual base class", we are referring to the base class that is inherited using virtual

6.4 Non public inheritance

Consider the following code

```
class A { };
class B : private A { };

A* b = new B(); // Error: Cannot convert B to its private base
↪  class A
```

In C++, when you privately inherit from a base class, it means that the ”is-a” relationship is hidden from the outside. Although class B is implemented in terms of class A, that relationship is not exposed to the outside world. Therefore, a pointer conversion from B* to A* is disallowed outside of B.

With public inheritance, B would be recognized as a subtype of A and the conversion would be allowed:

But with private inheritance, you’re effectively saying that the inheritance is an implementation detail, not a part of B’s public interface. That’s why you get the error ”Cannot cast B to its private base class A”.

Note that the same thing is true for protected inheritance

In C++, public inheritance is used to express an ”is-a” relationship. That is, if class B publicly inherits from class A, then every B object can be used wherever an A is expected—because a B is a kind of A.

On the other hand, when you use non-public (private or protected) inheritance, you’re not promising that B can be used as an A to the outside world. Instead, you’re saying that B is implemented in terms of A, and you’re using A’s implementation internally without exposing its interface as part of B’s public contract. This hides the base class interface from users of B.

Because of this hidden relationship, non-public inheritance can sometimes be seen as analogous to a ”has-a” relationship. In a ”has-a” relationship (or composition), a class contains another class as a member and uses it to implement part of its behavior, but it doesn’t expose that member’s interface as its own. Similarly, with private inheritance, B contains the functionality of A (via inheritance) but does not allow implicit conversion to A* or access to A’s public members from outside of B.

6.5 Can we inherit from a standard container (such as std::vector)? If so what are the implications?

The standard containers declare their constructors as public and non-final, so yes it is possible to inherit from them. In fact, it’s a well-known and used technique to benefit from strongly typed containers

the lack of a virtual destructor might lead to undefined behaviour and a memory leak. Both can be serious issues, but the undefined behaviour is worse because it can not just lead to crashes but even to difficult to detect memory corruption eventually leading to strange application behaviour.

But the lack of virtual destructor doesn't lead to undefined behaviour and memory leak by default, you have to use your derived class in such a way.

If you delete an object through a pointer to a base class that has a non-virtual destructor, you have to face the consequences of undefined behaviour. Plus if the derived object introduces new member variables, you'll also have some nice memory leak. But again, that's the smaller problem.

6.6 What is a destructor and how can we overload it?

A destructor is a special member function of a class. It has the same name as the class and it is also prefixed with a tilde symbol. If available, it is executed automatically whenever an object goes out of scope

A destructor has no parameters, it cannot be const, volatile or static and just like constructors, it has no return type.

By default, it is generated by the compiler, but you have to pay attention as the rule of 5 applies. If any of the other 4 special functions is implemented manually, the destructor will not be generated

As a quick reminder, the special functions besides the destructor are

- copy constructor
- copy assignment operator
- move constructor
- move assignment operator

A destructor is needed if the class acquires resources that have to be released. Remember, you should write RAII class, meaning that resources are acquired on construction and released on destruction. This can be things like releasing connections, closing file handles, saving transactions, etc

As said a destructor has no parameter, it cannot be const, volatile or static, and there can be only one destructor. Hence it cannot be overloaded.

Lambdas

7.1 When could IILF's (Immediately invoked lambda functions) be useful.

Consider the following example

```
const std::string S = [caseA, caseB, caseC]() -> std::string {
    if (caseA) {
        return std::string("A string")
    } else if (caseB) {
        return std::string("B string")
    } else if (caseC) {
        return std::string("C string")
    } else {
        ...
    }
}
```

In this way, we could perform complex initializations beyond the limitations of the conditional operator.

7.2 What kind of captures are available for lambda expressions?

The capture is a comma-separated list of zero or more captures, optionally beginning with the capture-default. The capture list defines the outside variables that are accessible from within the lambda function body.

The only capture defaults are:

- & (implicitly capture the used automatic variables by reference) and
- = (implicitly capture the used automatic variables by copy).

The current object (*this) can be implicitly captured if either of the capture defaults is present. If implicitly captured, it is always captured by reference, even if the capture default is =. However since C++20 the implicit capture of *this with the capture default = is deprecated.

7.2.1 The following types of captures are available:

- **By-copy capture** :

```
int num;
auto l = [num](){};
```

- **By-copy capture that is a pack expansion** :

```
template <typename Args>
void f(Args... args) {
    auto l = [args...] {
        return g(args...);
    };
    l();
}
```

- **by-reference capture**:

```
nt num=42;
auto l = [&num](){};
```

- **by-reference capture that is a pack expansion** :

```
template <typename Args>
void f(Args... args) {
    auto l = [&args...] { return g(args...);};
    l();
}
```

- **by-copy capture of the current object**:

```
auto l = [*this](){};
```

- **by-reference capture of the current object**:

```
auto l = [this](){};
```

- **by-copy capture with an initializer**

```
auto l = [num=5](){};
```

- **by-reference capture with an initializer**:

```
int num=42;
auto l = [&num2=num](){};
```

How to use the const qualifier in C++

8.1 What are the advantages of using const local variables?

By declaring a local variable const you mark it immutable. It should never change its value. If you still try to modify it later on, you'll get a compilation error. For global variables, this is rather useful. Otherwise, you have no idea who might modify their values. Of course, we should avoid using global variables, do you remember the static initialization order fiasco?

Moreover, declaring variables as const also helps the compiler to perform some optimizations. Unless you explicitly mark a variable const, the compiler will not know (at least not for sure) that the given variable should not be changed. Again this is something that we should use whenever it is possible.

8.2 What happens when a class has const member variables?

classes with const members are not assignable. When we have class or struct with const member variables, the copy assignment operator is implicitly deleted

```
0  struct A {
1      const int x = 12;
2      A(int x) : x(x) { cout << "Constructed A object with x = "
   ↪  << x << endl; }
3
4  };
5  A a1(50);
6  A a2(20);
7  a1 = a2; // Error: Copy assignment implicitly deleted
```

Although we can still use the copy constructor just fine.

If you think about it, it makes perfect sense. A variable is something you cannot change after initialization. And when you want to assign a new value to an object, thus to its members, it's not possible anymore.

As such it also makes it impossible to use move semantics, for the same reason.

From the error messages, you can see that the corresponding special functions, such as the assignment operator or the move assignment operator were deleted. Which means we have to implement them by hand. Don't forget about the rule of 5. If we implement one, we have to implement all the 5

Let's take the assignment operator as an example. What should we do with it?

Do we skip assigning to the const members? Not so great, either we depend on that value somewhere, or we should not store the value.

If we really want to implement it, we must use const_cast as a workaround. As you cannot cast the constness away from values, you have to turn the member values into temporary non-const pointers.

```
struct A {
    const int x = 12;
    A(int x) : x(x) { cout << "Constructed A object with x = "
    ↪ << x << endl; }

    A& operator=(const A& other) {
        int* tmp = const_cast<int*>(&x);
        *tmp = other.x;
        return *this;
    }
};
A a1(50);
A a2(20);
a1 = a2; // Error: Copy assignment implicitly deleted
```

Recall that const cast can introduce UB. We've just had a look at the copy assignment and it wouldn't work without risking undefined behaviour.

It's not worth it!

8.3 Should you take plain old data types by const reference as a function parameter?

They should not be passed as const references or pointers. It's inefficient. These data types can be accessed with one memory read if passed by value. On the other hand, if you pass them by reference/pointer, first the address of the variable will be read and then by dereferencing it, the value. That's 2 memory reads instead of one.

We shall not take fundamental data types by const&.

But should we take them simply by const? As always, it depends. If we don't plan to modify their value, yes we should. For better readability, for the compiler and for the future.

```
void f(const int x) { }
// More efficient than
void f(const int& x) {}
```

For a simple type like an int, passing by value (i.e. using "void f(int x)") is generally as efficient—or even slightly more efficient—than passing by reference. This is because:

- **Small Data Type:** An int is typically a small, built-in type that fits into a register, so copying it has negligible cost.
- **Reference Overhead:** Passing by reference usually involves an extra level of indirection (a pointer under the hood), which isn't necessary for such a small type.

- **Compiler Optimizations:** Modern compilers are very good at optimizing simple value copies, making any potential overhead even less noticeable.

That said, the difference in performance for an int is extremely minor. For larger, more complex types, or when you need to modify the original variable, passing by reference may be more appropriate.

Best practices in modern C++

9.1 What are explicit constructors and what are their advantages?

The explicit specifier specifies that a constructor cannot be used for implicit conversions.

If not used, the compiler is allowed to make one implicit conversion to resolve the parameters to a function. The compiler can use constructors callable with a single parameter to convert from one type to another in order to get the right type for a parameter.

```
0  struct A {
1      int x{}, y{};
2
3      A(int x) : x(x) {}
4
5      void f(A a) {}
6  };
7  A a(5);
8  a.f(20); // Ok, compiler will use the constructor to implicitly
   ↪  convert
9  A a2 = 10; // Also ok
```

But if we make the constructor explicit,

```
0  struct A {
1      int x{}, y{};
2
3      explicit A(int x) : x(x) {}
4
5      void f(A a) {}
6  };
7  A a = 5; // No vaiable conversion
8  a.f(20); // No vaiable conversion
```

9.2 What are user-defined literals?

User-defined literals allow integer, floating-point, character, and string literals to produce objects of user-defined type by defining a user-defined suffix.

User-defined literals can be used with integer, floating-point, character and string types

9.3 Why should we use nullptr instead of NULL or 0?

The literal 0 is an int, not a pointer. If the compiler is looking at a 0 where only a pointer can be used, it will interpret 0 as a null pointer, but that's only an implicit conversion, a second option if

you prefer. The same is true for NULL, though implementations are allowed to give NULL an integral type other than int, such as long, which is uncommon

This has the following implication. In case you have a function with three overloads, including integral types and pointers, you might get some surprises:

```
0  void f(int) { cout << "called int fn"; }
1  void f(void*) { cout << "called void* fn"; }
2  f(0); // calls int fn
3  f(nullptr) // calls void* fn
4  f(NULL); // Ambiguous, does not compile
```

On the other hand, nullptr doesn't have an integral type. It's type is std::nullptr_t. It is a distinct type that is not itself a pointer type or a pointer to member type. At the same time, it implicitly converts to all raw pointer types, and that's what makes nullptr act as if it were a pointer of all types.

9.4 What advantages does alias have over typedef?

In case of function pointers, they are more readable:

```
0  typedef void (*MyFunctionPointer)(int, int);
1  using MyFunctionPointerAlias = void(*)(int, int);
```

Another advantage is that typedefs don't support templatization, but alias declarations do

9.5 Should you explicitly delete unused/unsupported special functions or declare them as private?

You might not want a class to be copied or moved, so you want to keep related special functions unreachable for the caller. One option is to declare them as private or protected and the other is that you explicitly delete them.

Before C++11 there was no other option than declaring the unneeded special functions private and not implementing them.

Since C++11 you can simply mark them deleted by declaring them as = delete;

The C++11 way is a better approach because

1. it's more explicit than having the functions in the private section which might only be a mistake
2. in case you try to make a copy, you'll already get an error at compilation time

Deleted functions should be declared as public, not private. It's not a mandate by the compiler, but some compilers might only complain that you call a private function, not that it's deleted.

9.6 What is a trivial class in C++?

When a class or struct in C++ has only compiler-provided or explicitly defaulted special member functions, then it is a trivial type. It occupies a contiguous memory area. It can have members with different access specifiers. In C++, the compiler is free to choose how to order members in this situation. Therefore, you can memcpy such objects but you cannot reliably consume them from a C program. A trivial type T can be copied into an array of char or unsigned char and safely copied back into a T variable. Note that because of alignment requirements, there might be padding bytes between type members.

Trivial types have a trivial a default constructor, trivial copy and move constructors, trivial copy and move assignment operators and a trivial destructor. In each case, trivial means the constructor/operator/destructor is not user-provided and belongs to a class that has

1. no virtual functions or virtual base classes,
2. no base classes with a corresponding non-trivial constructor/- operator/destructor
3. no data members of class type with a corresponding nontrivial constructor/operator/destructor

Whether a class is trivial or not, you can verify with the std::is_trivial trait class. It checks whether the class is trivially copyable (std::is_trivially_copyable) and is trivially default constructible std::is_trivially_default_constructible.

Smart pointers

10.1 RAII

RAII is the bread and butter idiom of C++. It says that anything that exists in a system only in limited supply must be acquired before we start using it.

By such resources, we mean things like

- allocated heap memory,
- execution thread,
- open sockets,
- files,
- locked mutex,
- disk space,
- or database connections.

On the other hand, resources that are not acquired before using them, are not part of RAII, such as

- CPU cores and time,
- cache capacity,
- network bandwidth,
- electric power consumption
- or even stack memory.

In practical terms, an RAII class acquires all the resources upon construction and releases everything on destruction time. You shouldn't have to call methods such as init()/open() or destroy/close.

In the standard library std::string, std::vector or std::thread are such RAII classes.

On the other hand, if you consider raw pointers, they don't share the RAII concept. When a pointer goes out of scope, it doesn't get destroyed automatically, you have to delete it before it's lost and creates a memory leak. On the other hand, the smart pointers of the standard library (std::unique_ptr, std::shared_ptr) provide such a wrapper.

```
0  SomeLimitedResource* resource = new SomeLimitedResource();
1  resouce->doIt(); // oops, it throws an exception...
2
3  delete resource; // this never gets called, so we have a leak
```

Applying RAII by smart pointers, it should be ok:

```
std::unique_ptr<SomeLimitedResource> resource =
std::make_unique<SomeLimitedResource>();

// even if it throws an exception, the resource gets released
resouce->doIt();
```

```
// RAII struct
struct A{
    int* p;
    A(int* p) : p(p) { }

    void ex() {
        throw std::exception();
    }

    ~A() {
        delete p;
    }
};
A a(new int(10));
a.ex(); // Ok! destructor will be called
```

10.2 When should we use unique pointers?

If you need a smart pointer, by default, you should reach for std::unique_ptr. It is a small, fast, move-only smart pointer for managing resources with exclusive-ownership semantics.

They have the same size as a raw pointer, and for most operations, they require the same amount of instructions.

As mentioned, it's for exclusive ownership. Whatever it points to, it also owns it. std::unique_ptr is a move only type, copying is not allowed, there can be only one owner. There is no reference counting, this makes it smaller and faster than the std::shared_- ptr.

By default, resource destruction takes place via delete, but custom deleters can be specified. Stateful deleters and function pointers as deleters increase the size of std::unique_ptr objects. Resource destruction happens as soon as the pointer goes out of scope.

```
std::unique_ptr<T> ptr (new T());
// or;
T* t = new T();
std::unique_ptr<T> ptr2 (t);
```

C++14 introduced std::make_unique to ease the creation:

```
std::unique_ptr<T> ptr = std::make_unique<T>();
```

The new way of pointer creation is safer, because before you could accidentally pass in a raw pointer twice to a new unique pointer like this:

```
T* t = new T();
std::unique_ptr<T> ptr (t);
std::unique_ptr<T> ptr2 (t);
```

A common use for std::unique_ptr is as a factory function return type for objects in a hierarchy. In case it turns out that a shared pointer would be a better fit, the conversion is really easy:

```
std::unique_ptr<T> unique = std::make_unique<T>();
std::shared_ptr<T> shared = std::move(unique);
```

10.3 What are the reasons to use shared pointers?

Shared pointers brought C++ developers the advantages of two worlds. It offers automatic cleanup (a.k.a. garbage collection) that is applicable to all types with a destructor and it is predictable, not like Garbage Collectors in other languages.

Compared to std::unique_ptr or to a raw pointer, std::shared_- ptr objects are typically twice as big because they don't just contain a raw pointer, but they also contain another raw pointer to a dynamically allocated memory area where the reference counting happens.

By default, destruction happens via delete, but just like for std::unique_ptr, custom deleters can be passed. It's worth noting that the type of the deleter has no effect on the type of the std::shared_ptr.

Resource destruction happens as soon as the pointer goes out of scope.

It's available since C++11 and there are two ways to initalize a shared pointer:

```
std::shared_ptr<T> ptr (new T());
std::shared_ptr<T> ptr2 = std::make_shared<T>();
```

The second way of pointer creation - via std::make_shared is safer, because before you could accidentally pass in a raw pointer twice and the cost of the dynamic allocation for the reference count memory is avoided.

Avoid creating std::shared_ptrs from variables of raw pointer type as it's difficult to maintain, difficult to understand when the pointed object would be destroyed.

Use std::shared_ptr for shared-ownership resource management.

10.4 When to use a weak pointer?

A weak pointer - std::weak_ptr is a smart pointer that doesn't affect the object's reference count and as such what it points to might have been already destroyed.

std::weak_ptr is created from a shared_ptr and in case the pointed at object gets destroyed, the weak pointer expires.

```
shared_ptr<int> sp = std::make_shared<int>(10);
weak_ptr<int> wp(sp);

sp = nullptr;
if (wp.expired()) {
    cout << "wp exired" << endl;
}
```

In case you want to use it, you can either call lock() on it that either returns a std::shared_ptr or nullptr in case the pointer is expired, or you can directly pass a weak ptr to shared_ptr constructor.

```
std::shared_ptr<T> sp = wp.lock();
std::shared_ptr<T> sp2(wp);
```

It can be useful for cyclic ownerships, to break the cycle

it can also be useful for caching and for the observer pattern.

10.5 What are the advantages of std::make_shared and std::make_unique compared to the new operator?

Let's start with a reminder. While std::make_shared was added to the STL in C++11, std::make_unique was only added in C++14.

Compared to the direct use of new, make functions eliminate source code duplication, improve exception safety, and std::make_shared generates code that's smaller and faster.

When you use new, if during construction there is an exception thrown, in some circumstances, there might be a resource leak, when the pointer has not yet been "processed" by the make function.

std::make_shared is also faster than simply using new as it allocates memory only once to hold the object and the control block for reference counting. Whereas when you use new it uses two allocations.

Sadly, the mentioned make functions cannot be used if you want to specify custom deleters. At least, they are not often used.

10.6 Should you use smart pointers over raw pointers all the time?

No, raw pointers are, although considered dangerous in many cases, still have their place.

std::unique_ptr transfers and std::shared_ptr shares ownership.

In case a function has nothing to do with ownership, there should be no need for it to take pointer parameters by a smart pointer. Taking smart pointers in such cases will only make its API more restrictive and the run-time cost higher.

If you don't want to share or transfer ownership, just use a raw pointer.

References, universal references, a bit of a mixture

11.1 What does std::move move?

std::move doesn't move anything. At runtime, it does nothing at all. It doesn't even generate a single byte of executable code.

std::move is in fact just a tool to cast whatever its input is to an rvalue reference.

It returns an rvalue reference and that is a candidate for a move operation. Applying std::move to an object tells the compiler that the object is eligible to be moved from. That's why std::move has the name it does have: to make it easy to designate objects that may be moved from.

It's worth noting that moving from a const variable is not possible as the move constructor and the move assignment can change the object from where the move is performed. Yet if you try to move from a const object, the compiler will say nothing. No compiler warning not to say error. Move requests on const objects are silently transformed into copy operations.

11.2 What does std::forward forward?

Just like std::move doesn't move anything, std::forward doesn't forward anything either. Similarly, it does nothing at all at runtime. It doesn't even generate a single byte of executable code.

std::forward is also a cast, just like std::move. But how it is used?

The most common scenario for std::forward is a function template that takes a universal reference parameter that is to be passed to another function:

It is also called perfect forwarding. It has two overloads.

One forwards lvalues as lvalues and rvalues as rvalues, while the other is a conditional cast. It forwards rvalues as rvalues and prohibits forwarding of rvalues as lvalues. Attempting to forward an rvalue as an lvalue, is a compile-time error.

11.3 What is the difference between universal and rvalue references?

If a function template parameter has type T&& for a deduced type T, or if an object is declared using auto&&, the parameter or object is a universal reference.

```
0  template<typename T>
1  void f(T&& param);
2
3  // universal reference
4  auto&& v2 = v; // universal reference
```

But what is a universal reference, you might ask. Universal references correspond to rvalue references if they're initialized with rvalues. They correspond to lvalue references if they're initialized with lvalues. They are either this or that depending on what is passed in.

If the form of the type declaration isn't precisely type&&, or if type deduction does not occur - there is no auto used - we have an rvalue reference.

```
0  void f(MyClass&& param); // rvalue reference
1
2  MyClass&& var1 = MyClass(); // rvalue reference
3
4  template<typename T>
5  void f(std::vector<T>&& param); // rvalue reference
```

The takeaway is that by knowing the differences between rvalue and universal references, you can read source code more accurately. Is this an rvalue type that can be bound only to rvalues or is this a universal reference that can be bound to either rvalue or lvalue references?

11.4 What is reference collapsing?

Reference collapsing can happen in four different scenarios:

- template instantiation
- auto type generation
- creation and use of typedefs and alias declarations
- using decltype

You are not allowed to declare a reference to a reference, but compilers may produce them in the above-listed contexts. When compilers generate references to references, reference collapsing dictates what happens next

There are two kinds of references (lvalue and rvalue), so there are four possible reference-reference combinations:

- lvalue to lvalue
- lvalue to rvalue
- rvalue to lvalue
- rvalue to rvalue

If a reference to a reference arises in one of the four listed contexts, the references collapse to one single reference according to this rule:

> If either reference is an lvalue reference, the result is an lvalue reference. Otherwise (i.e., if both are rvalue references) the result is an rvalue reference.

Universal references are considered as rvalue references in contexts where type deduction distinguishes lvalues from rvalues and where reference collapsing occurs.

11.5 When constexpr functions are evaluated?

constexpr functions might be evaluated at compile-time, but it's not guaranteed. They can be executed both at runtime and at compile time. It often depends on the compiler version and the optimisation level.

If the value of a constexpr function is requested during compile time with constexpr variable, then it will be executed at compile time: constexpr auto foo = bar(42) where bar is a constexpr function.

Also, if a constexpr function is executed in the context of a C-array initialization or static assertion, it will be evaluated at compile time.

In case a constant is needed, but you provide only a runtime function, the compiler will let you know.

It's not a good idea to make all functions constexpr as most computations are best done at run time. At the same time, it's worth noting that constexpr functions will be always threadsafe and inlined.

11.6 When should you declare your functions as noexcept?

You should definitely put noexcept on every function written completely in C or in any other language without exceptions. The C++ Standard Library does that implicitly for all functions in the C Standard Library.

Otherwise, you should use noexcept for functions that don't throw an exception, or if it throws, then you don't mind letting the program crash

Here is a small code sample to show how to use it

```
0  void func1() noexcept; // does not throw
1  void func2() noexcept(true); // does not throw
2  void func3() throw(); // does not throw
3  void func4() noexcept(false); // may throw
```

But what does it mean that a function doesn't throw an exception? It means it cannot use any other function that throws, it is declared as noexcept itself and it doesn't use dynamic_cast to a reference type.

The six generated special functions are implicitly noexcept functions.

If an exception is thrown in spite of noexcept specifier being present, std::terminate is called.

So you can use noexcept when it's better to crash than actually handling an exception, as the Core Guidelines also indicates

Using noexcept can give hints both for the compiler to perform certain optimizations and for the developers as well that they don't have to handle possible exceptions.

C++20