Daily C++ Interview

Prepare yourself for your next interview one question a day

Sandor Dargo

www.sandordago.com

Daily C++ Interview

Prepare yourself for your next interview one question a day

Sandor Dargo

This book is for sale at http://leanpub.com/cppinterview

This version was published on 2022-09-05

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

Table of Contents

Introduction

Daily C++ Interview is an around 150 day program and it offers you questions and coding problems organized around different topics, such as polymorphism, smart pointers, modern C++ best practices, etc. Some questions come back multiple times during the 5 months so that you better remember, yet there are more than 120 unique questions.

I recommend you take a question every day, try to answer it on your own and then read the answer. The depth of the answers is limited due to space and time reasons, but for almost every question you get a list of references, so you're more than welcome to continue the quest.

Before starting to take the questions, let's discuss the different kinds of interviews and see how Daily C++ Interview will help you nail them.

The different typical interview processes

In order to better understand where Daily C++ Interview can help you prepare for your next job interview, let's differentiate between two typical interview processes.

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.

If you're reading this book, most probably that's C++ for you. 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
  - 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
  - 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.

Given the several rounds, scheduling conflicts and sometimes flying in for the last round, such a process can go quite long, it can easily take at least a month if not two.

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.

As this book is concentrating on C++-specific detailed knowledge and not on coding, you'll find it helpful in any type of interview process.

`auto` and type deduction

In this chapter, we are going to learn about C++'s type deduction rules and about how to use the `auto` keyword that was introduced in C++11.

Question 1: Explain auto 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.

Here are a couple of examples:

```
int* ip;
auto aip = ip;          // aip is a pointer to an integer
const int* cip;
auto acip = cip;        // acip is a pointer to a const in\
t (the value cannot be modified, but the memory address i\
t points can)
const int* const cicp = ip;
auto acicp = cicp;      // acicp is still a pointer t a co\
nst int, the constness of the pointer is discarded

auto x = 27;            // (x is neither a pointer nor a r\
eference), x's type is int
const auto cx = x;      // (cx is neither a pointer nor a \
reference), cs's type is const int
const auto& rx = x;     // (rx is a non-universal referenc\
e), rx's type is a reference to a const int

auto&& uref1 = x;       // x is int and lvalue, so uref1's\
 type is int&
auto&& uref2 = cx;      // cx is const int and lvalue, so \
uref2's type is const int &
auto&& uref3 = 27;      // 27 is an int and rvalue, so ure\
f3's type is int&&

auto x3 = { 27 };       // type is std::initializer_list<i\
nt>, value is { 27 }
auto x4{ 27 };          // type is std::initializer_list<i\
nt>, value is { 27 }
                        // in some compilers type may be d\
educed as an int with a
```

```
31                        // value of 27. See remarks for mo\
32 re information.
33 auto x5 = { 1, 2.0 }    // error! can't deduce T for std::\
34 initializer_list<t>
```

As you can see if you use braced initializers, `auto` is forced into creating a variable of type `std::initializer_list`. If it can't deduce the type of T, the code is rejected.

We've also seen that `auto` can give us the correct type for pointers, but in order to get a reference, we must write `auto&`. For consistency, we can also write `auto*` in case we are expecting a pointer.

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

References:

- Effective Modern C++ by Scott Meyers
- Modernes C++

Question 2: When can `auto` deduce undesired types?

"Invisible" proxy types can cause `auto` to deduce the "wrong" type for an initializer expression. One type like that is `std::vector<bool>`. In case you have a function returning such a type, and you are interested in only one bit of information that you want to save into a variable that you declare with auto and later you want to use an item of that vector of `bool`s, it's undefined behaviour.

It sounds a little bit complex. Let's have a look at the code.

```
1 std::vector<bool> foo() {
2   // ...
3 }
4
5 void bar(bool b) {
6   // ...
7 }
8
```

```
 9 auto someBit = foo()[2];
10 bar(bits[2]); // Undefined behaviour
```

The `std::vector`'s `[]` operator returns T&. `std::vector<bool>` is a specialized form of a vector containing only bits, and C++ doesn't allow references to bits. It's a bit complex of what is going on (pun intended), you can read about it more in detail in Effective Modern C++. Long story short, if you ask for a `bool`, an implicit conversion will happen. While the deduced type is something implementation-dependent, it will be a pointer to a temporary object. That's something one really doesn't want.

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

```
1 auto highPriority = static_cast<bool>(features(w)[5]);
```

Similar problems can arise when you'd want a less precise type than returned by a function - to save some memory when you know the range of the possible outputs. Like when a function returns a `double`, but you want a `float`. With `auto`, there would be no implicit conversion, but you can force it with the above idiom.

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

References:

- Effective Modern C++ by Scott Meyers
- Wikipedia

Question 3: 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 `{ }`.

If you are worried about readability the previous technique helps, but otherwise you shouldn't be troubled, modern IDEs tell you the exact types by hovering over the variable.

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 implementation-specific 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.

References:

- Effective Modern C++ by Scott Meyers
- Sutter's Mill

Question 4: What is the type of `myCollection` after the following declaration?

```
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.

References:

- C++ Reference: std::initializer_list
- C++ Reference: List initialization

Question 5: What are trailing return types?

Trailing return type allows the function's return type to be declared following the parameter list (after the ->):

```
auto getElement(const std::vector<int>& container, int in\
dex) const -> int;
```

Instead of starting with `int` as a return type, we put the `auto` keyword at the beginning of the line and we add `int` at the end as a return type after an arrow (->).

With the help of the trailing return type, we can omit the scope of enums for example.

Let's suppose we have this class definition:

```
class Wine {
 public:
  enum WineType { WHITE, RED, ROSE, ORANGE };

  void setWineType(WineType wine_type);

  WineType getWineType() const;

  //...
 private:
  WineType _wine_type;
};
```

Instead of writing:

```
Wine::WineType Wine::getWineType() {
  return _wine_type;
}
```

We can write:

```
auto Wine::getWineType() -> WineType {
  return _wine_type;
}
```

At the beginning of the line, the compiler cannot know the scope, hence we have to write `Wine::WineType`. Whereas when we declare the return type at the end, the compiler already knows what we are in the scope of `Wine`, so we don't have to repeat that information.

Depending on your scope's name, you might spare some characters, but at least you don't have to duplicate the class name.

Probably a more compelling reason to use trailing return types is simplifybg function templates when the return type depends on the argument types. More about that in the next question.

References:

- Effective Modern C++ by Scott Meyers
- Sandor Dargo's Blog

Question 6: Explain `decltype`!

`decltype` is a keyword used to query the type of a variable or an expression. Introduced in C++11, its primary intended use is in generic programming, where it is often difficult, or even impossible, to express types that depend on template parameters.

Given a name or an expression, decltype tells you the name's or the expression's type.

Let's try with some variables and function calls:

```
1 const int&& foo();
2
3 const int bar();
4
5 int i;
6
7 decltype(foo()) x1; // type is const int&&
8 decltype(bar()) x2; // type is int
9 decltype(i) x3; // type is int
```

No surprise. Try with some expressions:

```cpp
struct A {
  double x;
};

const A* a;

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

// type of z is const double& (lvalue expression)
decltype((a->x)) z = y;
```

Still what we wanted.

Perhaps the primary usage of `decltype` is declaring function templates where the function's return type depends on its parameters' types. A usual use-case is a simple addition: adding two values of possibly different types can give a result of many different types, especially when operator overloading is involved.

```cpp
template <class T, class U>
auto add(T const& t, U const& u) -> decltype(t+u) {
  return t+u;
}
```

In C++14, return type deduction for functions was added as a feature, so we could simply use `auto`.

`decltype` is also useful for lambda-related types:

```cpp
auto f = [](int a, int b) -> int {
  return a * b;
};

// the type of a lambda function is unique and unnamed
decltype(f) g = f;
```

References:

- C++ Reference
- Effective Modern C++ by Scott Meyers
- Simplify C++

Question 7: When to use `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.

It allows return type forwarding in generic code, as we referred to it in the previous question. You can use `auto` type deduction even for return types, and you can even return a reference like that or you can declare your return type const:

```
auto const& Example(int const& i) {
  return i;
}
```

On the other hand, 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.

It can also be used to declare local variables when you want to apply the `decltype` type deduction rules to the initializing expression.

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;  // auto type deduction:
                      // myWidget1's type is Widget

decltype(auto) myWidget2 = cw;   // decltype type deducti\
on:
                               // myWidget2's type is con\
st Widget&
```

References:

- Effective Modern C++ by Scott Meyers
- What are some uses of decltype(auto)?

Question 8: Which data type do you get when you add two `bool`s?

To be more practical, what's the output of this code snippet:

```
#include <iostream>

auto foo(bool n, bool m) {
  return n + m;
}

int main() {
  bool a = true;
  bool b = true;
  auto c = a + b;
  std::cout << c << ", " << typeid(c).name() << '\n';
  std::cout << foo(a,b) << '\n';
}
```

The answer is that both `c` and the return value of `foo()` will equal 2. So obviously their type is not `bool` but rather `int`.

In order to understand what happens, the best you can do is to copy-paste the code to CppInsights that does a source-to-source transformation. It helps you to see your source code with the eyes of a compiler.

You'll see this:

```
int c = static_cast<int>(a) + static_cast<int>(b);
```

And something very similar for `foo`. So both of our booleans, `a` and `b` are promoted, they are cast to integers before they are added together.

The keyword `static` and its different usages

In this chapter our schwerpunkt, or main point of focus is on the keyword static.

Question 9: 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.

```
1 class A {
2   static MyType s_var;
3 };
4
5 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:

```
1 class A {
2   static int s_var{42};
3 };
```

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

```
1 class A {
2   // this would work for any class supporting consexpr in\
3 itialization
4   static constexpr std::array<int, 3> s_array{ 1, 2, 3 };
5 };
```

If you are calling a `static` data member within a member function, the member function should be declared as `static`.

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.

Reference:

- C++ Reference: static

Question 10: 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. If you have a `static void bar()` on

`Foo` class, you can call it like this `Foo::bar()` without having `Foo` ever instantiated. (You can also call it on an instance by the way: `aFoo.bar()`).

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…

References:

- LearnC++: Static member functions
- C++Reference: Static members

Question 11: 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**.

Let's see an example for the fiasco.

```
// Logger.cpp

#include <string>

std::string theLogger = "aNiceLogger";

// KeyBoard.cpp

#include <iostream>
#include <string>

extern std::string theLogger;
std::string theKeyboard = "The Keyboard with logger: " + \
theLogger;

int main() {
  std::cout << "theKeyboard: " << theKeyboard << '\n';
}
```

In this example, we have a Keyboard (driver) and a Logger. Let's consider that the keyboard driver needs a logger, so in case of an error, it has somewhere to log them. Due to some reasons, we decided that there can be one keyboard and one logger, so we made them global. Obviously not a great design decision, but it's a quite common scenario and it's also for the sake of the example.

If the keyboard is initialized sooner than the logger, there is an issue, as you can see it in the below example:

```
sdargo@host ~/static_fiasco $ g++ -c Logger.cpp
sdargo@host ~/static_fiasco $ g++ -c Keyboard.cpp
sdargo@host ~/static_fiasco $ g++ Logger.o Keyboard.o -o \
LoggerThenKeyboard
sdargo@host ~/static_fiasco $ g++ Keyboard.o Logger.o -o \
KeyboardThenLogger

sdargo@host ~/static_fiasco $ ./KeyboardThenLogger

theKeyboard: The Keyboard with logger:

```

```
sdargo@host ~/static_fiasco $ ./LoggerThenKeyboard
theKeyboard: The Keyboard with logger: aNiceLogger
```

This is the static initialization order fiasco in action.

Beware that dependencies on `static` variables in different translation units are a code smell and in fact, should be a good reason for refactoring.

References:

- C++ FAQ
- ModernesC++

Question 12: How to solve the static initialization order fiasco?

As a reminder, `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.

In case, 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**.

Dependencies on static variables in different translation units are code smells and in fact, should be a good reason for refactoring. Hence the most straightforward way to solve this problem is to remove such dependencies.

I recite here our example from yesterday.

```
// Logger.cpp

#include <string>

std::string theLogger = "aNiceLogger";

// KeyBoard.cpp

#include <iostream>
#include <string>
```

```
11
12 extern std::string theLogger;
13 std::string theKeyboard = "The Keyboard with logger: " + \
14 theLogger;
15
16 int main() {
17   std::cout << "theKeyboard: " << theKeyboard << '\n';
18 }
```

There is a 50-50% chance of failure. If the compilation unit Logger.cpp gets initialized first, we are good. If the keyboard, not so.

Probably the simplest solution is to replace `theLogger` variable in Logger.cpp with a function like this:

```
1 std::string theLogger() {
2   static std::string aLogger = "aNiceLogger";
3   return aLogger;
4 }
```

Then in the `Keyboard.cpp` we just have to make sure that we use `extern` on the function and we call the function later on instead of referencing the variable. This works because the local `static std::string aLogger` variable will be initialized the first time that `theLogger()` function is called. Hence, it's guaranteed that when `theKeyboard` is constructed, `theLogger` will be initialized.

You might face other issues if `theLogger` would be used during program exit by another static variable after `theLogger` got constructed. Again, dependencies on static variables in different translation units are code smells…

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. You can check that solution here.

References:

- C++ FAQ
- ModernesC++

Polymorphism, inheritance and virtual functions

For the next sixteen questions, we'll focus on polymorphism, inheritance, virtual functions and similar topics.

Question 13: What is the difference between function overloading and function overriding?

Function overriding is a term related to polymorphism. If you declare a virtual function in a base class with a certain implementation, in a derived class you can override its behaviour using the very same signature. If the `virtual` keyword is missing in the base class, it's still possible to create a function with the same signature in the derived class, but it doesn't override it. Since C++11 there is also the `override` specifier which helps you to make sure that you didn't mistakenly fail to override a base class function. You can find more details here.

Function overriding enables you to provide specific implementations in derived classes for functions that are already defined in the base class.

On the other hand, overloading has nothing to do with polymorphism. When you have two functions with the same name, same return type, but different numbers or types of parameters, or qualifiers.

Here is an example for overloading based on parameters

```
1 void myFunction(int a);
2
3 void myFunction(double a);
4
5 void myFunction(int a, int b);
```

And here is another example based on qualifiers:

```
class MyClass {
 public:
  void doSomething() const;
  void doSomething();
};
```

It is even possible to overload a function based on whether the actual instance is an lvalue or an rvalue.

```
class MyClass {
 public:
  // ...
  void doSomething() &; // used when *this is a lvalue
  void doSomething() &&; // used when *this is a rvalue
};
```

You can learn more about this here.

References

- Why to use the override specifier in C++ 11?
- How to use ampersands in C++

Question 14: What is a `virtual` function?

A `virtual` function is used to replace, in other words, to override the implementation provided by the base class.

The code in the derived class is always called whenever the object is actually of the derived class, even if the object is accessed by a base class pointer rather than a derived one.

```
#include <iostream>

class Car {
 public:
  virtual void printName() {
   std::cout << "Car\n";
  }
};

class SUV: public Car {public:
```

```
11   virtual void printName() override {
12    std::cout << "SUV\n";
13   }
14 };
15
16 int main() {
17   Car* car = new SUV();
18   car->printName();
19 }
20 /*
21 SUV
22 */
```

A `virtual` function is a member function that is present in the base class and might be redefined by the derived class(es). When we use the same function name both in the base and derived classes, the function in the base class must be declared with the keyword `virtual` - otherwise it's not overriden just shadowed.

When the function is made `virtual`, then C++ determines at run-time which function is to be called based on the type of the object pointed by the base class pointer. Thus, by making the base class pointer point to different objects, we can execute different versions of the `virtual` functions.

Some rules for `virtual` function:

- 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.

Reference:

- C++ Reference: virtual function specifier

Question 15: What is the `override` keyword and what are its advantages?

The `override` specifier will tell both the compiler and the reader that the function where it is used is actually overriding a method from its base class.

It tells the reader that "this is a `virtual` method, that is overriding a `virtual` method of the base class."

Use it correctly and you see no effect:

```
class Base {
 virtual void foo();
};

class Derived : Base {
  void foo() override; // OK: Derived::foo overrides Base\
::foo
};
```

But it will help you revealing problems with constness:

```
class Base {
  virtual void foo();
  void bar();
};

class Derived : Base {
  void foo() const override; // Error: Derived::foo does \
not override Base::foo\n
                             // It tries to override Base\
::foo const that doesn't exist
};
```

Let's not forget that in C++, methods are non-`virtual` by default. If we use `override`, we might find that there is nothing to override. Without the `override` specifier we would just simply create a brand new method. No more base methods are forgotten to be declared as `virtual` if you use consistently the `override` specifier.

\n

```
class Base {
  void foo();
```

```
};

class Derived : Base {
  void foo() override; // Error: Base::foo is not virtual
};
```

We should also keep in mind that when we override a method - with or without the `override` specifier - no conversions are possible:

```
class Base {
 public:
  virtual long foo(long x) = 0;
};

class Derived: public Base {
 public:
  // error: 'long int Derived::foo(int)' marked override,\
 but does not override\n
  long foo(int x) override {
    // ...
  }
};
```

In my opinion, using the override specifier from C++11 is part of clean coding principles. It reveals the author's intentions, it makes the code more readable and helps to identify bugs at build time. Use it without moderation!

References

- C++ Reference: override specifier
- Why to use the override specifier in C++ 11?

Question 16: Explain the concept of covariant return types and show a use-case where it comes in handy!

Using covariant return types for a `virtual` function and for all its overridden versions means that you can replace the original return type with something narrower, in other words, with something more specialized.

Let's say you have a CarFactoryLine producing Cars. The specialization of these factory lines might produce SUVs, SportsCars, etc. How do you

represent it in code? The obvious way is still having the return type as a Car pointer.

```
class CarFactoryLine {
 public:
  virtual Car* produce() {
    return new Car{};
  }
};

class SUVFactoryLine : public CarFactoryLine {
 public:
  virtual Car* produce() override {
    return new SUV{};
  }
};
```

This way, getting an SUV* requires a dynamic cast.

```
SUVFactoryLine sf;
Car* car = sf.produce();
SUV* suv = dynamic_cast<SUV*>(car);
```

Instead, we directly return an SUV*

```
class Car {
 public:
  virtual ~Car() = default;
};

class SUV : public Car {};

class CarFactoryLine {
 public:
  virtual Car* produce() {
   return new Car{};
  }
};

class SUVFactoryLine : public CarFactoryLine {
 public:
  virtual SUV* produce() override {
    return new SUV{};
  }
};
```

So that you can simply do this:

```
SUVFactoryLine sf;
SUV* car = sf.produce();
```

In C++, in a derived class, in an overridden function, you don't have to return the same type as in the base class, but you can return a covariant return type. In other words, you can replace the original type with a "narrower" one, in other words, with a more specified data type.

References:

- Covariant return types
- How to Return a Smart Pointer AND Use Covariance

Question 17: What is virtual inheritance in C++ and when should you use it?

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.

As you probably guessed, this technique is useful when you have to deal with multiple inheritance and you can solve the infamous diamond inheritance.

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.

```
1 struct Person {
2   virtual ~Person() = default;
3   virtual void speak() {}
```

```
};

struct Student: Person {
  virtual void learn() {}
};

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

struct TeachingAssistant: Student, Worker {};

int main() {
  TeachingAssistant aTeachingAssistant;
}
```

As declared above, 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:

```
TeachingAssistant aTeachingAssistant;
Person& aPerson = aTeachingAssistant;  // error: which Pe\
rson subobject should
                                       // a TeachingAssist\
ant cast into,
                                       // a Student::Perso\
n or a Worker::Person?
```

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

```
TeachingAssistant aTeachingAssistant;
Person& student = static_cast<Student&>(aTeachingAssistan\
t);
Person& worker = static_cast<Worker&>(aTeachingAssistant);
```

In order to call `speak()`, the same disambiguation, or explicit qualification is needed: `static_cast<Student&>(aTeachingAssistant).speak()` or `static_cast<Worker&>(aTeachingAssistant).speak()` or alternatively `aTeachingAssistant.Student::speak()` and `aTeachingAssistant.Worker::speak()`. Explicit qualification not only

uses an easier, uniform syntax for both pointers and objects but also allows for static dispatch, so it would arguably be the preferable method.

In this case, the double inheritance of `Person` is probably unwanted, as we want to model that the relation (`TeachingAssistant` is a `Person`) exists only once; that a `TeachingAssistant` is a `Student` and is a `Worker` does not imply that it is a `Person` twice (unless the `TeachingAssistant` is schizophrenic): a `Person` base class corresponds to a contract that `TeachingAssistant` implements (the "is a" relationship above really means "implements the requirements of"), and a `TeachingAssistant` only implements the `Person` contract once.

The real-world meaning of "only once" is that `TeachingAssistant` should have only one way of implementing `speak`, not two different ways, depending on whether the `Student` view of the `TeachingAssistant` is eating, or the `Worker` view of the `TeachingAssistant`. (In the first code example we see that `speak()` is not overridden in either `Student` or `Worker`, so the two `Person` subobjects will actually behave the same, but this is just a degenerate case, and that does not make a difference from the C++ point of view.)

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

```
1  struct Person {
2    virtual ~Person() = default;
3    virtual void speak() {}
4  };
5
6  // Two classes virtually inheriting Person:
7  struct Student: virtual Person {
8    virtual void learn() {}
9  };
10
11 struct Worker: virtual Person {
12   virtual void work() {}
13 };
14
15 // A teaching assistant is still a student and the worker
16 struct TeachingAssistant: Student, Worker {};
```

Now we can easily call `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.

References:

- Sandor Dargo's Blog: What is virtual inheritance in C++ and when should you use it?
- C++ Core Guidelines
- Wikipedia - Virtual inheritance

Question 18: 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.

So let's see the main drawbacks.

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 and forces you to make some additional revisions in all those classes during refactoring. All this leads to new bugs and makes the code less readable.

Troubles with type conversions may also be a source of bugs. You can partly solve the issues by using the expensive `dynamic_cast` wherever you used to use `static_cast`. Unfortunately, `dynamic_cast` is much slower, and if you have to use it too often in your code, it means that your project's architecture has quite some room for improvement.

References:

- Sandor Dargo's Blog: What is virtual inheritance in C++ and when should you use it?
- C++ Core Guidelines
- Wikipedia - Virtual inheritance

Question 19: What's the output of the following sample program? Is that what you'd expect? Why? Why not?

```
#include <iostream>
#include <memory>

class Animal {
 public:
  ~Animal() = default;
  virtual void eat(int quantity) {
    std::cout << "Animal eats " << quantity << std::endl;
  }

  void speak() {
    std::cout << "Animal speaks" << std::endl;
  }
};

class Dog : public Animal {
 public:
```

```
18   void eat(unsigned int quantity) {
19     std::cout << "Dog eats " << quantity << std::endl;
20   }
21
22   void speak() {
23     std::cout << "Dog speaks" << std::endl;
24   }
25 };
26
27 int main() {
28   Dog d;
29   d.speak();
30   d.eat(42u);
31
32   std::unique_ptr<Animal> a = std::make_unique<Dog>();
33   a->speak();
34   a->eat(42u);
35 }
```

If you copied the above piece of program into a compiler you could easily get the solution. I hope that first you tried to reason about it.

```
1 Dog speaks
2 Dog eats 42
3 Animal speaks
4 Animal eats 42
```

If this is what you expected, congratulations! Most probably then you also know why.

If you didn't figure it out and the above solution surprised you, don't worry. Check the code once more and read on.

There are two small mistakes in the original code that are so easy to ignore, and until C++11 you only had your eyes or your code reviewers' eyes to help you with.

But since C++11, you have the `override` specifier. It helps you to explicitly mark that a function in a derived class is supposed to override something from its base.

```
1 std::unique_ptr<Animal> a = std::make_unique<Dog>();
2 a->speak();
```

When we write such code, we expect `a` to invoke the behaviour defined in the derived class, in `Dog`. Yet we fall back to `Animal` behaviour.

As the `override` is missing from the Dog method signatures, let's add them, and recompile.

```
#include <iostream>
#include <memory>

class Animal {
 public:
  ~Animal() = default;
  virtual void eat(int quantity) {
    std::cout << "Animal eats " << quantity << std::endl;
  }

  void speak() {
    std::cout << "Animal speaks" << std::endl;
  }
};

class Dog : public Animal {
 public:
  void eat(unsigned int quantity) override {
    std::cout << "Dog eats " << quantity << std::endl;
  }

  void speak() override {
    std::cout << "Dog speaks" << std::endl;
  }
};

int main() {
  Dog d;
  d.speak();
  d.eat(42u);

  std::unique_ptr<Animal> a = std::make_unique<Dog>();
  a->speak();
  a->eat(42u);
}
```

According to the error messages, neither `Dog::eat` nor `Dog::speak` overrides anything.

In the first case, it's because in the base class the parameter is a simple signed `int`, while in the derived class we have a function with the same name, same return type, but with a different parameter: an `unsigned int`. We have to match the two, in case we are looking for polymorphic behaviour.

In the other case, we simply forgot to add the `virtual` keyword to the signature in the base class, hence it cannot be overridden.

Fix these two mistakes and you'll get the behaviour that we would commonly expect.

```
#include <iostream>
#include <memory>

class Animal {
 public:
  ~Animal() = default;
  virtual void eat(int quantity) {
    std::cout << "Animal eats " << quantity << std::endl;
  }

  virtual void speak() {
    std::cout << "Animal speaks" << std::endl;
  }
};

class Dog : public Animal {
 public:
  void eat(int quantity) override {
    std::cout << "Dog eats " << quantity << std::endl;
  }

  void speak() override {
    std::cout << "Dog speaks" << std::endl;
  }
};

int main() {
  Dog d;
  d.speak();
  d.eat(42u);

  std::unique_ptr<Animal> a = std::make_unique<Dog>();
  a->speak();
  a->eat(42u);
}
```

The takeaway is that you should always use the `override` specifier for functions that are meant to override base class functions so that you can catch these subtle bugs already at compile time.

Reference::

- Sandor Dargo's Blog: the `override` specifier

Question 20: Can you access the public and protected members and functions of a base class if you have private inheritance?

The access specifier of the inheritance doesn't affect the inheritance of the implementation. The implementation is always inherited based on the function's access level. The inheritance's access specifier only affects the accessibility of the class interface.

This means that all the public and protected variables and functions will be useable from the derived class even when you use private inheritance.

On the other hand, those public and protected elements of the base class will not be accessible from the outside through the derived class.

A grandchild of a base class, if its parent inherited privately from the base (the grandparent…), won't have any access to the base's members and functions. Not even if they were originally protected or even public.

```
#include <iostream>

class Base {
 public:
  Base() = default;
  virtual ~Base() = default;
  virtual int x() {
    std::cout << "Base::x()\n";
    return 41;
  }

 protected:
  virtual int y() {
    std::cout << "Base::y()\n";
    return 42;
  }
};

class Derived : private Base {
 public:
  int x() override {
    std::cout << "Derived::x()\n";
    return Base::y();
  }
};

```

```
27 class SoDerived : public Derived {
28  public:
29   int x() override {
30     std::cout << "SoDerived::x()\n";
31     return Base::y();
32     }
33 };
34 
35 int main() {
36   SoDerived* p = new SoDerived();
37   std::cout << p->x() << std::endl;
38 }
39 /*
40 main.cpp: In member function 'virtual int SoDerived::x()':
41 main.cpp:31:12: error: 'class Base Base::Base' is private\
42  within this context
43    31 |     return Base::y();
44       |            ^~~~
45 main.cpp:19:7: note: declared private here
46    19 | class Derived : private Base {
47       |       ^~~~~~~
48 main.cpp:31:19: error: 'Base' is not an accessible base o\
49 f 'SoDerived'
50    31 |     return Base::y();
51       |            ~~~~~~~^~
52 */
```

References::

- Sandor Dargo's Blog: The quest of private inheritance in C++
- C++ Core Guidelines: Inheritance — private and protected inheritance

Question 21: What is private inheritance used for?

So first, let's have a small reminder on what private inheritance is.

> The access specifier of the inheritance doesn't affect the inheritance of the implementation. The implementation is always inherited based on the function's access level. The inheritance's access specifier only affects the accessibility of the class interface.

This means that all the public and protected variables and functions will be useable from the derived class even when you use private inheritance.

On the other hand, those public and protected elements of the base class will not be accessible from the outside through the derived class.

When can this be useful?

We probably all learnt that inheritance is there for expressing *is-a* relationships, right?

If there is `Car` class inheriting from `Vehicle`, we can all say that a `Car` is a `Vehicle`. Then `Roadster` class inherits from `Car`, it's still a `Vehicle` having access to all `Vehicle` member(function)s.

But what if that inheritance between `Vehicle` and `Car` was private? Then that little shiny red `Roadster` will not have access to the interface of `Vehicle`, even if it publicly inherits from `Car` in the middle.

We simply cannot call it an *is-a* relationship anymore.

It's a *has-a* relationship. Derived classes, in this specific example `Car`, will have access to the Base class (e.g.`Vehicle`) and expose it based on the access level, protected or private. Well, this latter means that it's not exposed. It serves as a private member.

In the case of protected, you might argue that well, `Roadster` still have access to `Vehicle`, that is true.

But you cannot create a `Roadster` as a `Vehicle` when you use non-public inheritance. This line will not compile.

```
1 Vehicle* p = new Roadster();
```

Just to repeat it, non-public inheritance in C++ expresses a *has-a* relationship.

Just like composition.

So if we want to keep the analogy of cars, we can say that a `Car` can privately inherit from the hypothetical `Engine` class - while it still publicly inherits from `Vehicle`. And with this small latter addition of multiple

inheritance, you probably got the point, why composition is easier to maintain than private inheritance.

But even if you have no intention of introducing an inheritance tree, I think private inheritance is not intuitive and it's so different from most of the other languages that it's simply disturbing to use it. It's not evil at all, it'll be just more expensive to maintain.

That's exactly what you can find in the C++ Core guidelines.

> Use composition when you can, private inheritance when you have to.

But when do you **have to** use private inheritance?

According to the Core Guidelines, you have a valid use-case when the following conditions apply:

- The derived class has to make calls to (non-virtual) functions of the base
- The base has to invoke (usually pure-virtual) functions from the derived

References::

- Sandor Dargo's Blog: The quest of private inheritance in C++
- C++ Core Guidelines: Inheritance — private and protected inheritance

Question 22: Can you call a `virtual` function from a constructor or a destructor?

Technically you can, the code will compile. But the code can be misleading and it can even lead to undefined behaviour.

Attempting to call a derived class function from a base class under construction is dangerous.

```
1 #include <iostream>
2
3 class Base {
```

```
 public:
  Base() {
    foo();
  }
 protected:
  virtual void foo() {
    std::cout << "Base::foo\n";
  }
};

class Derived : public Base {
 public:
  Derived() {}
  void foo() override {
    std::cout << "Derived::foo\n";
  }
};

int main() {
  Derived d;
}
```

The output is simply `Base::foo`:

- By contract, the derived class constructor starts by calling the base class constructor.
- The base class constructor calls the base member function and not the one overridden in the child class, which is confusing for the child class' developer.

In case, the virtual method is also a pure virtual method, you have undefined behaviour. If you're lucky, at link time you'll get some errors.

What's the solution?

The simplest probably is just to fully reference the function that you'll call:

```
Base() {
  Base::foo();
}
```

From the `Base` class, you can call only the `Base` class' function. When you're in the `Derived` class, you can decide which class' version do you want to call.

A more elegant solution is if you wrap the `virtual` functions in non-`virtual` functions and from the constructors/destructors you only use the non-`virtual` wrappers.

For full source code, check out the references.

References:

- SEI CERT: OOP50-CPP. Do not invoke virtual functions from constructors or destructors
- SonarSource: RPSEC-1699

Question 23: What role does a `virtual` destructor play?

The destructor of an object is executed when the object goes out of scope. It doesn't take arguments, it cannot be overloaded, but it can be `virtual`.

Using `virtual` destructors, you can destroy objects without knowing their types — the correct destructor for the object is invoked using the virtual function mechanism. Destructors can also be declared as pure virtual functions for abstract classes.

A derived class' destructor (whether or not you explicitly define one) automatically invokes the destructors for base class subobjects. Base classes are destructed after member objects. In the event of multiple inheritance, direct base classes are destructed in the reverse order of their appearance in the inheritance list.

But we still don't know why it's needed to declare a destructor `virtual`.

It's because if you delete an object of a derived class through a pointer to its base class that has a non-`virtual` destructor, the behaviour is undefined. Instead, the base class should destructor should be declared as `virtual`.

Making base class destructor `virtual` guarantees that the object of a derived class is destructed properly, i.e. both base class and derived class

destructors are called. As a guideline, any time you have a `virtual` function in a class, you should immediately add a `virtual` destructor (even if it does nothing). This way, you ensure against any surprises later.

References:

- C++ Core guidelines
- docs.microsoft.com
- SEI CERT C++ Coding Standard

Question 24: 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.

```
1 class Squad : public std::vector {
2   using std::vector::vector;
3   // ...
4 };
```

It's simple, it's readable, yet you'll find a lot of people at different forums who will tell you that this is the eighth deadly sin and if you are a serious developer you should avoid it at all costs.

Why do they say so?

There are two main arguments. One is that algorithms and containers are well-separated concerns in the STL. The other one is about the lack of `virtual` destructors.

But are these valid concerns?

They might be. It depends.

Let's start with the one about the lack of a `virtual` destructor. It seems more practical.

Indeed, 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.

On the other hand, this also means that those who rigidly oppose inheriting from `std::vector` - or from any class without a `virtual` destructor - because of undefined behaviour and memory leaks, are not right.

If you know what you are doing, and you only use this inheritance to introduce a strongly typed vector, not to introduce polymorphic behaviour and additional states to your container, it is perfectly fine to use this technique. You simply have to respect the limitations, though probably this is not the best strategy to use in the case of a public library. But more on that just in a second.

So the other main concern is that you might mix containers and algorithms in your new object. It's bad because the creators of the STL said so. And so what? Alexander Stepanov who originally designed the STL and the others who have been later contributed to it are smart people and there is a fair chance that they are better programmers than most of us. They designed functions, objects that are widely used in the C++ community. I think it's okay to say that they are used by everyone.

Most probably we are not working under such constraints, we are not preparing something for the whole C++ community. We are working on

specific applications with very strict constraints. Our code will not be reused as such. Never. Most of us don't work on generic libraries, we work on one-off business applications.

As long as we keep our code clean (whatever it means), it's perfectly fine to provide a non-generic solution.

As a conclusion, we can say that for application usage, inheriting from containers in order to provide strong typing is fine, as long as you don't start to play with polymorphism.

Reference:

- Sandor Dargo's blog: Strong types for containers

Question 25: What does a strong type mean and what advantages does it have?

A strong type carries extra information, a specific meaning through its name. While you can use booleans or strings everywhere, the only way they carry can carry meaning is tthrough the name of the variables.

The main advantages of strong typing are readability and safety.

If you look at this function signature, perhaps you think it's alright:

```
1 Car::Car(unit32_t horsepower, unit32_t numberOfDoors,
2          bool isAutomatic, bool isElectric);
```

It has relatively good names, so what is the issue?

Let's look at a possible instantiation.

```
1 auto myCar{Car(96, 4, false, true)};
```

Yeah, what?

God knows…

And you too! But only if you take your time to actually look up the constructor and do the mind mapping. Some IDEs can help you visualizing parameter names, like if they were Python-style named parameters, but you shouldn't rely on that.

Of course, you could name the variables as such:

```
1 constexpr unit32_t horsepower = 96;
2 constexpr unit32_t numberOfDoors = 4;
3 constexpr bool isAutomatic = false;
4 constexpr bool isElectric = false;
5
6 auto myCar = Car{horsepower, numberOfDoors,
7                  isAutomatic, isElectric};
```

Now you understand right away which variable represents what. You have to look a few lines upper to actually get the values, but everything is in sight. On the other hand, this requires willpower. Discipline. You cannot enforce it. Well, you can be a thorough code reviewer, but you won't catch every case and anyway, you won't be there all the time.

Strong typing is there to help you!

Imagine the signature as such:

```
1 Car::Car(Horsepower hp, DoorsNumber numberOfDoors,
2          Transmission transmission, Fuel fuel);
```

Now the previous instantiation could look like this:

```
1 auto myCar{Car{Horsepower{98u}}, DoorsNumber{4u},
2            Transmission::Automatic, Fuel::Gasoline};
```

This version is longer and more verbose than the original version - which was quite unreadable -, but much shorter than the one where we introduced well-named temporary variables for each parameter.

So one advantage of strong typing is readability and one other is safety. It's much harder to mix up values. In the previous examples, you could have easily mixed up door numbers with performance, but by using strong typing, that would actually lead to a compilation error.

References:

- Fluent Cpp
- SandorDargo's Blog
- Correct by Construction: APIs That Are Easy to Use and Hard to Misuse - Matt Godbolt

Question 26: Explain short-circuit evaluation

What's the output of the following piece of code and why?

```cpp
#include <iostream>

bool a() {
  std::cout << "a|";
  return false;
}

bool b() {
  std::cout << "b|";
  return true;
}

int main() {
  if (a() && b()) {
    std::cout << "main";
  }
}
```

The output will simply be `a|`.

In `main()`, in the `if` statement, the first subexpression `a()` is called and it prints `a|`. As it's evaluated to `false`, and therefore the whole expression cannot be `true`, the second subexpression (`b()`) is not executed.

In C++, like in many other programming languages there is short-circuit evaluation meaning that once it's for sure that the whole condition cannot be fulfilled, the evaluation stops.

As the two subexrepssions are joined with `&&`, both sides must be `true` in order to have the full expression `true`:

a()	b()	a() && b()
false	false	false
false	true	false
true	false	false
true	true	true

In case, `a()` is not true, the rest of the expression is not evaluated to save some precious CPU cycles.

This short-circuiting is also a reason - besides sheer readability -, why calls in a condition should not have side effects (changing states of an object/variable). When a boolean expression is evaluated it should simply return a boolean. The caller cannot be sure whether it would be evaluated.

I know this sounds simple and probably we all learnt this at the beginning of our C++ career. But there is something else to remember. When you define the logical operators for your own class, so when you overload for example `operator&&`, you lose short-circuiting.

In case you have something like `myObjectA && myObjectB`, both sides will be evaluated, even if the first one evaluates to false.

The reason is that before calling the overloaded `operator&&`, both the left-hand-side and the right-hand-side arguments of the overloaded function are evaluated. A function call is a sequence-point and therefore all the computations and the side-effects are complete before making the function call. This is an eager strategy.

References:

- Cpp Truths
- StackOverflow

Question 27: 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 (e.g. `~MyClass`). 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.

On the other hand, a destructor can be `virtual`, but that's the topic for tomorrow.

References:

- C++ Reference: Destructor
- C++ Reference: The rule of three/five/zero

Question 28: What is the output of the following piece of code and why?

```
1 #include <iostream>
2
```

```
3  class A {
4   public:
5    int value() { return 1; }
6    int value() const { return 2; }
7  };
8
9  int main() {
10   A a;
11   const auto b = a.value();
12   std::cout << b << '\n';
13 }
```

The output is going to be 1. If `a` would have been declared as `const` and `b` not, the result would be 2. In fact, the result doesn't depend on `b`'s constness at all.

Why is that?

If a function has two overloaded versions where one is `const` and the other is not, the compiler will choose which one to call based on whether the object is `const` or not.

On the other hand, it has nothing to do with the constness of the variable returned.

This was just a simple introduction to the coming days when we are going to talk about constness, `const` correctness and how to use the `const` keyword.

Reference:

- C++ Core Guidelines: What's the deal with "const-overloading"

Question 29: How to use the `= delete` specifier in C++?

The question could also be how to disallow implicit conversions for function calls?

You have a function taking integer numbers. Whole numbers. Let's say it takes as a parameter how many people can sit in a car. It might be 2, there are

some strange three-seaters, for some luxury cars it's 4 and for the vast majority, it's 5. It's not 4.9. It's not 5.1 or not even 5 and a half. It's 5. We don't traffic body parts.

How can you enforce that you only receive whole numbers as a parameter?

Obviously, you'll take an integer parameter. It might be `int`, even `unsigned` or simply a `short`. There are a lot of options. You probably even document that the `numberOfSeats` parameter should be an integral number.

Great!

So what happens if the client call still passes a float?

```
1 #include <iostream>
2
3 void foo(int numberOfSeats) {
4   std::cout << "Number of seats: " << numberOfSeats << '\\
5 n';
6   // ...
7 }
8
9 int main() {
10   foo(5.6f);
11 }
12 /*
13 Number of seats: 5
14 */
```

The floating-point parameter is accepted and narrowed down into an integer. You cannot even say that it's rounded, it's implicitly converted into an integer.

You might say that this is fine and in certain situations it probably is. But in others, this behaviour is simply not acceptable.

What can you do in such cases to avoid this problem?

Obviously, you can handle it on the caller side, but

- if `foo` is often used, it'd be tedious to do checks
- if `foo` is part of an API used by the external world, it's out of your control

Since C++11, we can use the delete specifier in order to restrict certain types for copying of moving, and in fact even from polymorphic usage.

But `= delete` can be used for more. It can be applied to any function, should they be members or free.

If you don't want to allow implicit conversions from floating-point numbers, you can simply delete foo's overloaded version with a float:

```
#include <iostream>

void foo(int numberOfSeats) {
  std::cout << "Number of seats: " << numberOfSeats << '\\
n';
  // ...
}

void foo(double) = delete;
int main() {
  foo(5.6f);
}
/*
main.cpp: In function 'int main()':
main.cpp:10:6: error: use of deleted function 'void foo(d\
ouble)'
   10 |   foo(5.6f);
      |   ~~~^~~~~
main.cpp:8:6: note: declared here
    8 | void foo(double) = delete;
      |      ^~~

*/
```

That's it. By deleting some overloads of a function, you can forbid implicit conversions from certain types to the types you expect. Now, you are in complete control of what kind of parameters your users can pass to your API.

Reference:

- Sandor Dargo's blog: Three ways to use the = delete specifier in C++

Lambda functions

The next two questions will be about lambda functions, one of the most important features of C++11. These questions assume that you understand what lambda functions are. If that's not the case, check out this introduction.

Question 30: What are immediately invoked lambda functions?

They are lambda functions that are invoked immediately one could say. Ok, but what does it mean **immediately**?

It means, that the lambda is not even stored and denoted by a variable, but where you create it, you invoke it right away.

This is not an immediately invoked lambda:

```
1 auto l = [](){return 42;}; // Here we created a simple la\
2 mbda
3
4 int fortyTwo = l(); // Here we invoke it
```

In the above case, you can reuse the lambda, you can pass it around, you can invoke it as often as you want, as you need it.

On the other hand, in the following example, you immediately invoke it, which implies that you don't store the lambda itself. By definition, IILFs cannot be stored. If they are stored, they are not invoked immediately.

```
1 auto fortyTwo = [](){return 42;}(); // those parantheses \
2 at the end invoke the lambda expression!
```

Why is this concept powerful and worth mentioning?

It helps to perform complex initialization of variables. This is important because a good way to help the C++ compiler is to declare all variables not

supposed to change as `const`.

In most cases, it's very easy, you just put `const` next to the type and initialize your variable right on the spot. But you might run into situations where you have different possible values at your hands.

Let's start with a simple example.

```
1 // Bad Idea
2 std::string someValue;
3
4 if (caseA) {
5   return std::string{"Value A"};
6 } else {
7   return std::string{"Value B"};
8 }
```

This is bad, because as such `someValue` is not `const`. Can we make it `const`? Sure we can. We might use a ternary operator.

```
1 const std::string someValue = caseA ? std::string{"Value \
2 A"} : std::string{"Value B"};
```

Easy peasy.

But what to do if there are 3 different possibilities or even more?

You have different options and among those IIFLs is one.

```
1 const std::string someValue = [caseA, caseB] () {
2   if (caseA) {
3     return std::string{"Value A"};
4   } else if (caseB) {
5     return std::string{"Value B"};
6   } else {
7     return std::string{"Value C"};
8   }
9 }();
```

This way, you can perform complex initializations of `const` variables, yet you don't have to find a proper place nor a name for the initialization logic.

Performance-wise we get all the performance gain of having a const variable and we lose nothing compared to ternaries or helper functions. You can find

more details regarding the performance analysis here.

References:

- Sandor Dargo's Blog: Immediately Invoked Lambda Functions
- My Least Favorite Anti-Pattern by Conor Hoekstra

Question 31: What kind of captures are available for lambda expressions?

First of all, a lambda expression looks like this:

capture -> returnType

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.

The following types of captures are available:

- By-copy capture (since C++11)

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

- By-copy capture that is a pack expansion (since C++11)

```
1 template <typename Args>
2 void f(Args... args) {
3   auto l = [args...] {
```

```
    return g(args...);
  };
  l();
}
```

* by-reference capture (since C++11)

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

* by-reference capture that is a pack expansion (since C++11)

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

- by-reference capture of the current object (since C++11)

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

- by-copy capture with an initializer (since C++14)

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

- by-reference capture with an initializer (since C++14)

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

- by-copy capture of the current object (since C++17)

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

- by-copy capture with an initializer that is a pack expansion (since C++20)

```
template <typename Args>
auto delay_invoke_foo(Args... args) {
  return [...args=std::move(args)]() -> decltype(auto) {
    return foo(args...);
  };
}
```

- by-reference capture with an initializer that is a pack expansion (since C++20)

```
template <typename Args>
auto delay_invoke_foo(Args... args) {
  return [&...args=std::move(args)]() -> decltype(auto) {
    return foo(args...);
  };
}
```

References:

- C++ Reference: Lambda expressions
- Learn C++: Lambda captures

How to use the `const` qualifier in C++

In this chapter, during the next couple of questions, we'll learn about the `const` qualifier and its proper usage.

Question 32: What is the output of the following piece of code and why?

```
#include <iostream>

class A {
 public:
  int value() { return 1; }
  int value() const { return 2; }
};

int main() {
  A a;
  const auto b = a.value();
  std::cout << b << '\n';
}
```

The output is going to be 1. If `a` would have been declared as const and `b` not, the result would be 2. In fact, the result doesn't depend on `b`'s constness.

Why is that?

If a function has two overloaded versions where one is `const` and the other is not, the compiler will choose which one to call based on whether the object is const or not.

On the other hand, it has nothing to do with the constness of the variable returned.

This was just a simple introduction to the coming days when we are going to talk about constness and const correctness. That's an important topic in C++,

as using the `const` keyword.

Reference:

- Tutorials Point: Function overloading and const keyword in C++

Question 33: 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.

In real life, I find that we tend to forget the value making variables const, even though there are good examples at conference talks and it really has no bad effect on your code, on maintainability.

This is such an important idea that in Rust, all your variables are declared as `const` unless you say they should be mutable.

We have no reason not to follow similar practices.

Declare your local variables `const` if you don't plan to modify them. Regarding global variables, well, avoid using them, but if you do, also make them `const` whenever possible.

Reference:

- CppCon 2016: Jason Turner “Rich Code for Tiny Computers: A Simple Commodore 64 Game in C++17”
- Sandor Dargo’s Blog: When to use const in C++? Part I: functions and local variables

Question 34: Is it a good idea to have `const` members in a class?

Why would you have `const` members in the first place?

Because you might want to signal that they are immutable, that they should never change.

Unfortunately, there are some implications.

The first is that classes with const members are not assignable:

```
1  class MyClassWithConstMember {
2   public:
3    MyClassWithConstMember(int a) : m_a(a) {}
4
5   private:
6    const int m_a;
7  };
8
9  int main() {
10   MyClassWithConstMember o1{666};
11   MyClassWithConstMember o2{42};
12   o1 = o2;
13 }
14 /*
15 main.cpp: In function 'int main()':
16 main.cpp:12:8: error: use of deleted function 'MyClassWit\
17 hConstMember& MyClassWithConstMember::operator=(const MyC\
18 lassWithConstMember&)'
19    12 |   o1 = o2;
20       |        ^~
21 main.cpp:1:7: note: 'MyClassWithConstMember& MyClassWithC\
22 onstMember::operator=(const MyClassWithConstMember&)' is \
23 implicitly deleted because the default definition would b\
24 e ill-formed:
25     1 | class MyClassWithConstMember {
26       |       ^~~~~~~~~~~~~~~~~~~~~~
27 main.cpp:1:7: error: non-static const member 'const int M\
28 yClassWithConstMember::m_a', cannot use default assignmen\
```

```
29 t operator
30 */
```

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.

```
1 #include <utility>
2 #include <iostream>
3
4 class MyClassWithConstMember {
5  public:
6   MyClassWithConstMember(int a) : m_a(a) {}
7   MyClassWithConstMember& operator=(const MyClassWithCons\
8 tMember& other) {
9     int* tmp = const_cast<int*>(&m_a);
10    *tmp = other.m_a;
11    std::cout << "copy assignment\n";
12    return *this;
13  }
14
15  int getA() {return m_a;}
16
17 private:
18  const int m_a;
19 };
20
21 int main() {
```

```
22   MyClassWithConstMember o1{666};
23   MyClassWithConstMember o2{42};
24   std::cout << "o1.a: " << o1.getA() << std::endl;
25   std::cout << "o2.a: " << o2.getA() << std::endl;
26   o1 = o2;
27   std::cout << "o1.a: " << o1.getA() << std::endl;
28 }
```

Is this worth it?

You have your `const` member, fine. You have the assignment working, fine. Then if anyone comes later and wants to do the same "magic" outside of the special functions, for sure, it would be a red flag in a code review.

Let's not commit this so quickly.

Having this "magic" in the special functions is already a red flag! `const_cast` might lead to undefined behaviour:

> $5.2.11/7 - Note: Depending on the type of the object, a write operation through the pointer, lvalue or pointer to data member resulting from a const_cast that casts away a const-qualifier may produce undefined behaviour (7.1.5.1).

We've just had a look at the copy assignment and it wouldn't work without risking undefined behaviour.

It's not worth it!

References:

- Fluent C++: Compiler-generated Functions, Rule of Three and Rule of Five
- Sandor Dargo's Blog: When to use const in C++? Part II: member variables

Question 35: Does it make sense to return `const` objects by value?

Most probably it won't make so much sense.

Why is that?

Putting `const` somewhere shows the reader (and the compiler of course) that something should not be modified. When we return something by value it means that a copy will be made for the caller. Okay, you might have heard about copy elision and its special form, return value optimization (RVO), but essentially we are still on the same page. The caller gets his own copy.

Does it make sense to make that own copy `const`?

Imagine that you buy a house but you cannot modify it? While there can be special cases, in general, you want your house to be your castle. Similarly, you want your copy to really be **your** object and you want to be able to do with it just whatever as an owner of it.

It doesn't make sense and it's misleading to return by value a `const` object.

Not just misleading, but probably even hurting you.

Even hurting? How can it be?

Let's say you have this code:

```
1  class SgWithMove {
2    // ...
3  };
4
5  SgWithMove foo() {
6    // ...
7  }
8
9  int main() {
10   SgWithMove o;
11   o = foo();
12 }
```

By using a debugger or by adding some logging to your special functions, you can see that RVO was perfectly applied and there was a move operation taking place when `foo()`s return value was assigned to `o`.

Now let's add that infamous `const` to the return type.

```
1  class SgWithMove {
2    // ...
3  };
4
5  const SgWithMove bar() {
6    // ...
7  }
8
9  int main() {
10   SgWithMove o;
11   o = bar();
12 }
```

Following up with the debugger we can see that we didn't benefit from a move, but actually, a copy was made.

We are returning a `const SgWithMove` and that is something we cannot pass as `SgWithMove&&`. It would discard the `const` qualifier. (A move would alter the object being moved) Instead, the copy assignment (`const SgWithMove&`) is called and we just made another copy.

Please note that there are important books advocating for returning user-defined types by const value. They were right in their own age, but since then C++ went through a lot of changes and this piece of advice became obsolete.

References:

- C++ Reference: Copy elision
- C++ Reference: When to use const in C++? Part III: return types

Question 36: How should you return `const` pointers from a function?

Pointers are similar to references in the sense that the pointed object must be alive at least as long as the caller wants to use it. You can return the address of a member variable if you know that the object will not get destroyed as long as the caller wants the returned address. What is important to emphasize once again is that we can never return a pointer to a locally initialized variable.

But even that is not so self-evident. Let's step back a little bit.

What do we return when we return a pointer?

We return a memory address. The address can be of anything. Technically it can be a random place, it can be a null pointer or it can be the address of an object. (OK, a random place can be the address of a valid object, but it can be simply garbage. After all, it's random.)

Even if we talk about an object that was declared in the scope of the enclosing function, that object could have been declared either on the stack or on the heap.

If it was declared on the stack (no `new`), it means that it will be automatically destroyed when we leave the enclosing function.

If the object was created on the heap (with `new`), that's not a problem anymore, the object will be alive, but you have to manage its lifetime. Except if you return a smart pointer, but that's beyond the scope of this article.

So we have to make sure that we don't return a dangling pointer, but after that, does it make sense to return a `const` pointer?

```
int * const func () const
```

The function is constant, and the returned pointer is constant but the data we point at can be modified. However, I see no point in returning a `const` pointer because the ultimate function call will be an rvalue, and rvalues of non-class type cannot be `const`, meaning that `const` will be ignored anyway.

```
const int* func () const
```

This is a useful thing. The pointed data cannot be modified.

```
const int * const func() const
```

Semantically this is almost the same as the previous option. The data we point at cannot be modified. On the other hand, the constness of the pointer itself will be ignored.

So does it make sense to return a `const` pointer? It depends on what is `const`. If the constness refers to the pointed object, yes it does. If you try to make the pointer itself `const`, it doesn't make sense as it will be ignored.

Reference:

- Sandor Dargo's Blog: When to use const in C++? Part III: return types

Question 37: Should functions return `const` references?

It depends, you might face an issue. Maybe you'll have a dangling reference. The problem with returning `const` references is that the returned object has to outlive the caller. Or at least it has to live as long.

```
1 void f() {
2   MyObject o;
3   const auto& aRef = o.getSomethingConstRef();
4   aRef.doSomething();
5 }
```

Will that call work? It depends. If `MyObject::getSomethingConstRef()` returns a `const` reference of a local variable it will not work. It is because that local variable gets destroyed immediately once we get out of the scope of the function.

```
1 const T& MyObject::getSomethingConstRef() {
2   T ret;
3   // ...
4   return ret; // ret gets destroyed right after, the retu\
5 rned reference points at its ashes
6 }
```

This is what is called a dangling reference.

On the other hand, if we return a reference to a member of `MyObject`, there is no problem in our above example.

```
class MyObject {
 public:
  // ...

  const T& getSomethingConstRef() {
    // ...
    return m_t; // m_t lives as long as our MyObject inst\
ance is alive
  }
 private:
  T m_t;
};
```

It's worth noting that outside of `f()` we wouldn't be able to use `aRef` as the instance of `MyObject` gets destroyed at the end of the function `f()`.

So shall we return const references?

As so often the answer is it depends. Definitely not automatically and by habit. We should return constant references only when are sure that the referenced object will be still available by the time we want to reference it.

At the same time:

Never return locally initialized variables by reference!

References:

- Wikipedia: Dangling Reference
- Sandor Dargo's Blog: When to use const in C++? Part III: return types

Question 38: 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.

```
1 void setFoo(const int foo) {
2   this->m_foo = foo;
3 }
```

I know this seems like overkill, but it doesn't hurt, it's explicit and you don't know how the method would grow in the future. Maybe there will be some additional checks done, exception handling, and so on.

And if it's not marked as `const`, maybe someone will accidentally change its value and cause some subtle errors.

If you mark `foo` const, you make this scenario impossible.

What's the worst thing that can happen? You'll actually have to remove the `const` qualifier, but you'll do that intentionally.

On the other hand, if you have to modify the parameter, don't mark it as `const`.

From time to time, you can see the following pattern:

```
1 void doSomething(const int foo) {
2   // ...
3   int foo2 = foo;
4   foo2++;
5   // ...
6 }
```

Don't do this. There is no reason to take a `const` value if you plan to modify it. One more variable on the stack in vain, one more assignment without any reason. Simply take it by value.

```
1 void doSomething(int foo) {
2   // ...
3   foo++;
4   // ...
5 }
```

So we don't take fundamental data types by `const&` and we only mark them `const` when we don't want to modify them.

Reference:

- Sandor Dargo's Blog: When to use const in C++? Part IV: parameters

Question 39: Should you pass objects by `const` reference as a function parameter?

If we'd take a class by value as a parameter, it'd mean that we would make a copy of them. In general, copying an object is more expensive than just passing a reference around.

So the rule of thumb to follow is not to take an object by value, but by `const&` to avoid the copy.

Obviously, if you want to modify the original object, then you only take it by reference and omit the const.

You might take an object by value if you know you'd have to make a copy of it.

```
1 void doSomething(const ClassA& foo) {
2   // ...
3   ClassA foo2 = foo;
4   foo2.modify();
5   // ...
6 }
```

In that case, just simply take it by value. We can spare the cost of passing around a reference and the mental cost of declaring another variable and calling the copy constructor.

Although it's worth noting that if you are accustomed to taking objects by `const&` you might have done some extra thinking whether passing by value was on purpose or by mistake.

So the balance of extra mental efforts is questionable.

```
1 void doSomething(ClassA foo) {
2   // ...
3   foo.modify();
4   // ...
5 }
```

You should also note that there are objects where making the copy is less expensive or similar to the cost of passing a reference. It's the case for Small String Optimization or for `std::string_view`. This is beyond the scope of today's lesson.

For objects, we can say that by default we should take them by `const reference` and if we plan to locally modify them, then we can consider taking them by value. But never by `const` value, which would force a copy but not let us modify the object.

Reference:

- Sandor Dargo's Blog: When to use const in C++? Part IV: parameters

Question 40: Does the signature of a function declaration has to match the signature of the function definition?

No, it does not. At least not always. The `const` qualifier for parameters is not always relevant for function definitions, but it is for function declarations.

For example, the following code is valid:

```
#include <iostream>

class MyClass {
 public:
  void f(const int);
};

void MyClass::f(int a) {
  a = 42;
  std::cout << a << std::endl;
}

int main() {
  int a=5;
  MyClass c;
  c.f(a);
}
```

The `const` in the function declaration would imply that it won't modify what you pass in. But it can. At the same time, it's not a big deal, as it will only modify its own copy, not the original variable.

Please note that as soon as we turn the parameter into a reference in both the declaration and the definition, the code stops compiling.

```
#include <iostream>

class MyClass {
 public:
  void f(const int&);
};

void MyClass::f(int& a) {
  a = 42;
  std::cout << a << std::endl;
}

int main() {
  int a=5;
  MyClass c;
  c.f(a);
  std::cout << a << std::endl;
}
/*
main.cpp:8:6: error: no declaration matches 'void MyClass\
::f(int&)'
    8 | void MyClass::f(int& a) {
      |      ^~~~~~~
main.cpp:5:8: note: candidate is: 'void MyClass::f(const \
```

```
int&)'
    5 |   void f(const int&);
      |        ^
main.cpp:3:7: note: 'class MyClass' defined here
    3 | class MyClass {
      |       ^~~~~~
*/
```

But even for our original use-case, `clang-tidy` warns:

> Parameter 1 is const-qualified in the function declaration; const-qualification of parameters only has an effect in function definitions

Reference:

- How to use const in C++ by Sandor Dargo

Question 41: Explain what `consteval` and `constinit` bring to C++?

C++11 introduced `constexpr` expressions that might be evaluated at compile-time.

C++20 introduces two new related keywords, `consteval` and `constinit`.

`consteval` can be used with functions:

```
consteval int sqr(int n) {
  return n * n;
}
```

`consteval` functions are **guaranteed to be executed at compile-time**, thus they create compile-time constants. They cannot allocate or deallocate data, nor they can interact with `static` or thread-local variables.

`constinit` can be applied to variables with static storage duration or thread storage duration. So a local or member variable cannot be `constinit`. What it guarantees is that the initialization of the variable happens at compile-time.

It must be noted that while `constexpr` and `const` variables cannot be changed once they are assigned, a `constinit` variable is not constant, its value can change.

I hope you enjoyed our journey around constness.

References:

- CppReference: consteval specifier
- CppReference: constinit specifier
- Modernes C++: Two new Keywords in C++20: consteval and constinit

Some best practices in modern C++

Now let's switch and discuss some miscellaneous practices that modern C++ brought for us.

Question 42: What is aggregate initialization?

Aggregate initialization or brace-initialization (sometimes written as {}-initialization) was added to the language in C++11.

So what is an aggregate?

It's either an array type or a class type with some restrictions that you can read here.

In its simplest form, it looks exactly like normal constructor calls with parentheses, but this time with braces.

```
1 std::string text{"abcefg"};
2
3 Point a{5,4,3}; //Point is class taking 3 integers as par\
4 ameters
```

You can also use it with `auto`:

```
1 auto text = std::string{"abcefg"};
2
3 auto a = Point{5,4,3}; // Point is class taking 3 integer\
4 s as parameters\n
```

Aggregate initialization can be also used for initliazing containers:

```
1 std::vector<int> numbers{1, 2, 3, 4, 5};
2
3 auto otherNumbers = std::vector<int>{6, 7, 8, 9, 10};
```

This is much better than the previous ways of initialization, where after creating the container we had to add the elements one by one. With aggregate initialization, we can easily create `const` containers if we know that they won't change anymore. That's something that was not possible before C++11.

```
1 // Before C++11
2
3 std::vector<int> myNums;
4 myNums.push_back(3);
5 myNums.push_back(42);
6 myNums.push_back(51);
7
8 // From C++11
9 const std::vector<int> myConstNums{3, 42, 51};
```

But this is not everything. {}-initialization has two advantages.

It's not prone to most vexing parse

According to the C++ standard whatever can be interpreted as a declaration, it will be interpreted as one.

As such `MyClass o();` is not a command to create an instance of `MyClass` with the name `o` calling the default constructor of `MyClass`. Instead, it's a declaration of a function called `o` that takes no parameters but returns a `MyClass` type.

If one uses braces, it won't be interpreted as a declaration, but rather a variable declaration as in 99% of the cases the developer intended.

It doesn't allow narrowing conversions

Narrowing or more precisely narrowing conversion is an implicit conversion of arithmetic values including a loss of accuracy. Depending on your circumstances, that can be extremely dangerous.

The following examples show the problem with the classical initialization for numerical types. It doesn't matter whether we use direct initialization or assignment.

```cpp
#include <iostream>

int main() {
  int i1(3.14);
  int i2=3.14;
  std::cout << "i1: " << i1 << std::endl;
  // i1: 3
  std::cout << "i2: " << i2 << std::endl;
  // i2: 3
  int i3{3.14} // error: narrowing conversion of '3.14000\
000000000
  unsigned int i4{-41}; // error: narrowing conversion of\
 '-41'
}
```

As we can see, with brace initialization the compiler informs us about the narrowing in front of a nice error.

So the three advantages of {}-initialization are

- Direct initialization of containers
- No more most vexing parse
- Detection of implicit narrowing conversions

Reference:

- C++ Reference: Aggregate initialization

Question 43: 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.

Let's take this example:

```
class WrappedInt {
 public:
  // single parameter constructor, can be used as an impl\
icit conversion
  WrappedInt(int number) : m_number (number) {}

  // using explicit, implicit conversions are not allowed
  // explicit WrappedInt(int number) : m_number (number) \
{}

  int GetNumber() { return m_number; }
 private:
  int m_number;
};

void doSomething(WrappedInt aWrappedInt) {
  int i = aWrappedInt.GetNumber();
}

int main() {
  doSomething(42);
}
```

The argument is not a `WrappedInt` object, just a plain old `int`. However, there exists a constructor for `WrappedInt` that takes an `int` so this constructor can be used to convert the parameter to the correct type.

The compiler is allowed to do this once for each parameter.

Prefixing the explicit keyword to the constructor prevents the compiler from using that constructor for implicit conversions. Adding it to the above class will create a compiler error at the function call `doSomething(42)`. It is now necessary to call for conversion explicitly with `doSomething(WrappedInt(42))`.

The reason you might want to do this is to avoid accidental construction that can hide bugs. Let's think about strings. You have a `MySpecialString` class with a constructor taking an `int`. This constructor creates a string with the size of the passed in `int`. You have a function `print(const MySpecialString&)`, and you call print(3) (when you actually intended to call print("3")). You expect it to print "3", but it prints an empty string of length 3 instead.

A third example, is also about strings, let's say this is `MySpecialString`:

```
class MySpecialString {
 public:
  MySpecialString(int n); // allocate n bytes to the MySp\
ecialString object
  MySpecialString(const char *p); // initializes object w\
ith char *p
};
```

What happens with this constructor call?

```
MySpecialString mystring = 'x';
```

The character `x` will be implicitly converted to int and then the MySpecialString(int) constructor will be called. But, this is not what the user might have intended. So, to prevent such conditions, we shall define the constructor as explicit:

```
class MySpecialString {
 public:
  explicit MySpecialString (int n); //allocate n bytes
  MySpecialString(const char *p); // initializes object w\
ith string p
};
```

To avoid such subtle bugs, one should always consider making single argument constructors explicit initially (more or less automatically), and removing the explicit keyword only when the implicit conversion is wanted by design.

Reference:

- C++ Reference: explicit specifier

Question 44: 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.

They can be used for conversions, you can write a degrees-to-radian converter:

```
constexpr long double operator"" _deg ( long double deg )\
 {
  return deg * 3.14159265358979323846264L / 180;
}
//...
std::cout << "50 degrees as radians: " << 50.0_deg << std\
::endl;

// 50 degrees as radians: 0.872665
```

But what is probably more interesting is how it can help readability and safety when you use strong types.

Let's take this example from an earlier lesson:

```
auto myCar{Car{Horsepower{98u}}, DoorsNumber{4u},
           Transmission::Automatic, Fuel::Gasoline};
```

It's not very probable that you'd mix up the number of doors with the power of the engine, but there can be other systems to measure performance or other numbers taken by the `Car` constructor.

A very readable and safe way to initialize `Car` objects with hardcoded values (so far example in unit tests) is via user-defined literals:

```
//...
Horsepower operator"" _hp(int performance) {
  return Horsepower{performance};
}

DoorsNumber operator"" _doors(int numberOfDoors) {
  return DoorsNumber{numberOfDoors};
}
//...
auto myCar{Car{98_hp, 4_doors,
           Transmission::Automatic, Fuel::Gasoline};
```

Of course, the number of ways to use user-defined literals is limited only by your imagination, but numerical conversions, helping strong typing are definitely two of the most interesting ones.

References:

- C++ Reference: User-defined literals
- Modernes C++: User-Defined Literals

Question 45: 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:

```
#include <iostream>

void foo(int) {
  std::cout << "foo(int) is called" << std::endl;
}

void foo(bool) {
  std::cout << "foo(bool) is called" << std::endl;
}

void foo(void*) {
  std::cout << "foo(void*) is called" << std::endl;
}

int main() {
  foo(0); // calls foo(int), not foo(void*)
  foo(NULL); // might not compile, but
             // typically calls foo(int), never foo(void*)
}
```

Hence was created the guideline to avoid overloading on the pointer and integral types.

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.

As such, with `nullptr` the pointer overload can be called:

```
1 foo(nullptr);  // calls foo(void*) overload
```

For further advantages related to `nullptr`, we'd have to experiment with template metaprogramming. For further details, I urge you to check Item 8 or Effective Modern C++.

References:

- C++ Reference
- Effective Modern C++ by Scott Meyers

Question 46: What advantages does `alias` have over `typedef`?

First, let's see what is a typedef. In case, you want to avoid writing all the time complex typenames, you can introduce a "shortcut", a `typedef`:

```
typedef std::unique_ptr<std::unordered_map<std::string,
std::string>> MyStringMap;
```

Since C++11, you can use alias declarations instead: `using MyStringMap = std::unique_ptr<std::unordered_map<std::string, std::string>>`

They are doing exactly the same thing, but alias declarations offer a couple of advantages.

In case of function pointers, they are more readable:

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

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

```
template<typename T> using MyAllocList = std::list<T,
MyAlloc<T>>;
```

With typedefs it's much more obscure, and in some cases (check here) you even have to use the "::type" suffix and in templates, the "typename" prefix is often required to refer to typedefs.

C++14 even offers alias templates for all the C++11 type traits transformations, such as `std::remove_const_t<T>`, `std::remove_reference_t<T>` or `std::add_lvalue_reference_t<T>`.

References:

- C++ Reference - Type Alias
- C++ Reference - Typedef specifier
- Effective Modern C++ by Scott Meyers

Question 47: What are the advantages of scoped `enum`s over unscoped `enum`s?

C++98-style enums are now known as unscoped `enum`s. You'd declare them as such:

```
1 enum Transmission { manual, automatic };
2
3 // compilation error, as manual is already declared
4 // as a different kind of entity
5 auto manual = false;
```

The modern scoped `enum`s use the `class` keyword and the enumerators are visible only within the `enum`. They convert to other types only with a cast. They are usually called either *scoped enums* or *enum classes*.

```
1 enum class Transmission { manual, automatic };
2
3 // now compiles, manual has not been declared
4 auto manual = false;
```

```
5
6 // error, no enumerator
7 // manual without the enum scope is just a bool
8 Transmission t = manual;
9
10 Transmission t = Transmission::manual; // OK
11 auto t = Transmission::manual; // OK
```

Both scoped and unscoped `enum`s support the specification of the underlying type. The default underlying type for scoped `enum`s is `int`. Unscoped enums have no default underlying type.

The specification happens with the same syntax:

```
1 enum Status: std::uint32_t { /*..*/ };
2 enum class Status: std::uint32_t { /*...*/ };
```

Unscoped `enum`s may be forward-declared only if their declaration specifies an underlying type, otherwise, the size of an `enum` is unknown. On the other hand, as scoped `enum`s by default have an underlying type, they can be always forward declared.

References:

- C++ Reference
- Effective Modern C++ by Scott Meyers

Question 48: Should you explicitly delete unused/unsupported special functions or declare them as private?

The first question to answer is why would you do any of the options? 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.

```
1 class NonCopyable {
2  public:
3   NonCopyable() {/*...*/}
4
```

```
  // ...

 private:
  NonCopyable(const NonCopyable&); //not defined
  NonCopyable& operator=(const NonCopyable&); //not defin\
ed
};
```

Before C++11 there was no other option than declaring the unneeded special functions private and not implementing them. As such one could disallow copying objects (there was no move semantics available back in time). The lack of implementation/definition helps against accidental usages in members, friends, or when you disregard the access specifiers. You'll face a problem at linking time.

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

```
class NonCopyable {
 public:
  NonCopyable() {/*...*/}

  NonCopyable(const NonCopyable&) = delete;
  NonCopyable& operator=(const NonCopyable&) = delete;

  // ...
 private:
  // ...
};
```

The C++11 way is a better approach because

- it's more explicit than having the functions in the private section which might only be a mistake
- 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.

References:

- C++ Reference
- Effective Modern C++ by Scott Meyers

Question 49: How to use the = delete specifier in C++?

The question could also be how to disallow implicit conversions for function calls?

You have a function taking integer numbers. Whole numbers. Let's say it takes as a parameter how many people can sit in a car. It might be 2, there are some strange three-seaters, for some luxury cars it's 4 and for the vast majority, it's 5. It's not 4.9. It's not 5.1 or not even 5 and a half. It's 5. We don't traffic body parts.

How can you enforce that you only receive whole numbers as a parameter?

Obviously, you'll take an integer parameter. It might be `int`, even `unsigned` or simply a `short`. There are a lot of options. You probably even document that the `numberOfSeats` parameter should be an integral number.

Great!

So what happens if the client call still passes a float?

```
#include <iostream>

void foo(int numberOfSeats) {
  std::cout << "Number of seats: " << numberOfSeats << '\\
n';
  // ...
}

int main() {
  foo(5.6f);
}
/*
Number of seats: 5
*/
```

The floating-point parameter is accepted and narrowed down into an integer. You cannot even say that it's rounded, it's implicitly converted into an integer.

You might say that this is fine and in certain situations it probably is. But in others, this behaviour is simply not acceptable.

What can you do in such cases to avoid this problem?

Obviously, you can handle it on the caller side, but

- if `foo` is often used, it'd be tedious to do checks
- if `foo` is part of an API used by the external world, it's out of your control

Since C++11, we can use the delete specifier in order to restrict certain types for copying of moving, and in fact, even from polymorphic usage.

But `= delete` can be used for more. It can be applied to any function, should they be members or free.

If you don't want to allow implicit conversions from floating-point numbers, you can simply delete foo's overloaded version with a float:

```
1 #include <iostream>
2
3 void foo(int numberOfSeats) {
4   std::cout << "Number of seats: " << numberOfSeats << '\\
5 n';
6   // ...
7 }
8
9 void foo(double) = delete;
10
11 int main() {
12   foo(5.6f);
13 }
14
15 /*
16 main.cpp: In function 'int main()':
17 main.cpp:11:6: error: use of deleted function 'void foo(d\
18 ouble)'
19    11 |   foo(5.6f);
20       |   ~~~^~~~~
21 main.cpp:8:6: note: declared here
```

```
22      8 | void foo(double) = delete;
23        |      ^~~
24  */
```

That's it. By deleting some overloads of a function, you can forbid implicit conversions from certain types to the types you expect. Now, you are in complete control of what kind of parameters your users can pass to your API.

Reference:

- Three ways to use the = delete specifier in C++

Question 50: 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

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

Some examples:

```cpp
#include <iostream>
#include <type_traits>

class A {
 public:
  int m;
};

class B {
 public:
  B() {}
};

class C {
 public:
  C() = default;
};

class D : C {};

class E { virtual void foo() {} };

int main() {
  std::cout << std::boolalpha;
  std::cout << std::is_trivial<A>::value << '\n';
  std::cout << std::is_trivial<B>::value << '\n';
  std::cout << std::is_trivial<C>::value << '\n';
  std::cout << std::is_trivial<D>::value << '\n';
  std::cout << std::is_trivial<E>::value << '\n';
}
/*
true
false
true
true
false
*/
```

References:

- C++ Reference
- Microsoft Docs

Smart pointers

During the the next couple of questions, we'll focus on what smart pointers are, when and why should we use them.

But first, let's discuss a strongly related idiom.

Question 51: Explain the Resource acquisition is initialization (RAII) idiom

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.

But RAII is not just about acquiring resources, but also about releasing them. RAII also guarantees that all resources are released when the lifetime of their controlling object ends, in reverse order of acquisition. Similarly, when the resource acquisition of an object fails, all the resources that already have been successfully acquired by the object or by any of its members must be released in reverse order.

This is probably already enough to show what a bad name this idiom has, something the language creator Bjarne Stroustrup also regrets, a better name would probably be **Scope Bound Resource Management,** but most people still refer to it as RAII.

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.

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

Applying RAII by smart pointers, it should be ok:

```
1 std::unique_ptr<SomeLimitedResource> resource =
2     std::make_unique<SomeLimitedResource>();
3
4 // even if it throws an exception, the resource gets rele\
5 ased
6 resouce->doIt();
```

Of you can write your own RAII handler:

```
class SomeLimitedResourceHandler {
 public:
  SomeLimitedResourceHandler(SomeLimitedResource* resourc\
e) :
      m_resource(resource) {}

  ~SomeLimitedResourceHandler() { delete m_resource; }

  void doit() {
    m_resource->doit();
  }

 private:
  SomeLimitedResource* m_resource;
};


SomeLimitedResourceHandler resourceHandler(new SomeLimite\
dResource());
// resource will be released even if there is an exception
resourceHandler.doit();
```

References:

- C++ Reference: RAII
- Quora: What is Bjarne Stroustrup's biggest regret in the design history of C++?
- Wikipedia: Resource acquisition is initialization

Question 52: 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.

In C++11, one can create a unique poitner as such:

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

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

```
1 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:

```
1 T* t = new T();
2 std::unique_ptr<T> ptr (t);
3 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:

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

References:

- C++ Reference
- Converting unique_ptr to shared_ptr: Factory function example
- Effective Modern C++ by Scott Meyers

Question 53: 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:

```
1 std::shared_ptr<T> ptr (new T());
2 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_ptr`s 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.

References:

- C++ Reference
- Effective Modern C++ by Scott Meyers

Question 54: 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.

```
auto* sp = std::make_shared<T>();
std::weak_ptr<T> wp(sp);

//...
sp = nullptr;

if (wp.expired()) {
  std::cout << "wp doesn't point to a valid object anymor\
e" << '\n';
}
```

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);
```

So how it can be useful?

It can be useful for cyclic ownerships, to break the cycle. Let's say that you have an instance of a Keyboard, a Logger, and a Screen object. Both the Screen and the Keyboard has a shared pointer of the Logger and the Logger should have a pointer to the Screen.

Keyboard -> Logger <–> Screen

What pointer can it use? If we use a raw pointer when the Screen gets destroyed, the Logger will be still alive and the Keyboard still has shared ownership on it. The Logger has a dangling pointer to the Screen.

If it's a shared pointer, there is a cyclic dependency between them, they cannot be destroyed, there is a resource leak.

Here comes the `std::weak_ptr` to the rescue. There is still a dangling pointer from Logger to the Screen, when the Screen gets destroyed, but it can be easily detected as we have seen above.

Otherwise, it can be useful for caching and for the observer pattern. Check out the referenced book to get more details.

Reference:

- Effective Modern C++ by Scott Meyers: Item 20

Question 55: 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.

```
1 auto ptr = std::make_shared<T>(); // Prefer this
2 std::shared_ptr<T> ptr2(new T);
```

As you can see, with make functions, we have to type the type name only once, we don't have to duplicate it.

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.

References:

- Effective Modern C++ by Scott Meyers: Item 21
- Stackoverflow: Hot pas a deleter to make_shared

Question 56: 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.

At the same time, we should refrain from using the `new` keyword. `new` almost always means that you should `delete` somewhere. Unless you are creating a `std::unique_ptr` in C++11 where `std::make_unique` is not available.

To summarize, try not to use `new`. When it comes to ownership use smart pointers and the related factory functions - unless some special cases apply (check Effective Modern C++: Item 21). Otherwise, if you don't want to share or transfer ownership, just use a raw pointer. It'll keep the API more user-friendly and the run-time performance faster.

References:

- Core Guidelines: F7
- Effective Modern C++ by Scott Meyers (Item 21)

Question 57: When and why should we initialize pointers to nullptr?

If it's sure that the pointer will get initialized, it doesn't make sense to pre-initialize with a `nullptr`. Approaching from the other direction, you should initially assign `nullptr` to a variable in case it's not sure that you'd initialize it otherwise.

For example, the following piece of code might emit a warning based on your compiler and its settings. And hopefully, you treat warnings errors:

```
1  T* fun() {
2    T* t;
3    if (someCondition()) {
4      t = getT();
5    }
6    return t;
7  }
8  /*
9  Warnings by Clang:
10 prog.cc:22:6: warning: variable 't' is used uninitialized\
11  whenever 'if' condition is false [-Wsometimes-uninitiali\
12 zed]
13         if (someCondition()) {
14             ^~~~~~~~~~~~~~~
15 prog.cc:25:9: note: uninitialized use occurs here
16         return t;
17                ^
18 prog.cc:22:2: note: remove the 'if' if its condition is a\
19 lways true
20         if (someCondition()) {
21         ^~~~~~~~~~~~~~~~~~~~
22 prog.cc:21:6: note: initialize the variable 't' to silenc\
23 e this warning
24         T* t;
25             ^
26              = nullptr
27 */
```

But why is this a problem? What can go wrong?

When you don't initialize a pointer and then try to use it, you have 3 possible problems:

- The pointer might point at a memory location that you don't have access to, in such cases it causes a segmentation fault and your program crashes
- The pointer - accidentally - might point at some real data, and if you don't know what it's pointing at, you might cause unpredictable (and very hard to debug) changes to your data.
- You have no way to determine if the pointer has been initialized or not. How would you tell the difference between a valid address and the address that just happened to be there when you declared the pointer?

If you cannot initialize your pointer at declaration time, initializing it always to `nullptr` seriously decreases or eliminates those problems:

- While it's true that if we use the pointer it will still cause a segmentation fault, but at least we can reliably test if it's `nullptr` and act accordingly. If it's a random value, we don't know anything until it crashes.
- If we initialize it to `nullptr`, it will never point to any data, therefore it won't modify anything accidentally, only what it meant to.
- Hence, we can deterministically tell if a pointer is initialized or not and make decisions based on that.

```
1 T* fun() {
2   T* t = nullptr;
3   if (someCondition()) {
4     t = getT();
5   }
6   return t; // No warnings anymore!
7 }
```

It's a matter of style, but I'd personally prefer to avoid having to initialize a pointer as `nullptr`. In those cases, I'd prefer to return a `nullptr` right away and otherwise just initialize the pointer with the correct value at declaration time.

```
1 T* fun() {
2   if (!someCondition()) {
3     return nullptr;
4   }
5   return getT();
6 }
```

PS: It's even better to avoid the need of returning a `nullptr`.

Reference:

- Quora: Is the null pointer same as an uninitialized pointer?

References, universal references, a bit of a mixture

The next couple of questions will cover some mixture of references, universal references, and even `noexcept`.

Question 58: 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.

Hence, `std::move` is not a great name, probably `rvalue_cast` would be better, but it's called as it is. Let's just remember that it doesn't move anything. 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.

References:

- C++ Reference: std::move
- Effective Modern C++ by Scott Meyers: Item 21

Question 59: 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:

```
1 void process(MyClass&& rvalueArgument);
2
3 template<typename T>
4 void logAndProcess(T&& param) {
5   auto now = std::chrono::system_clock::now();
6   makeLogEntry("Calling 'process'", now);
7   process(std::forward(param));
8 }
```

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.

References:

- C++ Reference: std::forward
- Effective Modern C++ by Scott Meyers: Item 21

Question 60: 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.

```cpp
template<typename T>
void f(T&& param);

// universal reference
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.

```cpp
void f(MyClass&& param); // rvalue reference

MyClass&& var1 = MyClass(); // rvalue reference

template<typename T>
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? This understanding will also avoid ambiguities when you communicate with your colleagues ("I'm using a universal reference here, not an rvalue reference…")

References:

- ISOCpp.org
- Effective Modern C++ by Scott Meyers: Item 24

Question 61: What is reference collapsing?

Reference collapsing can happen in four different scenarios:

- template instantiation
- auto type generation
- creation and use of `typedef`s 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.

```
int x;
auto& & rx = x; // error! can't declare reference to refe\
rence

typedef int& T;
// a has the type int&

T&& a; // (&& + & => &)
template <typename T> void func(T&& a);
auto fp = func<int&&>; // && of func + && of int => &&
```

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.

References:

- IBM Knowledge Center: Reference collapsing (C++11)
- Effective Modern C++ by Scott Meyers: Item 28

Question 62: 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.

References:

- C++ Core Guidelines: If a function may have to be evaluated at compile-time, declare it constexpr
- Modernes C++: Programming at Compile Time with constexpr

Question 63: 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 from ModernesC++ to show how to use it:

```
1 void func1() noexcept; // does not throw
2 void func2() noexcept(true);  // does not throw
3 void func3() throw();  // does not throw
4 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.

With the next few questions, we'll discover the main features of C++20. Stay tuned!

References:

- C++ Core Guidelines: F6
- C++ Core Guidelines: E12
- ModernesC++: C++ Core Guidelines: The noexcept Specifier and Operator

C++20

The next couple of questions are about some basic knowledge of the latest C++ features from C++20. Knowing the answers proves that you - at least - try to keep up with the changes.

Question 64: What are concepts in C++?

Concepts are an extension to templates. They are compile-time predicates that you can use to express a generic algorithm's expectations on its template arguments.

Concepts allow you to formally document constraints on templates and have the compiler enforce them. As a bonus, you can also take advantage of that enforcement to improve the compile time of your program via concept-based overloading.

The main uses of concepts are:

- Introducing type-checking to template programming
- Simplified compiler diagnostics for failed template instantiations
- Selecting function template overloads and class template specializations based on type properties
- Constraining automatic type deduction

Here is how you can define concepts:

```
1 template<typename T>
2 concept integral = std::is_integral<T>::value;
```

Then you could use it as such:

```
1 auto add(integral auto a, integral auto b) {
2   return a+b;
3 }
```

What are the advantages of concepts?

- Requirements for templates are part of the interface.
- The overloading of functions or specialisation of class templates can be based on concepts.
- We get improved error messages because the compiler compares the requirements of the template parameter with the actual template arguments
- You can use predefined concepts or define your own.
- The usage of `auto` and concepts is unified. Instead of `auto`, you can use the `conceptName auto` syntax.
- If a function declaration uses a concept, it automatically becomes a function template. Writing function templates is, therefore, as easy as writing a function.

References:

- Modernes C++: Concepts, the details
- Modernes C++: Defining Concepts
- CppReference
- Microsoft C++ Blog
- Wikipedia

Question 65: What are the available standard attributes in C++?

First, what is an attribute? And how does it look like?

A simple one looks like this: `[[attribute]]`. But it can have parameters (`[[deprecated("because")]]`) or a namespace (`[[gnu::unused]]`) or both.

Attributes provide the unified standard syntax for implementation-defined language extensions, such as the GNU and IBM language extensions. They can be used almost anywhere in a C++ program, but our focus is not on them today.

We are interested in attributes defined by the C++ standard.

The first ones were introduced by C++11:

- `[[noreturn]]` indicates that a function does not return. It doesn't mean that it returns a void, but that it doesn't return. It can mean that it always throws an exception. Maybe different ones based on their input.
- `[[carries_dependency]]` indicates that a dependency chain in release-consume `std::memory_order` propagates in and out of the function, which allows the compiler to skip unnecessary memory fence instructions.

Then C++14 added another type of attribute with two versions:

- `[[deprecated]]` and `[[deprecated(reason)]]` indicate that the usage of that entity is discouraged. A reason can be specified as a parameter.

C++17 fastened up and added three more.

- `[[fallthrough]]` indicates in a `switch-case` that a `break` or `return` is missing on purpose. The fall through from one case label to another is intentional
- `[[nodiscard]]` indicates that the return value of a function should not be discarded - in other words, it must be saved into a variable - otherwise, you'll get a compiler warning
- `[[maybe_unused]]` suppresses compiler warnings on unused entities, if any. For example, you'll not get a compiler warning for an unused variable if it was declared with the `[[maybe_unused]]` variable.

C++20 added another 4 attributes:

- `[[nodiscard("reason")]]` it's the same as `[[nodiscard]]` but with a reason specified.
- `[[likely]]` indicates to the compiler that a `switch-case` or an `if-else` branch is likely to be executed more frequently than the others and as such it lets the compiler optimize for that evaluation path.

- `[[unlikely]]` has the same concept as `[[likely]]`, but in this case, such labelled paths are less likely to be executed than the others.
- `[[no_unique_address]]` indicates that this data member need not have an address distinct from all other non-static data members of its class.

Reference:

- C++ Reference

Question 66: What is 3-way comparison?

The three-way comparison operator is also known as the starship operator and it looks like this: `lhs <=> rhs`.

It helps to decide which operand is greater, lesser or whether they are equal.

If you ever implemented comparison operators in C++, you know what a menial, tedious task it is. You have to define six operators (==, `!=`, <, <=, >, >=).

With C++20, you can simply `=default` the spaceship operator and it will generate all the six `constexpr` and `noexcept` operators for you and they perform lexicographical comparison.

For the exact rules on the supported types, check CppReference.

References:

- ModernesC++
- CppReference

Question 67: Explain what consteval and constinit bring to C++?

C++11 introduced `constexpr` expressions that might be evaluated at compile-time.

C++20 introduces two new related keywords, `consteval` and `constinit`.

`consteval` can be used with functions:

```
1 consteval int sqr(int n) {
2   return n * n;
3 }
```

`consteval` functions are guaranteed to be executed at compile-time, thus they create compile-time constants. They cannot allocate or deallocate data, nor they can interact with `static` or thread-local variables.

`constinit` can be applied to variables with static storage duration or thread storage duration. So a local or member variable cannot be `constinit`. What it guarantees that the initialization of the variable happens at compile-time.

It must be noted that while `constexpr` and `const` variables cannot be changed once they are assigned, a `constinit` variable is not constant, its value can change.

References:

- CppReference: consteval specifier
- CppReference: constinit specifier
- Modernes C++

Question 68: What are modules and what advantages do they bring?

As we already discussed `#include` statements are basically textual inclusions. The preprocessor macro replaces the `#include` statement with the content of the file to be included.

Hence a simple hello world program can grow from around 100 bytes up to 13,000. Simply because of the inclusion of `<iostream>`.

All the header headers will be copied, even if you only want to use one small function.

Modules, introduced by C++20, finally bring a solution. Importing a module is basically free, unlike for inclusion, the order of imports doesn't matter.

With modules, you can easily structure your libraries and with `export` qualifiers you can easily decide what you want to expose and what not.

Thanks to the modules, there is no more need for separating header and implementation files.

Here is a short example:

```
1  // math.cppm
2  export module math;
3
4  export int square(int n){
5    return n*n;
6  }
7
8  // main.cpp
9  import math;
10
11 int main(){
12   square(42);
13 }
```

For more details - there are a lot! - check out the references.

During the next seven days, we'll learn about the special functions of C++ classes and the related rules we should follow.

References:

- CppReference
- Microsoft Devblogs
- ModernesC++

Special function and the rules of how many?

Question 69: Explain the rule of three

If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.

When you return or pass an object by value, you manipulate a container, etc., these member functions will be called. If they are not user-defined, they are generated by the compiler (since C++98).

Since C++98 the compiler tries to generate

- a default constructor (`T()`), that calls the default constructor of each class member and base class

-a copy constructor (`T(const T& other)`), that calls a copy constructor on each member and base class

- a copy assignment operator (`T& operator=(const T& other)`), that calls a copy assignment operator on each class member and base class
- the destructor (`~T()`), which calls the destructor of each class member and base class. Note that this default-generated destructor is never virtual (unless it is for a class inheriting from one that has a virtual destructor).

In case you have a class holding on a raw or smart pointer, a file descriptor, database connection, or other types managing resources, there is a fair chance that the generated functions would be incorrect and you should implement them by hand.

And here comes the rule of three, if you implement by hand either the copy constructor, the copy assignment operator or the destructor, the compiler will not generate those that you didn't write. So if you need to write any of these three, the assumption is that you'll need the rest as well.

This is the rule of three introduced by C++98 and tomorrow will move on to the rule of 5.

References:

- C++ Reference
- Fluent C++
- Modernes C++

Question 70: Explain the rule of five

Alright, yesterday we talked about the rule of three. Do you remember what it was about?

Let's recap.

If you implement by hand either the copy constructor, the copy assignment operator or the destructor, the compiler will not generate those that you didn't write. It's because the compiler assumes that if you'd have to implement one of them, it's almost for sure that you'd need the others too.

The rule of three was introduced by C++98, and the rule of five was introduced by C++11.

What's that extra two?

It's about move semantics that was introduced by C++11. So if you implement by hand any of the following special functions, then none of the others will be generated. You have to take care of implementing all of them:

- a copy constructor: `T(const T&)`
- a copy assignment: `operator=(const T&)`
- a move constructor: `T(X&&)`

- a move assignment: `operator=(T&&)`
- a destructor: `~T()`

Tomorrow will move on to the rule of zero.

References:

- C++ Reference
- Fluent C++
- Modernes C++

Question 71: Explain the rule of zero

The last two days, we talked about the rule of three introduced by C++98 and then about the rule of five which was complemented by the move semantics introduced by C++11.

Let's repeat the rule of five:

If you implement by hand any of the following special functions, then none of the others will be generated. You have to take care of implementing all of them:

- a copy constructor: `T(const T&)`
- a copy assignment: `operator=(const T&)`
- a move constructor: `T(X&&)`
- a move assignment: `operator=(T&&)`
- a destructor: `~T()`

Today we finish the mini-series of these rules with the short episode of the rule of zero.

It's the nickname of one of the rules defined by the C++ Core Guidelines:

> C.20: If you can avoid defining default operations, do

If all the members have all their special functions, you're done, you don't need to define any, zero.

```
class MyClass {
 public:
  // ... no default operations declared
 private:
  std::string name;
  std::map<int, int> rep;
};

MyClass mc; // default constructor
MyClass mc2 {nm};  // copy constructor
```

As both std::map and std::string have all the special functions, none is needed in `MyClass`

The idea is that a class needs to declare any of the special functions, then it should deal exclusively with ownership and other classes shouldn't declare these special functions.

So keep in mind, that if you need any of the special functions, implement all of them, but try not to need them in the first place.

References:

- C++ Reference
- Fluent C++
- Modernes C++

Question 72: 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.

Hence, `std::move` is not a great name, probably `rvalue_cast` would be better, but it's called as it is. Let's just remember that it doesn't move anything. 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: 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.

References:

- C++ Reference: std::move
- Effective Modern C++ by Scott Meyers: Item 21

Question 73: 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 name and is also prefixed with a tilde symbol (`MyClass`). 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.

It is generated by the compiler by default, 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
- assignment operator
- move constructor
- move assignment

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.

On the other hand, a destructor can be `virtual`, but that's the topic for tomorrow.

References:

- C++ Reference: destructor
- C++ Reference: RAII

Question 74: Should you explicitly delete unused/unsupported special functions or declare them as private?

The first question to answer is why would you do any of the options?

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.

```
1 class NonCopyable {
2  public:
3   NonCopyable() {/*...*/}
4   // ...
5
6  private:
7   NonCopyable(const NonCopyable&); //not defined
8   NonCopyable& operator=(const NonCopyable&); //not defin\
9 ed
10 };
```

Before C++11 there was no other option than declaring the unneeded special functions `private` and not implementing them. As such, one could disallow copying objects (there was no move back in time). The lack of implementation/definition helps against accidental usages in members,

friends, or when you disregard the access specifiers. You'll face a problem at linking time.

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

```
class NonCopyable {
 public:
  NonCopyable() {/*...*/}
  NonCopyable(const NonCopyable&) = delete;
  NonCopyable& operator=(const NonCopyable&) = delete;
  // ...
 private:
  // ...
};
```

The C++11 way is a better approach because

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

Deleted functions should be declared `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.

References:

- C++ Reference
- Effective Modern C++ by Scott Meyers

Question 75: What is a trivial class in C++?

When a class or struct in C++ has 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 default constructor, trivial copy and move constructor, trivial copy and move assignment operator and trivial destructor. In each case, trivial means the constructor/operator/destructor is not user-provided and belongs to a class that has

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

Some exaples:

```
#include <iostream>
#include <type_traits>

class A {
 public:
  int m;
};

class B {
 public:
  B() {}
};

class C {
 public:
  C() = default;
};

class D : C {};

class E {
  virtual void foo() {}
};

```

```cpp
int main() {
  std::cout << std::boolalpha;
  std::cout << std::is_trivial<A>::value << '\n';
  std::cout << std::is_trivial<B>::value << '\n';
  std::cout << std::is_trivial<C>::value << '\n';
  std::cout << std::is_trivial<D>::value << '\n';
  std::cout << std::is_trivial<E>::value << '\n';
}
/*
true
false
true
true
false
*/
```

References:

- C++ Reference
- Microsoft Docs

Question 76: What advantages does having a default constructor have?

We could say that a default constructor gives us the possibility of simple object creation, but it's not so much the case.

It's true that it's simple to create an object without passing any parameters, but it's only useful if the created object is fully usable. If it still has to be initialized, then the simple creation is worth nothing and in fact, it's even misleading and harmful.

On the other hand, many features of the standard library require having a default constructor.

Think about `std::vector`, when you create a vector of 10 elements (`std::vector<T> ts(10);`), 10 default constructed `T` objects will be added to the new vector.

Having a default constructor also helps to define how an object should look like that was just moved away from.

It’s worth noting that having a default constructor doesn’t mean that you should define it. Whenever possible, let the compiler generate it. So for example, if a default constructor only default initializes data members, then you are better off using in-class member initializers instead and let the compiler generate the default constructor.

So whenever possible, you should have a default constructor, because it lets you use more language and standard library features, but also make sure that a default constructor leaves a fully usable object behind.

In the next three weeks, we’ll discover some parts of object-oriented design, inheritance, how C++ handles polymorphism, the Curiously Recurring Template Pattern, etc. Stay tuned!

References:

- Core Guidelines: C43
- Core Guidelines: C45

Object oriented design, inheritance, polymorphism

During the next approximately 20 questions, we'll discover some parts of object-oriented design, inheritance, how C++ handles polymorphism, the Curiously Recurring Template Pattern, etc.

Question 77: What are the differences between a class and a struct?

In C++, there is very little difference between classes and structs. Specifically, the default accessibility of member variables and methods are public in a struct and private in a class. Nothing more. In practice, many programmers use the struct type as a storage class, only. This is perhaps a throwback to C, where structures did not support functions (or methods).

On the contrary, in C++ structs can have both functions, constructors, even virtual functions. They can use inheritance, they can be templated, exactly like a class.

When we speak about default visibility, it's worth mentioning that it applies to inheritance as well. When you write `class Derived : Base ...` you get private inheritance, on the other hand, when you write `struct Derived: Base ...` what you have is public inheritance.

As we see, there is not much technical difference between the two, but on the other hand according to the established conventions what we express with one and the other is quite different.

A struct is a bundle of related elements, usually without any logic encapsulated. It's only there to effectively group things, to raise the abstraction.

At the same time, a class usually does things. It encapsulates data and the related logic. It offers an interface, at a level, it separates the interface from the implementation. As a general rule of thumb, if you have at least one private member, it's better to use a class.

There are the following Core guidelines discussing this topic more in detail:

- C.1: Organize related data into structures (structs or classes)
- C.2: Use class if the class has an invariant; use struct if the data members can vary independently
- C.3: Represent the distinction between an interface and an implementation using a class
- C.8: Use class rather than struct if any member is non-public

Question 78: What is constructor delegation?

Constructor delegation means that one constructor can call another one from the same class. This is something that is possible in C++ since C++11.

Constructor delegation helps to simplify the code by removing code duplication for class initialization.

```
1 class T {
2  public:
3   T() : T(0, ""){}
4   T(int iNum, std::string iText) : num(iNum), text(iText)\
5  {};
6  private:
7   int num;
8   std::string text;
9 };
```

In the above example, we can see that the default constructor simply calls the constructor that takes 2 arguments. This becomes practical when you have more members and you have to add one more. You have to update the constructor that directly initializes all the members, and the compiler will gently remind you of all the places where you have to pass in the extra variable.

Question 79: Explain the concept of covariant return types and show a use-case where it comes in handy!

Using covariant return types for a virtual function and for all its overridden versions means that you can replace the original return type with something narrower, in other words, with something more specialized.

Let's say you have a CarFactoryLine producing Cars. The specialization of these factory lines might produce SUVs, SportsCars, etc. How do you represent it in code? The obvious way is still having the return type as a Car pointer.

```
class CarFactoryLine {
 public:
  virtual Car* produce() {
    return new Car{};
  }
};

class SUVFactoryLine : public CarFactoryLine {
 public:
  virtual Car* produce() override {
    return new SUV{};
  }
};
```

This way, getting an SUV* requires a dynamic cast.

```
SUVFactoryLine sf;
Car* car = sf.produce();
SUV* suv = dynamic_cast<SUV*>(car);
```

Instead, we directly return an SUV*

```
class Car {
 public:
  virtual ~Car() = default;
};

class SUV : public Car {};

class CarFactoryLine {
 public:
```

```
10   virtual Car* produce() {
11     return new Car{};
12   }
13 };
14
15 class SUVFactoryLine : public CarFactoryLine {
16  public:
17   virtual SUV* produce() override {
18     return new SUV{};
19   }
20 };
```

So that you can simple do this:

```
1 SUVFactoryLine sf;
2 SUV* car = sf.produce();
```

In C++, in a derived class, in an overridden function, you don't have to return the same type as in the base class, but you can return a covariant return type. In other words, you can replace the original type with a "narrower" one, in other words, with a more specified data type.

Question 80: What is the difference between function overloading and function overriding?

Function overriding is a term related to polymorphism. If you declare a virtual function in a base class with a certain implementation, in a derived class you can override its behaviour in case it uses the very same signature. In case, the `virtual` keyword is missing in the base class, it's still possible to create a function with the same signature in the derived class, but it doesn't override it. Since C++11 there is also the override specifier which helps you to make sure that you didn't mistakenly fail to override a base class function. You can find more details here.

Function overriding enables you to provide specific implementation of the function that is already defined in its base class.

On the other hand, overloading has nothing to do with polymorphism. When you have two functions with the same name, same return type, but different

numbers or types of parameters, or qualifiers.

Here is an example for overloading based on parameters

```
void myFunction(int a);
void myFunction(double a);
void myFunction(int a, int b);
```

And here is another example based on qualifiers:

```
class MyClass {
 public:
  void doSomething() const;
  void doSomething();
};
```

Or actually, it is possible to overload your function based on whether their hosting objects are rvalue of lvalue references.

```
class MyClass {
 public:
  // ...

  void doSomething() &; // used when *this is a lvalue
  void doSomething() &&; // used when *this is a rvalue
};
```

You can learn more about this here.

Question 81: What is the `override` keyword and what are its advantages?

The `override` specifier will tell both the compiler and the reader that the function where it is used is actually overriding a method from its base class.

It tells the reader that "this is a virtual method, that is overriding a virtual method of the base class."

Use it correctly and you see no effect:

```
class Base {
  virtual void foo();
};
```

```
4
5 class Derived : Base {
6   void foo() override; // OK: Derived::foo overrides Base\
7 ::foo
8 };
```

But it will help you revealing problems with constness:

```
1 class Base {
2   virtual void foo();
3   void bar();
4 };
5
6 class Derived : Base {
7   void foo() const override; // Error: Derived::foo does \
8 not override Base::foo
9                              // It tries to override Base\
10 ::foo const that doesn't exist
11 };
```

Let's not forget that in C++, methods are non-virtual by default. If we use `override`, we might find that there is nothing to override. Without the `override` specifier we would just simply create a brand new method. No more base methods are forgotten to be declared as virtual if you use consistently the `override` specifier.

```
1 class Base {
2   void foo();
3 };
4
5 class Derived : Base {
6   void foo() override; // Error: Base::foo is not virtual
7 };
```

We should also keep in mind that when we override a method - with or without the `override` specifier - no conversions are possible:

```
1 class Base {
2  public:
3   virtual long foo(long x) = 0;
4 };
5
6 class Derived: public Base {
7  public:
8   // error: 'long int Derived::foo(int)' marked override,\
9  but does not override
10   long foo(int x) override {
11     // ...
```

```
12   }
13 };
```

In my opinion, using the override specifier from C++11 is part of clean coding principles. It reveals the author's intentions, it makes the code more readable and helps to identify bugs at build time. Use it without moderation!

Question 82: Explain what is a friend class or a friend function

Sometimes, there is a need for allowing a particular class to access private or protected members of our class. Loosen the access level of that member globally would be way too much.

Unless you can tweak the design to remove this need, the solution is a friend class, which is capable of accessing the protected as well as the private members of the class in which it is declared as a friend.

```
1  #include <iostream>
2
3  class A {
4   public:
5    void foo() {
6      std::cout << "foo\n";
7    }
8
9   private:
10   void bar() {
11     std::cout << "bar\n";
12   }
13
14   friend class B;
15 };
16
17 class B {
18  public:
19   void doIt() {
20    A a;
21    a.foo();
22    a.bar(); // B is a friend class of A, so it has access\
23  to A::b
24   }
25 }
26
27 int main() {
28   A a;
29   a.foo();
```

```
  // a.bar();
  // this would fail to compile as A::bar is private
  B b;
  b.doIt()
}
```

Similarly to the friend class, a friend function is able to access private and protected class members. A friend function can either be a free function or a method of a class.

```
#include <iostream>

class A {
 public:
  void foo() {
    std::cout << "foo\n";
  }

 private:
  void bar() {
    std::cout << "bar\n";
  }

  friend void freeFunction();
};

void freeFunction() {
  A a;
  a.foo();
  a.bar(); // freeFunction is a friend function of A, so \
it has access to freeFunction
}

int main() {
  A a;
  a.foo();
  // a.bar(); // this would fail to compiler as A::bar is\
 private
  freeFunction();
}
```

Some important points about friend classes and friend functions:

- Friendship is not inherited
- Friendship isn't mutual i.e. if some class called `Friend` is a friend of some other class called `NotAFriend` then `NotAFriend` doesn't automatically become a friend of the `Friend` class
- The total number of friend classes and friend functions should be limited in a program as the overabundance of the same might lead to a

depreciation of the concept of encapsulation of separate classes, which is an inherent and desirable quality of object-oriented programming
- A common use-case for friend classes and functions is unit-testing. The test class becomes a friend of the class under test

Question 83: What are default arguments? How are they evaluated in a C++ function?

A default parameter is a value that is assigned to a parameter while declaring a function.

A default argument allows a function to be called without providing one or more trailing arguments.

The default value for a parameter is indicated at function declaration time at the parameter list. We simply assign a value to the parameter in the function declaration.

Here is an example:

```
int calculateArea(int a, int b);
int calculateArea(int a, int b=2);
int calculateArea(int a=3, int b=2);
```

These default values are used if one or more arguments are left blank while calling the function - according to the number of left blank arguments.

Let's see a complete example. If the value is not passed for any of the parameters during the function call, then the compiler uses the default value(s) provided. If a value is specified, then the default value is stepped on and the passed value is used.

```
#include <iostream>

int calculateArea(int a=3, int b=2) {
  return a*b;
}

int main() {
  std::cout << calculateArea(6, 4) << '\n';
  std::cout << calculateArea(6) << '\n';
```

```
10   std::cout << calculateArea() << '\n';
11 }
12 /*
13 24
14 12
15 6
16 */
```

As shown in the above code, there are three calls to `calculateArea` function. In the first call, we pass both arguments so the default values are overridden by the caller, in the second only one, so the last parameter defaults. In the last call, we provide no argument, so both parameters take their default values.

In a function declaration, after a parameter with a default argument, all subsequent parameters must have a default argument supplied in this or a previous declaration from the same scope…unless the parameter was expanded from a parameter pack, and keep in mind that ellipsis(. . .) is not a parameter.

This means that that the `int calculateArea(int a=5, int b);` would not compile. On the other hand, `int g(int n = 0, ...)` would compile as ellipsis does not count as a parameter.

Default arguments are only allowed in the parameter lists of function declarations and lambda-expressions, (since C++14) and are not allowed in the declarations of pointers to functions, references to functions, or in typedef declarations.

One last thing to note is that you should always declare your defaults in the header, not in the cpp file.

Question 84: What is `this` pointer and can we delete it?

The `this` pointer is a constant pointer that holds the memory address of the current object.

Whenever we call a member function through its object, the compiler secretly passes the address of the calling object as the first parameter. Therefore `this` is available as a local variable within the body of all non-static functions. The `this` pointer is not available in static member functions as static member functions can be called without any object (only with the class name such as `MyClass::foo()`).

Ideally, the delete operator should not be used for `this` pointer. However, if it is used, then you must take into account the following points.

- `delete` operator works only for objects allocated on the heap (using operator `new`), if it's allocated on the stack the behaviour is undefined, your application might crash.

```
class A {
 public:
  void fun() {
    delete this;
  }
};

int main() {
  A *aPtr = new A;
  aPtr->fun(); // a valid call

  // make ptr NULL to make sure that things are not acces\
sed using ptr.
  aPtr = nullptr;

  A a;
  a.fun(); // undefined behaviour
}
```

- Once `delete this` is done, any member of the deleted object should not be accessed after deletion.

```
#include <iostream>

class A {
 public:
  void fun() {
    delete this;
    // undefined behaviour, your application might crash
    std::cout << x << '\n';
  }
 private:
  int x{0};
```

```
12 };
13
14 int main() {
15   A a;
16   a.fun();
17 }
```

The best thing is to not do delete `this` at all because it is easy to accidentally access member variables after the deletion. Besides, the caller might not realize your object has self-destructed.

Also, `delete this` is a "code smell" indicating that your code might not have an asymmetric strategy for object ownership (who allocates and who deletes).

When still used, It is usually found in reference-counted classes that, when the ref-count is decremented to 0, the DecrementRefCount()/Release()/whatever member function calls `delete this`.

Question 85: What is virtual inheritance in C++ and when should you use it?

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.

As you probably guessed, this technique is useful when you have to deal with multiple inheritance and you can solve the infamous diamond inheritance.

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 in the bottom) have very little if any data.

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

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

struct Student: Person {
  virtual void learn() {}
};

struct Worker: Person {
  virtual void work() {}
};

// A teaching assistant is both a worker and a student
struct TeachingAssistant: Student, Worker {};
TeachingAssistant ta;
```

As declared above, 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:

```
TeachingAssistant ta;
Person& a = ta;  // error: which Person subobject should \
a TeachingAssistant cast into,
                 // a Student::Person or a Worker::Person?
```

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

```
TeachingAssistant ta;
Person& student = static_cast(ta);
Person& worker = static_cast(ta);
```

In order to call `speak()`, the same disambiguation, or explicit qualification is needed: `static_cast<student&>(ta).speak()</student&>` or `static_cast<worker&>(ta).speak()</worker&>` or alternatively

`ta.Student::speak()` and `ta.Worker::speak()`. Explicit qualification not only uses an easier, uniform syntax for both pointers and objects but also allows for static dispatch, so it would arguably be the preferable method.

In this case, the double inheritance of `Person` is probably unwanted, as we want to model that the relation (`TeachingAssistant` is a `Person`) exists only once; that a `TeachingAssistant` is a `Student` and is a `Worker` does not imply that it is a `Person` twice (unless the `TA` is schizophrenic): a `Person` base class corresponds to a contract that `TeachingAssistant` implements (the "is a" relationship above really means "implements the requirements of"), and a `TeachingAssistant` only implements the `Person` contract once.

The real-world meaning of "only once" is that `TeachingAssistant` should have only one way of implementing `speak`, not two different ways, depending on whether the `Student` view of the `TeachingAssistant` is eating, or the `Worker` view of the `TeachingAssistant`. (In the first code example we see that `speak()` is not overridden in either `Student` or `Worker`, so the two `Person` subobjects will actually behave the same, but this is just a degenerate case, and that does not make a difference from the C++ point of view.)

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

```
1  struct Person {
2    virtual ~Person() = default;
3    virtual void speak() {}
4  };
5
6  // Two classes virtually inheriting Person:
7  struct Student: virtual Person {
8    virtual void learn() {}
9  };
10
11 struct Worker: virtual Person {
12   virtual void work() {}
13 };
14
15 // A teaching assistant is still a student and the worker
16 struct TeachingAssistant: Student, Worker {};
```

Now we can easily call `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.

You can find more detail at the Core Guidelines\nAddition reference here.

Question 86: 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.

So let's see the main drawbacks.

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 and forces you to make some additional

revisions in all those classes during refactoring. All this leads to new bugs and makes the code less readable.

Troubles with type conversions may also be a source of bugs. You can partly solve the issues by using the expensive `dynamic_cast` wherever you used to use `static_cast`. Unfortunately, `dynamic_cast` is much slower, and if you have to use it too often in your code, it means that your project's architecture has quite some room for improvement.

Question 87: What does a strong type mean and what advantages does it have?

A strong type carries extra information, a specific meaning through its name. While you can use booleans or strings everywhere, the only way they carry can carry meaning is the name of their instances.

The main advantages of strong typing are readability and safety.

If you look at this function signature, perhaps you think it's alright:

```
1 Car::Car(unit32_t horsepower, unit32_t numberOfDoors,
2          bool isAutomatic, bool isElectric);
```

It has relatively good names, so what is the issue?

Let's look at a possible instantiation.

```
1 auto myCar{Car(96, 4, false, true)};
```

Yeah, what? God knows… And you if you take your time to actually look up the constructor and do the mind mapping. Some IDEs can help you visualizing parameter names, like if they were Python-style named parameters, but you shouldn't rely on that.

Of course, you could name the variables as such:

```
1 constexpr unit32_t horsepower = 96;
2 constexpr unit32_t numberOfDoors = 4;
3 constexpr bool isAutomatic = false;
```

```
4 constexpr bool isElectric = false;
5
6 auto myCar = Car{horsepower, numberOfDoors,
7                  isAutomatic, isElectric};
```

Now you understand right away which variable represents what. You have to look a few lines upper to actually get the values, but everything is in sight. On the other hand, this requires willpower. Discipline. You cannot enforce it. Well, you can be a thorough code reviewer, but you won't catch every case and anyway, you won't be there all the time.

Strong typing is there to help you!

Imagine the signature as such:

```
1 Car::Car(Horsepower hp, DoorsNumber numberOfDoors,
2          Transmission transmission, Fuel fuel);
```

Now the previous instantiation could look like this:

```
1 auto myCar{Car{Horsepower{98u}}, DoorsNumber{4u},
2            Transmission::Automatic, Fuel::Gasoline};\n
```

This version is longer and more verbose than the original version - which was quite unreadable -, but much shorter than the one where introduced well-named helpers for each parameter

So one advantage of strong typing is readability and one other is safety. It's much harder to mix up values. In the previous examples, you could have easily mixed up door numbers with performance, but by using strong typing, that would actually lead to a compilation error.

References:

- Fluent Cpp
- SandorDargo's Blog
- Correct by Construction: APIs That Are Easy to Use and Hard to Misuse - Matt Godbolt

Question 88: 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.

They can be used for conversions, you can write a degrees-to-radian converter:

```
constexpr long double operator"" _deg ( long double deg )\
 {
  return deg * 3.14159265358979323846264L / 180;
}

//...
std::cout << "50 degrees as radians: " << 50.0_deg << '\n\
';
// 50 degrees as radians: 0.872665\n
```

But what is probably more interesting is how it can help readability and safety when you use strong types.

Let's take this example from yesterday's lesson:

```
auto myCar{Car{Horsepower{98u}}, DoorsNumber{4u},
           Transmission::Automatic, Fuel::Gasoline};
```

It's not very probable that you'd mix up the number of doors with the power of the engine, but there can be other systems to measure performance or other numbers taken by the Car constructor.

A very readable and safe way to initialize Car objects with hardcoded values (so far example in unit tests) is via user-defined literals:

```
//...
Horsepower operator"" _hp(int performance) {
  return Horsepower{performance};
}

DoorsNumber operator"" _doors(int numberOfDoors) {
  return DoorsNumber{numberOfDoors};
}

//...
```

```
auto myCar{Car{98_hp, 4_doors,
               Transmission::Automatic, Fuel::Gasoline};
```

Of course, the number of ways to use user-defined literals is limited only by your imagination, but numerical conversions, helping strong typing are definitely two of the most interesting ones.

References:

- C++ Reference
- Modernes C++

Question 89: Why shouldn't we use boolean arguments?

Because it reduces readability on the client-side.

Let's say you have such a function signature:

```
placeOrder(std::string shareName, bool buy,
           int quantity, double price);
```

In the implementation of this function, it will be all fine, you might have something like this somewhere in it:

```
//...
if (buy) {
  // do buy
  // ...
} else {
  // do sell
  // ...
}
```

That's maybe not ideal, but it is readable.

On the other hand, think about the client-side:

```
placeOrder("GOOG", true, 1000, 100.0);
```

What the hell `true` means? What if we send `false`? You don't know without jumping to the signature. While modern IDEs can help you with tooltips showing the function signature even the documentation - if available -, we cannot take that for granted, especially for C++ where this kind of tooling is still not at the level of other popular languages.

And even if you look it up, it's so easy to overlook and mix up true with false.

Instead, let's use enumerations:

```
1 enum class BuyOrSell {
2   Buy, Sell
3 };
4
5 placeOrder("GOOG", BuyOrSell::Buy, 1000, 100.0);
```

Using enumerations instead of booleans slightly increases the number of lines in our codebase (well in this case with three lines), but it has a much bigger positive effect on readability. Both on the API side and on the client-side, the intentions become super clear with this technique.

Reference:

- Matt Godbolt: Correct by Construction

Question 90: Distinguish between shallow copy and deep copy

A shallow copy does memory dumping bit-by-bit from one object to another, in other words, it copies all of the member field values. A deep copy is a field by field copy from one object to another.

The big difference is that a shallow copy may not do what you want for members that are not storing values, but pointers to values. In other words, for members pointing to dynamically allocated memory. A shallow copy of those means that no new memory will be allocated and in fact, if you modify

such a member through one object, the changes will be visible from all the copies.

The default copy constructor and assignment operator make shallow copies. In case, a class has a reference or pointer members, there is a fair chance that you have written the copy constructor yourself and you should also write the assignment operator.

If you have to do so, don't forget about the rule of five. Implementing one of the special functions (such as the just mentioned copy constructor or the assignment operator) will likely make you implement all the other special functions.

To sum up, default copy constructor and default assignment operators do memberwise shallow copies, which is fine for classes that contain no dynamically allocated variables. On the other hand, classes with dynamically allocated variables need to have a copy constructor and assignment operator that do a deep copy.

References:

- Fluent C++
- Fredosaurus.com
- LearnCpp
- Tutorials Point

Question 91: Are class functions taken into consideration as part of the object size?

No, only the class member variables determine the size of the respective class object. The size of an object is at least the sum of the sizes of its data members. It will never be smaller than this.

How can it be more?

The compiler might insert padding between data members to ensure that each data member meets the alignment requirements of the platform. Some platforms are very strict about alignment, more than the others, yet in general, code with proper alignment will perform significantly better. Therefore the optimization setting might affect the object size.

Static members are not part of the class object and as such, they are not included in the object's layout, they don't add up to the object's size.

Virtual functions and inheritance give some complexity to the size calculation, we'll get back to it another day.

`sizeof(myObj)` or `sizeof(MyClass)` will always tell you the proper size of an object.

Question 92: What does dynamic dispatch mean?

In computer science, dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time. It is commonly employed in, and considered a prime characteristic of, object-oriented programming (OOP) languages and systems.

Dynamic dispatch contrasts with static dispatch, in which the implementation of a polymorphic operation is selected at compile time. The purpose of

dynamic dispatch is to defer the selection of an appropriate implementation until the run time type of a parameter (or multiple parameters) is known.

Dynamic dispatch is different from late or dynamic binding. A polymorphic operation has several implementations, all associated with the same name. Bindings can be made at compile time or (with late binding) at run time. With dynamic dispatch, one particular implementation of an operation is chosen at run time. While dynamic dispatch does not imply late binding, late binding does imply dynamic dispatch, since the implementation of a late-bound operation is not known until run time.

In practice, dynamic dispatch usually means the action of finding the right function to call. In the general case, when you define a method inside a class, the compiler will remember its definition and execute it every time a call to that method is encountered.

```
#include <iostream>

class A {
 public:
  virtual void foo() {
    std::cout << "This is A::foo()" << '\n';
  }
};

class B : public A {
 public:
  void foo() override {
    std::cout << "This is B::foo()" << '\n';
  }
}

int main() {
  A* a = new B();
  a->foo();
}
```

If we used static dispatch the call `a->foo()` would execute `A::foo()`, since (from the point of view of the compiler) `a` points to an object of type `A`. This would be wrong because `a` actually points to an object of type `B` and B::bar() should be called instead.

It’s dynamic dispatching that allows us to have this behaviour.

References:

- Pablo Arias' blog
- Wikipedia

Question 93: What are vtable and vpointer?

Before we continue on discovering how object sizes are calculated we should understand what `vtable`and`vpointer` are.

Virtual tables and pointers are means to implement dynamic dispatch. For every class that contains virtual functions, the compiler constructs a virtual table (`vtable`) serving as a lookup table of functions used to resolve function calls in a dynamic binding manner. The `vtable` contains an entry for each virtual function accessible by the class and stores a pointer to its definition. Only the most specific function definition callable by the class is stored in the `vtable`. Entries in the vtable can point either to virtual functions declared in the class itself or to virtual functions inherited from a base class.

A `vtable` containing function pointers are maintained on a class level, i.e, there is one `vtable` defined per class.

Every time the compiler creates a `vtable` for a class, it adds an extra argument to it: a pointer to the corresponding virtual table called the pointer. The pointer is just another class member added by the compiler. It increases the size of every object - where the class has a vtable - by the size of vpointer. This means that while there is one `vtable` per class, there is one `vpointer` per object.

When a virtual function is called, the `vpointer` of the object is used to find the corresponding `vtable` of the class. Next, the function name is used as an index to the `vtable` to find the most specific function to be executed.

References:

- Pablo Arias' blog
- LearnCpp

Question 94: Should base class destructors be virtual?

In the last couple of days, we've been talking about polymorphism, inheritance and whatever is implied. It's not a big surprise that we are going to discuss destructors once again.

While the usual answer is "yes, of course", it's not the correct answer. If you just think about the standard library itself, many classes don't have a virtual destructor. That's something I wrote about in strongly-types containers. So if for example `std::vector` doesn't have a virtual destructor, the answer "of course", cannot be correct.

According to Herb Sutter, "a base class destructor should be either public and virtual or protected and nonvirtual."

Any operation that will be performed through the base class interface, and that should behave virtually, should be virtual. If deletion, therefore, can be performed polymorphically through the base class interface, then it must behave virtually and must be virtual. The language requires it - if you delete polymorphically without a virtual destructor, you have to face something that is called undefined behaviour. Something that we always want to avoid.

```
1 class Base { /*...*/ };
2
3 class Derived : public Base { /*...*/ };
4
5 int main() {
6   Base* b = new Derived;
7   delete b; // Base::~Base() had better be virtual!
8 }
```

But base classes need not always allow polymorphic deletion. In such cases, make base class destructor protected and non-virtual. It'll make deletes through base class pointers fail:

```
1 class Base {
2   /*...*/
3  protected:
4   ~Base() {}
5 };
```

```
6
7 class Derived : public Base { /*...*/ };
8
9 int main() {
10   Base* b = new Derived;
11   delete b; // error, illegal
12 }
13 /*
14 main.cpp: In function 'int main()':
15 main.cpp:11:10: error: 'Base::~Base()' is protected withi\
16 n this context
17    11 |   delete b; // error, illegal
18       |          ^
19 main.cpp:4:3: note: declared protected here
20     4 |   ~Base() {}
21       |   ^
22 */
```

References:

- GotW.ca: Herb Sutter
- Sandor Dargo's Blog

Question 95: What is an abstract class in C++?

An abstract class in C++ is a class with at least one pure virtual function. A pure virtual is a function that has no implementation in that given class.

```
1 class AbstractClass {
2  public:
3   virtual void doIt() = 0;
4 };
```

Such classes cannot be instantiated. If you tried to declare a variable a of AbstractClass, you'd receive an error message similar to "cannot declare variable 'a' to be of abstract type 'AbstractClass' because the following virtual functions are pure within 'AbstractClass': virtual void AbstractClass::doIt()"

Any class that is meant to be instantiated and inherits from an abstract one, it must implement all the pure virtual - in other words, abstract - functions. Though non-leaf classes inheriting from an abstract class that are only used

as base classes of other classes, they don't have to implement the pure virtual functions:

```
class AbstractClass {
 public:
  virtual void doIt() = 0;
};

class StillAbstractClass : public AbstractClass {
 public:
  // no need to implement doIt, we can even add more pure\
 virtuals
  virtual void foo() = 0;
};

class Leaf : public StillAbstractClass {
  void doIt() override { /* ... */ }
  void foo() override {/* ... */ }
};
```

Though we cannot instantiate an abstract class, we can have pointers or references to it:

```
int main() {
  StillAbstractClass* s = new Leaf();
}
```

A class is abstract if it has at least one pure virtual function. So an abstract class can have non-virtual or just non-pure virtual functions. It can even have a constructor and some members.

Question 96: Is it possible to have polymorphic behaviour without the cost of virtual functions?

The short answer is yes.

How?

That's an exciting question! By inheriting from a class that takes the inheriting class as a template parameter. It's called the "Curiously Recurring Template Pattern" or "Upside Down Inheritance". When in Microsoft Christian

Beaumont saw it in a code review, he thought it cannot possibly compile, but it worked and it became widely used in some Windows libraries.

Its general form is:

```
// The Curiously Recurring Template Pattern (CRTP)
template <class T>
class Base {
  // methods within Base can use template to access membe\
rs of Derived
};

class Derived : public Base<Derived> {
  // ...};\n
```

The purpose of doing this is to use the derived class in the base class. From the perspective of the base object, the derived object is itself but downcasted. Therefore the base class can access the derived class by `static_cast`ing itself into the derived class.

```
template <typename T>
class Base {
 public:
  void doSomething() {
    T& derived = static_cast<T&>(*this);
    //use derived...
  }
};
```

Note that contrary to typical casts to the derived class, we don't use `dynamic_cast` here. A `dynamic_cast` is used when you want to make sure at run-time that the derived class you are casting into is the correct one. But here we don't need this guarantee: the `Base` class is designed to be inherited from by its template parameter, and by nothing else. Therefore it takes this as an assumption, and a `static_cast` is enough.

References:

- Fluent C++
- Wikipedia

Question 97: How would you add functionality to your classes with the Curiously Recurring Template Pattern (CRTP)?

The CRTP consists of:

- Inheriting from a template class
- Use the derived class itself as a template parameter of the base class

As mentioned yesterday, the main advantage of this technique is that the base class have access to the derived class methods. Why is that?

The base class uses the derived class as a template parameter.

In the base class, we can get the underlying Derived object with a static cast:

```
1 class Base {
2   void foo() {
3     X& underlying = static_cast<X&>(*this);
4     // now you can access X's public interface
5   }
6 };
```

In practice, this brings us the possibility of enriching our Derived class' interface through some base classes. In practice, you'd use this technique to add some general functionalities to a class such as some mathematical functions to a sensor class (such as explained by Johnathan Baccara). Although these functionalities can be implemented as non-member functions or non-member template functions, those are hard to know about when you check the interface of a class. Whereas the public methods of a CRTP's base class are part of the interface.

Here is the full example:

```
1 template <typename T>
2 struct NumericalFunctions {
3   void scale(double multiplicator);
4   void square();
5   void setToOpposite();
```

```
};

class Sensitivity : public NumericalFunctions<Sensitivity\
> {
 public:
  double getValue() const;
  void setValue(double value);

  // rest of the sensitivity's rich interface...
};

template <typename T>
struct NumericalFunctions {
  void scale(double multiplicator) {
    T& underlying = static_cast<T&>(*this);
    underlying.setValue(underlying.getValue() * multiplic\
ator);
  }

  void square() {
    T& underlying = static_cast<T&>(*this);
    underlying.setValue(underlying.getValue() * underlyin\
g.getValue());
  }

  void setToOpposite() {
    scale(-1);
  }
};
```

You can check the below links for additional practical uses.

Resources:

- Fluent C++
- Sandor Dargo's blog

Question 98: What are the good reasons to use init() functions to initialize an object?

It's a short list. There is none.

When an object is created, the user can rightly expect to have a fully usable, fully initialized object. If the user still has to call `init()`, that's not the case.

On the contrary, no `destroy()`, `close()` or alike function should be called by the user when he stops using the object. An object should take care of releasing whatever resources it acquired during its initialization. This concept is also known as RAII (Resource Acquisition Is Initialization).

In case, an object really cannot be instantiated through a constructor in a convenient way, then the construction and initialization should be encapsulated by a factory function so that the user still doesn't have to take care of this process.

To conclude, don't use `init()` functions. An object should take care of itself through its constructor and destructor. If the construction is not possible like that, encapsulate it by a factory function.

References:

- Core Guidelines: C41
- Core Guidelines: F50

Observable behaviours

In the next couple of questions, we'll discuss the different observable behaviours of a C++ program, such as unspecified and undefined behaviour, etc.

Question 99: What is observable behaviour of code?

The term observable behavior, according to the standard, means the following:

> — Accesses (reads and writes) to volatile objects occur strictly according to the semantics of the expressions in which they occur. In particular, they are not reordered with respect to other volatile accesses on the same thread.
> — At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.
> — The input and output dynamics of interactive devices shall take place in such a fashion that prompting output is actually delivered before a program waits for input. What constitutes an interactive device is implementation-defined.

The "as-if rule" is strongly related, in short, any code transformation is allowed that does not change the observable behavior of the program.

The C++ standard precisely defines the observable behavior of every C++ program that does not fall into one of the following classes:

- ill-formed
- ill-formed, no diagnostic required
- implementation-defined behaviour

- unspecified behaviour
- undefined behaviour

References:

- C++ Reference: The as-if rule
- C++ Reference: Undefined behaviour
- Stackoverflow: “Observable behaviour” and compiler freedom to eliminate/transform pieces c++ code

Question 100: What are the characteristics of an ill-formed C++ program?

Yesterday we saw that the C++ standard defines the behaviour of C++ programs. In case the program is not well-formed, the behaviour falls into one of the five other cases. One of them is “ill-formed” and another is “ill-formed, no diagnostic required”

ill-formed

In this case, the program has syntax errors and/or diagnosable semantic errors. The compiler will tell you about them. The violated rules are written in the standard with either shall, shall not or ill-formed.

ill-formed, no diagnostic required

Under this category, there will be no compiler errors. The program doesn’t have syntactic errors, only semantic ones, but in general, they are not diagnosable by the compiler.

These semantic errors are either detectable at link time, or if the program is executed, it results in undefined behaviour.

Such problems are violations against the One Definition Rule which says that only one definition of any variable, function, class type, enumeration type, concept (since C++20) or template is allowed in any one translation unit.

There might be multiple declarations, but only one definition. Check out the references for more details on ODR.

References:

- C++ Reference: ODR
- C++ Reference: Undefined Behaviour

Question 101: What is unspecified behaviour?

The behaviour of a program is said to be unspecified when the standard does not specify what should happen or it specifies multiple options. The exact behaviour depends on the implementation and may not be completely determined upon examination of the program's source code.

Even with the implementation, the compiler doesn't have to document how it resolves such situations.

This means that different result sets can be completely valid depending on which implementation you are using.

But whatever will happen it's guaranteed that the effects of the unspecified code are strictly limited to the commands affected. It's not the case for undefined behaviour when the compiler is free to remove complete execution branches.

Let's see two examples of unspecified behaviour.

```
1 int x;
2 int y;
3 bool isHigher = &x > &y;
```

In other words, if you take two local variables and for any reason, you want to compare their memory addresses, it's completely unspecified whose address will be higher. There is no right or good answer, it depends on the implementation, but it doesn't have to document it.

The other example is related to expression evaluation orders. Take the following piece of code

```cpp
#include <iostream>

int x = 333;

int add(int i, int j) {
  return i + j;
}

int left() {
  x = 100;
  return x;
}

int right() {
  ++x;
  return x;
}

int main() {
  std::cout << add(left(), right()) << std::endl;
}
```

In this example, you call an adder function with two parameters, both are the results of function calls on the same line. Their evaluation order is not specified by the standard and as the invoked functions operate on a global variable, the result depends on the evaluation order.

Different compilers might give you different results.

I recommend using Wandbox if you quickly want to run your code with different compilers and versions.

By the way, in the recent standards, a lot of similar cases have been specified. Check out the references for more details.

References:

- Wikipedia
- GeeksForGeeks
- Undefined behaviour in the STL - Sandor Dargo (C++ on Sea 2020)

Question 102: What is implementation-defined behaviour?

In many ways, implementation-defined behaviour is similar to unspecified behaviour. First of all, the standard doesn't impose on how the concerned item should be implemented. That will depend on the compiler.

It's also true that the effects of implementation-defined behaviour are limited to the very commands in question. So no full execution branches can be removed, like in the case of undefined behaviour.

What differs between unspecified and implementation-defined behaviour is that the implementation must document what is a valid result set.

Let's see some examples of implementation-defined behaviour and let's step out of the C++ world for a moment.

If you think about SQL and how `ORDER BY` works on NULL values you'll find that it differs between the different implementations. Some will put NULL at the beginning of an ordered result set, some will put such values in the back. You don't have to figure this out by trying, you'll find it in the documentation, it's implementation-defined behaviour.

In C++, `sizeof(int)` and the size of integer types, in general, is implementation specified. And even the size of a byte.

References:

- Quora: What is the difference between undefined, unspecified and implementation-defined behavior?
- StakcOverFlow: Undefined, unspecified and implementation-defined behavior
- Undefined behaviour in the STL - Sandor Dargo (C++ on Sea 2020)

Question 103: What is undefined behaviour in C++?

Among unspecified, implementation-defined and undefined behaviour, this latter one is the most dangerous.

When you implement a program invoking some behaviour that is undefined, it basically means that there are no requirements on the behaviour of your whole program. The possible effects are not limited to the calls whose behaviour are unspecified, the compiler can do anything so your software can:

- crash with seemingly no reasons
- return logically impossible results
- have non-deterministic behaviour

In fact, the compiler can remove whole execution paths. Let's say you have a big function with one line invoking undefined behaviour. It's possible that the whole function will be removed from the compiled code.

When you have undefined behaviour in your code, you break the rules of the language and it won't be detected until runtime. Turning on as many compiler warnings as possible usually helps a lot in removing undefined behaviour.

Some examples of undefined behaviour:

- Accessing uninitialized variables
- Accessing objects after the lifetime ended
- Deleting objects through base class pointers without a virtual destructor

Avoid undefined behaviour like the plague.

References:

- Quora: What is the difference between undefined, unspecified and implementation-defined behavior?
- StakcOverFlow: Undefined, unspecified and implementation-defined behavior
- Undefined behaviour in the STL - Sandor Dargo (C++ on Sea 2020)

Question 104: What are the reasons behind undefined behaviour's existence?

First of all, we have to remark that the concept of undefined behaviour was not introduced by C++. It was already there in C. And what is C after all? It's just a high-level assembler that had to work on completely different platforms and architectures.

From the language designer's point of view, undefined behaviour is a way to cope with significant differences between compilers and between platforms. Some even refer to that epoch as chaotic. Different compilers treated the language differently and to be fairly backwards-compatible, a lot of details (like layout, endianness) were not defined or specified.

This gave compiler writers a lot of flexibility and they could and still can get really creative with this freedom. They can use it to simplify, to shorten, to speed up the compiled code without violating any rules.

References:

- StackExchange: Why does C++ have 'undefined behaviour' (UB) and other languages like C# or Java don't?
- Undefined behaviour in the STL - Sandor Dargo (C++ on Sea 2020)

Question 105: What approaches to take to avoid undefined behaviour?

Maybe first we can discuss what will not work.

Using try-catch blocks will not going to work, undefined behaviour is not about exceptions handled in the wrong way. Similarly, some explicit checks for example on input containers are going to work either.

```
1 std::copy_if(numbers21.begin(), numbers22.end(),
2              std::back_inserter(copiedNumbers),
3              [](auto number) {return number % 2 == 1;});
```

There is no valid check to avoid such a typo, but it's true that with some defensive programming you could handle the situation.

```
1 std::vector<int> doCopy(std::vector<int> numbers) {
2   std::vector<int> copiedNumbers;
3   std::copy_if(numbers.begin(), numbers.end(),
4                std::back_inserter(copiedNumbers),
5                [](auto number) {return number % 2 == 1;});
6   return copiedNumbers;
7 }
8
9 // ...
10 auto copiedNumbers = doCopy(numbers);
```

This is just a simple example, but the bottom line is that by limiting the scope of a function, by limiting the number of available variables, you can limit the compilable typos you can make. Yet, it doesn't look nice to wrap every algorithm like that.

So what can you do against undefined behaviour?

- You can listen to your compiler! Turn on whatever warnings you can (`-Wall`, `-Wextra`, `-Wpedantic`) and treat them as errors. They will catch a lot of undefined behaviour.

- Use a sanitizer, both g++ and clang offer some
- Follow coding best practices and naming guidelines. In the above example, the typo was only possible due to low-quality names (`numbers21` and `number22`). Legacy code have a lot of such names. Use proper, descriptive names and you won't make such typos.
- Understand the concepts behind the language. If you consider that in C++ you shouldn't pay for things you don't use, it becomes self-evident why `std::binary_search` expects a sorted input range.
- Practice contractual programming. If you want to use elements of the standard library (you should), check the contract it proposes, in other words, read their documentation, read what kind of input they expect.
- Share the knowledge! If you learn something, share it with your teammates. Organize dedicated sessions, use the team chat and code reviews.

Question 106: What is iterator invalidation? Give a few examples.

When you declare an iterator pointing at an element of a container, you might expect it to be valid all the time. There are some iterators that will be always valid and point at the place you'd expect it to point to.

For example, insert iterators (e.g. `std::back_insert_iterator`) are guaranteed to remain valid as long as all insertions are performed through this iterator and no other independent iterator-invalidating event occurs. They will even remain valid if a container has to be reallocated as it grows (notably a `std::vector`).

Also, read-only methods never invalidate iterators or references. Methods that modify the contents of a container may invalidate iterators and/or references.

Refer to this table to have a full list of when a container is invalidated.

Here is an example:

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::vector<int> numbers { 1, 2, 3, 4, 5, 6, 4};

  int val = 4;
  std::vector<int>::iterator it;
  for (it = numbers.begin(); it != numbers.end(); ++it) {
    if (*it == val) {
      numbers.erase(it);
      numbers.shrink_to_fit();
    }
  }

  std::cout << "numbers after erase:";
  for (const auto num : numbers) {
    std::cout << num << " ";
  }
}
```

When `erase` is called with `it`, its position became invalidated and what should happen in the next iteration is undefined behaviour. You might find that everything is fine, or that the results are not coherent, or event you can get a segmentation fault. Compiler the above code and check it yourself, play with the inputs.

References:

- C++ Reference: Iterator invalidation
- Stackoverflow: Iterator invalidation rules
- Undefined behaviour in the STL - Sandor Dargo (C++ on Sea 2020)

The Standard Template Library

During a couple of questions, we'll learn about the Standard Template Library, its history, concepts and some algorithms.

Question 107: What is the STL?

STL stands for the Standard Template Library, but there is no such library as the STL. It is part of the C++ standard library.

It's a set of template classes and functions to provide solutions for common problems. The elements of the STL can be divided into 4 categories:

- algorithms, e.g.: `std::find`, `std::copy`
- containers, e.g.: `std::vector<T>`, `std::list<T>`
- function objects, e.g.: `std::greater<T>`, `std::logical_and<T>`
- iterators, e.g.: `std::iterator`, `std::back_inserter`

The STL was created by Alexander Stepanov, who already had the idea of a generic data processing library in the 1970s, but there was no language support to implement his dream.

In the 80s, he made his first attempts in ADA, but that language never got widely adopted outside the defence industry.

A former colleague convinced him to present the idea to the C++ Committee that he did in November 1993. Then things happened fast. In March 1994, Stepanov submitted the formal proposal which was accepted in just a mere 4 months. In August, HP - the employer of Stepanov - published the first implementation of the STL.

Question 108: What are the advantages of algorithms over raw loops?

First, what are raw loops?

“A raw loop is any loop inside a function where the function serves a purpose larger than the algorithm. implemented by the loop.” - Sean Parent, C++ Seasoning.

Why should you prefer using std algorithms instead of such raw loops?

Algorithms are virtually bug-free

If you have to write something a thousand times, there is a fair chance that you’ll make some mistakes once in a while. It’s OK, we all make mistakes. On the other hand, if you use functions that were written before and used a million times, you won’t face any bugs.

Algorithms have a better performance

This is only partially true. If we speak about C++, functions in the `<algorithms></algorithms>` header are not optimized for corner cases. They are optimized for certain portability between different systems and container types. You can use them on any STL container without knowing their exact type. As such, we cannot assume that they can take advantage of the characteristics of the underlying datasets. Especially that they don’t operate directly on the containers, but through the iterators that give access to data behind. I say that we cannot assume, because in fact, very few people understand what is going on under the hoods of the compiler and you might find or write an implementation of the standard library that is much bigger than the usual ones, but optimized for each container type.

At the same time, chances are good that your for loops are not optimized either. And it’s alright. Of course, as you write your loops, you are in control. You can optimize them, you can get the last cycles out of them. You cannot do the same with the already written functions of a library, even if it’s the standard library.

But what reasonably could be done, was already done for the standard algorithms, and most of those things are are not performed for the raw loops we so often used. In the end, algorithms have better performance.

Algorithms are more expressive

Raw loops contain low-level code. When you call algorithms, those calls expressive.\nLet's take a simple example.\nWhat does the following piece of code do?

```
const std::vector<int> v{1, 2, 3, 4, 5};
auto ans = false;
for (const auto& e : v) {
  if (e % 2 == 0) {
    ans = true;
    break;
  }
}
```

After some thinking, we can say that it tells you if there is any even element in the vector `v`.\nHow could we make it more readable? With an algorithm:

```
const std::vector<int> v{1, 2, 3, 4, 5};

const auto ans = std::any_of(v.begin(), v.end(),
                             [](const auto& e) {return e \
% 2 ==0;});
```

How much easier is that?\nAnd this was only one simple example.

Conclusion

Algorithms are most of the time better than plain old for loops.

They are less error-prone than loops as they were already written and tested - a lot. Unless you are going for the last drops of performance, algorithms will provide be good enough for you and actually more performant than simple loops.

But the most important point is that they are more expressive. It's straightforward to pick the good among many, but with education and

practice, you'll be able to easily find an algorithm that can replace a for loop in most cases.

Question 109: Do algorithms validate ranges?

Would the following piece of code compile? What do you expect the content of `copiedNumbers` to be?

```
1 auto numbers21 = { 1, 3 }
2 auto numbers22 = { 3, 5 };
3
4 std::vector<int> copiedNumbers;
5
6 std::copy_if(numbers21.begin(), numbers22.end(),
7              std::back_inserter(copiedNumbers),
8              [](auto number) {return number % 2 == 1;});
```

`std::copy_if` takes three parameters, the start and the end iterator of the range to be copied if the third parameter (a function pointer, a function object or a lambda expression) evaluates to `true`.

`std::copy_if` - or any function from the `<algorithm>` header by the way - don't, it cannot evaluate whether the start and the end iterator, its first two parameters, belong to the same container or not. It's the responsibility of the caller to make sure that the correct parameters are passed in. If they are not respecting the rules, the result is the dreaded undefined behaviour.

So what will happen is that inside the `copy_if`, the iterator pointing at the current position will be incremented as long as it doesn't reach the end iterator. What if the address of the end is actually before the starting point? What if the two containers are next to each other? What if there is some garbage between them?

It's all undefined behaviour. You might have seemingly correct results, like the combination of the two containers, you might get a timeout or a nice core dump.

The only thing the compiler can validate is that two iterators are of the same type. So you cannot combine a `list` with a `vector` or a `vector` of `int`s with a `vector` of `float`s.

You have to always double-check that what you pass in is valid and respect the contracts imposed by the given algorithm.

Question 110: Can you combine containers of different sizes?

Explain what the following piece of code does!

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::vector<int> values{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 \
};
  std::vector<int> otherValues{ 10, 20, 30 };
  std::vector<int> results;

  std::transform(values.begin(), values.end(),
                 otherValues.begin(),
                 std::back_inserter(results),
                 [](int number, int otherNumber) {
                   return number + otherNumber;
                  });

  std::cout << "copied numbers: ";
  for (const auto& number : results) {
    std::cout << ' ' << number;
  }
  std::cout << '\n';

  return 0;
}
```

First, what can be the intention behind such code? With `std::transform` we try to combine the content of two vectors, by adding up the nth element of the `values` with the nth element of the `otherValues` and we push the sum to the `results` vector.

It's simple code, but there is a catch. The two ranges that are passed in, don't have the same amount of elements.

Remember, `std::transform` takes the first range by a begin and an end iterator and a second input range (which is not mandatory by the way) only by the begin iterator.

Like all the similar functions in the `<algorithm>` header, `std::transform` assumes and in fact, expects that the second input range has at least as many elements as the first.

But what if this expectation is not met?

You might expect that a zero-initialized item will be taken instead, but it's not the case, even though it's easy to come up with some example code that would support this assumption, but no.

It's undefined behaviour.

In most cases, the runtime will just take any value that it finds in the next memory address even if that does not belong to the vector.

So you always have to make sure that the second input range, defined only by its starting point is always at least as long as the first one.

Question 111: How is a `vector`'s memory layout organized?

If you create a vector locally on the stack (without using `new`) you might expect that you will have the data on the stack. But in fact, the vector object on the stack itself is quite small and it consists of a pointer that point at some dynamically allocated memory on the heap.

So on the stack, you'll have the above-mentioned pointer and some other extra variables to keep track of the size and the capacity. You can take a look at this illustration.

Other implementations are also possible when iterators are stored pointing at the beginning of the allocated memory, at the current end and at the end of the total capacity. Check this illustration.

The bottom line is that you have some variables in the main vector object that help to find the data that is actually stored on the heap. That memory on the heap must be contiguous and if your vector grows and an extension is required it might happen that the whole content must be copied to somewhere else.

In order to protect yourself from these extra copies, if you know how big your vector will grow, it's good to reserve that capacity right away after its declaration, by calling `std::vector<T>::reserve(maxCapacity)`

References:

- Frogatto: How C++'s vector works: the gritty details
- STHU.org: Array-like C++ containers: Four steps of trading speed

Question 112: 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.

```
1 class Squad : public std::vector {
2   using std::vector::vector;
3   // ...
4 };
```

It's simple, it's readable, yet you'll find a lot of people at different forums who will tell you that this is the eighth deadly sin and if you are a serious developer you should avoid it at all costs.

Why do they say so?

There are two main arguments. One is that algorithms and containers are well-separated concerns in the STL. The other one is about the lack of virtual constructors.

But are these valid concerns?

They might be. It depends.

Let's start with the one about the lack of a virtual destructor. It seems more practical.

Indeed, 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.

On the other hand, this also means that those who rigidly oppose inheriting from `std::vector` - or from any class without a virtual destructor - because of undefined behaviour and memory leaks, are not right.

If you know what you are doing, and you only use this inheritance to introduce a strongly typed vector, not to introduce polymorphic behaviour and additional states to your container, you are perfectly fine to use this technique. Simply, you have to respect the limitations, though probably this is not the best strategy to use in the case of a public library. But more on that just in a second.

So the other main concern is that you might mix containers and algorithms in your new object. And it's bad because the creators of the STL said so. And so what? Alexander Stepanov who originally designed the STL and the others who have been later contributed to it are smart people and there is a fair chance that they are better programmers than most of us. They designed functions, objects that are widely used in the C++ community. I think it's okay to say that they are used by everyone.

Most probably we are not working under such constraints, we are not preparing something for the whole C++ community. We are working on specific applications with very strict constraints. Our code will not be reused as such. Never. We don't work on generic libraries, we work on one-off business applications.

As long as we keep our code clean (whatever it means), it's perfectly fine to provide a non-generic solution.

As conclusion, we can say that for application usage, inheriting from containers in order to provide strong typing is fine, as long as you don't start to play with polymorphism.

Question 113: What is the type of myCollection after the following declaration?

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

The type is `std::initializer_list`. 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="">()'</brace-enclosed></auto>. So no empty container, but a compilation error.

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 an 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 an std::initializer_list parameter
- a braced-init-list is bound to auto, including in a ranged for loop

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

Question 114: What are the advantages of const_iterators over iterators?

Iterators are like pointers to const objects but in the STL world. They point to values that cannot be modified. The usage of `const_iterators` simply follows the same principles as using `const` variables wherever it makes sense.

Since C++11 all STL containers have `cbegin` and `cend` member functions producing `const_iterators` even if the container itself is not const.

Since C++14, you also have `cbegin`, `cend`, `crbegin`, `crend` non-member functions available. It's better to use the non-member functions in case you want generic code, because these will also work on built-in arrays, not like the member functions.

Question 115: Binary search an element with algorithms!

What do you expect that this piece of code does? Will it find `7`?

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4
5 int main() {
6   std::vector<int> numbers{1,54,7,5335,8};
7   std::cout << std::binary_search(numbers.begin(), number\
8 s.end(), 7) << std::endl;
9 }
```

It will most probably not find the position of `7` and even if you have a good result, you cannot rely on it. In fact, the above code has undefined behaviour.

It's all about contracts and principles. In C++, one of the foundational concepts is that you should not pay for what you don't use. According to this spirit, `std::binary_search` and some other algorithms expect that its input range is sorted. After all, search algorithms should not be responsible for sorting and those who already pass in a sorted range should not pay for a needless sorting attempt.

So to make this code working, you just have to sort the vector before passing it to the search.

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
  std::vector<int> numbers{1,54,7,5335,8};
  std::sort(numbers.begin(), numbers.end());
            std::cout << std::binary_search(numbers.begin\
(),
                                            numbers.end(),\
 7) << '\n';}
```

The bottom line is that algorithms come with their contracts which you should consult first and - of course - respect.

Reference:

- Undefined behaviour in the STL - Sandor Dargo (C++ on Sea 2020)

Question 116: What is an Iterator class?

As you probably remember, the Standard Template Library is composed of containers, algorithms, functors and iterators. Iterators are the connecting dots between containers and functors.

In the STL, before ranges appeared, algorithms never operated directly on the containers, but on iterators instead.

An iterator is like a pointer. It's an object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators (with at least the increment (++) and dereference (*) operators).

Iterators have different categories and their classification depends on the functionality they implement. The list goes from the most generic towards the most specialized category:

- Random Access
- Bidirectional
- Forward
- Input
- Output (input and output are, in fact, on the same level)

To get more detailed information about the categories, check out this link.

References:

- CPlusPlus.com: Iterator
- Learn C++: Introduction to iterators

Miscalleanous

During the rest of this book, we are going to cover various topics, such as some tricky code problems, optimizations, and the C++ core guidelines.

Question 117: Can you call a virtual function from a constructor or a destructor?

Technically you can, the code will compile. But the code can be misleading and it can even lead to undefined behaviour.

Attempting to call a derived class function from a base class under construction is dangerous.

```
#include <iostream>

class Base {
 public:
  Base() {
    foo();
  }
 protected:
  virtual void foo() {
    std::cout << "Base::foo\n";
  }
};

class Derived : public Base {
 public:
  Derived() { }
  void foo() override {
    std::cout << "Derived::foo\n";
  }
};

int main() {
  Derived d;
}
```

The output is simply `Base::foo`:

- By contract, the derived class constructor starts by calling the base class constructor.
- The base class constructor calls the base member function and not the one overridden in the child class, which is confusing for the child class' developers.

In case, the virtual method is also a pure virtual method, you have undefined behaviour. If you're lucky, at link time you'll get some errors.

What's the solution?

The simplest probably is just to fully reference the function that you'll call:

```
1 Base() {
2   Base::foo();
3 }
```

In the case of the Base class, it can only be that, when you're in the Derived class, you can decide. A more elegant solution is if you wrap the virtual functions in non-virtual functions and from the constructors/destructors you only use them.

For full source code, check out the references.

References:

- SEI CERT: OOP50-CPP. Do not invoke virtual functions from constructors or destructors
- SonarSource: RPSEC-1699

Question 118: What are default arguments? How are they evaluated in a C++ function?

A default parameter is a value that is assigned to a parameter while declaring a function.

A default argument allows a function to be called without providing one or more trailing arguments.

The default value for a parameter is indicated at function declaration time at the parameter list. We simply assign a value to the parameter in the function declaration.

Here is an example:

```
int calculateArea(int a, int b);
int calculateArea(int a, int b=2);
int calculateArea(int a=3, int b=2);
```

These default values are used if one or more arguments are left blank while calling the function - according to the number of left blank arguments.

Let's see a complete example. If the value is not passed for any of the parameters during the function call, then the compiler uses the default value(s) provided. If a value is specified, then the default value is stepped on and the passed value is used.

```
#include <iostream>

int calculateArea(int a=3, int b=2) {
  return a*b;
}

int main() {
  std::cout << calculateArea(6, 4) << '\n';
  std::cout << calculateArea(6) << '\n';
  std::cout << calculateArea() << '\n';
}
/*
24
12
6
*/
```

As shown in the above code, there are three calls to `calculateArea` function. In the first call, we pass both arguments so the default values are overridden by the caller, in the second only one, so the last parameter defaults. In the last call, we provide no argument, so both parameters take their default values.

In a function declaration, after a parameter with a default argument, all subsequent parameters must have a default argument supplied in this or a previous declaration from the same scope…unless the parameter was

expanded from a parameter pack, and keep in mind that ellipsis(. . .) is not a parameter.

This means that that the `int calculateArea(int a=5, int b);` would not compile. On the other hand, `int g(int n = 0, ...)` would compile as ellipsis does not count as a parameter.

Default arguments are only allowed in the parameter lists of function declarations and lambda-expressions, (since C++14) and are not allowed in the declarations of pointers to functions, references to functions, or in typedef declarations.

One last thing to note is that you should always declare your defaults in the header, not in the cpp file.

Question 119: Can virtual functions have default arguments?

Yes, they can, but you should not rely on this feature, as you might not get what you'd expect.

While it's perfectly legal to use default argument initializers in virtual functions, there is a fair chance over the maintenance period, changes will lead to incorrect polymorphic code and unnecessary complexity in a class hierarchy.

Let's see an example:

```
1  #include <iostream>
2
3  class Base {
4   public:
5    virtual void fun(int p = 42) {
6      std::cout << p << std::endl;
7    }
8  };
9
10 class Derived : public Base {
11  public:
12   void fun(int p = 13) override {
13     std::cout << p << std::endl;
14   }
```

```
15 };
16
17 class Derived2 : public Base {
18  public:
19   void fun(int p) override {
20     std::cout << p << std::endl;
21   }
22 };
```

What would you expect from the following `main` function?

```
1 int main() {
2   Derived *d = new Derived;
3   Base *b = d;
4   b->fun();
5   d->fun();
6 }
```

If you expected, the following, congrats!

```
1 42
2 13
```

If not, don't worry. It's not evident. `b` points to a derived class, yet `B`'s default value was used.

Now, what about the following possible `main`?

```
1 int main() {
2   Base *b2 = new Base;
3   Derived2 *d2 = new Derived2;
4   b2->fun();
5   d2->fun();
6 }
```

You might expect 42 twice in a row, but that's incorrect. The code won't compile. The overriding function doesn't "inherit" the default value, so the empty `fun` call to Derived2 fails.

```
1 main.cpp: In function 'int main()':
2 main.cpp:28:10: error: no matching function for call to '\
3 Derived2::fun()'
4    28 |   d2->fun();
5       |   ~~~~~~~^~
6 main.cpp:19:8: note: candidate: 'virtual void Derived2::f\
7 un(int)'
8    19 |   void fun(int p) override {
```

```
 9       |         ^~~
10 main.cpp:19:8: note:   candidate expects 1 argument, 0 pr\
11 ovided
```

Now let's modify a bit our original example and we'll ignore Derived2.

```
 1 #include <iostream>
 2
 3 class Base {
 4  public:
 5   virtual void fun(int p = 42) {
 6     std::cout << "Base::fun " << p << std::endl;
 7   }
 8 };
 9
10 class Derived : public Base {
11  public:
12    void fun(int p = 13) override {
13      std::cout << "Derived::fun" << p << std::endl;
14    }
15 };
16
17 int main() {
18   Derived *d = new Derived;
19   Base *b = d;
20   b->fun();
21   d->fun();
22 }
```

What output do you expect now?

It is going to be:

```
1 Derived::fun 42
2 Derived::fun 13
```

The reason is that a virtual function is called on the dynamic type of the object, while the default parameter values are based on the static type. The dynamic type is `Derived` in both cases, but the static type is different, hence the different default values are used.

Given these differences compared to normal polymorphic behaviour, it's best to avoid any default arguments in virtual functions.

References:

- GotW.ca: Herb Sutter
- SonarSource

Question 120: Should base class destructors be virtual?

In the last couple of days, we've been talking about polymorphism, inheritance and whatever is implied. It's not a big surprise that we are going to discuss destructors once again.

While the usual answer is "yes, of course", it's not the correct answer. If you just think about the standard library itself, many classes don't have a virtual destructor. That's something I wrote about in strongly-types containers. So if for example `std::vector` doesn't have a virtual destructor, the answer "of course", cannot be correct.

According to Herb Sutter, "a base class destructor should be either public and virtual or protected and nonvirtual."

Any operation that will be performed through the base class interface, and that should behave virtually, should be virtual. If deletion, therefore, can be performed polymorphically through the base class interface, then it must behave virtually and must be virtual. The language requires it - if you delete polymorphically without a virtual destructor, you have to face something that is called undefined behaviour. Something that we always want to avoid.

```
1 class Base { /*...*/ };
2
3 class Derived : public Base { /*...*/ };
4
5 int main() {
6   Base* b = new Derived;
7   delete b; // Base::~Base() had better be virtual!
8 }
```

But base classes need not always allow polymorphic deletion. In such cases, make base class destructor protected and non-virtual. It'll make deletes through base class pointers fail:

```cpp
class Base {
  /*...*/
 protected:
   ~Base() {}
};

class Derived : public Base { /*...*/ };

int main() {
  Base* b = new Derived;
  delete b; // error, illegal
}
/*
main.cpp: In function 'int main()':
main.cpp:11:10: error: 'Base::~Base()' is protected withi\
n this context
   11 |   delete b; // error, illegal
      |          ^
main.cpp:4:4: note: declared protected here
    4 |    ~Base() {}
      |    ^

*/
```

References:

- GotW.ca: Herb Sutter
- Sandor Dargo's Blog

Question 121: What is the function of the keyword mutable?

`mutable` has two functions, two roles since C++11.

`mutable` specifier

Let's start with its older meaning.\n`mutable` permits modification of the class members even if they are declared `const`.

It may appear in the declaration of a non-static class members of non-reference non-const type:

```cpp
class X {
  mutable const int* p; // OK
```

```
3   mutable int* const q; // ill-formed
4 };
```

Mutable is used to specify that the member does not affect the externally visible state of the class (as often used for mutexes, memo caches, lazy evaluation, and access instrumentation).

So in case you have a `const` object, it's mutable members can be modified.

Another way to use mutable is in case of lazy initialization. Getters should be generally `const` methods as they are only supposed to return class members, and not alter them.

But when you apply lazy initialization, the first time you call the getter, it will actually alter the member. It will initialize it maybe from the DB, via the network, etc. So you cannot make that member const even if its value will not change after initialization.

If you use the mutable keyword, you can. Just pay attention not to do other things to that member.

```
1 class SomethingExpensive{
2   //...
3 };
4
5 class A{
6  public:
7   SomethingExpensive* getSomethingExpensive() const {
8     if (_lazyMember == nullptr) {
9       _lazyMember = new SomethingExpensive();
10    }
11    return _lazyMember;
12  }
13
14  private:
15    mutable SomethingExpensive* _lazyMember;
16 };
```

Mutable lambda expressions

If you declare a lambda mutable, then it allows the lambda to modify the objects captured by copy, and to call their non-const member functions. Otherwise, it's not possible.

```cpp
#include <iostream>

struct A{
  void a() const {
    std::cout << "a\n";
  }

  void b() {
    std::cout <<"b\n";
  }
};

int main(){
  A a;

  // error: passing 'const A' as 'this'
  // auto l = [a]() {a.b();};

  auto lm = [a]() mutable {a.b();};
  return 0;
}
```

It's worth trying in C++ Insights the above code and try to check what's the difference when you declare a lambda mutable and not. You will find that when a lambda is non-mutable, then the `operator()` is const. When it's mutable it becomes non-const. You can also observe that the generated object takes the captured variables as members. As such, it becomes straightforward why a regular lambda expression cannot call non-const functions on the captured variables.

Question 122: What is the function of the keyword volatile?

It is a keyword to declare that the corresponding variable is volatile and thereby tells the compiler that the variable can change externally and as such, the compiler optimization on the variable reference should be avoided.

What does this mean in plain English?

It means that if you declare a variable `volatile`, you acknowledge that the value stored at the variable's memory address might change independently.

How could that happen?

It might happen because of changes introduced by hardware or by another thread.

Speaking of this latter one, while it's possible, according to the C++11 ISO Standard, the volatile keyword is only meant for use for hardware access; do not use it for inter-thread communication.

For inter-thread communication, the standard library provides `std::atomic` templates.

While the original guarantees for `volatile` promised that the order of assignments to volatile qualified variable is preserved, it does not mean that no reordering will happen around volatile assignments.

So don't use `volatile` for inter-thread communication, it is a type qualifier that you can use in order to declare that an object can be modified in the program by the hardware.

A memory model was added to the C++11 standard to support multi-threading. But in C++ the decision was to keep the keyword volatile as a mean to solve the original memory mapped I/O problems.

Question 123: What is an inline function?

A function prefixed with the keyword `inline` before the function definition is called as inline function. If a function is `inline` the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because the compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, simply place the keyword inline before the function name and define the function before any calls are made to the function. It's important to remember that it's often only a hint to the compiler. The

compiler can ignore the inline qualifier in case the defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

To understand how `inline` functions can help, we should understand the costs of a regular function call.

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function.

This can become overhead if the execution time of a function is less than the switching time from the caller function to called function. For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because the execution time of a small function is less than the switching time.

What we should take away is that the usage of the `inline` keyword is only a request to the compiler, it is NOT a command. The compiler might ignore it and at the same time, it might inline functions as a form of optimization even if it was not requested. It can bring some performance advantages in case the function is small and often used, but at the same time, it can damage the caching punctuality and bloat binary size.

References:

- CPlusPlus.com
- Geeks For Geeks
- Tutorials Point

Question 124: What do we catch?

Consider the following piece of code. What is going to be the output if we uncomment the commented print statements in the two catch blocks? And why so?

```
#include <iostream>
#include <exception>

class SpecialException : public std::exception {
 public:
  virtual const char* what() const throw() {
    return "SpecialException";
  }
};

void a() {
  try {
    throw SpecialException();
  } catch (std::exception e) {
    // std::cout << "exception caught in a(): " << e.what\
() << '\n';
    throw;
  }
}

int main () {
  try {
    a();
  } catch (SpecialException& e) {
    // std::cout << "exception caught in main(): " << e.w\
hat() << '\n';
  }
}
```

The output is apart from a compiler warning advising you not to catch anything by value is:

```
exception caught in a(): std::exception\nexception caught\
 in main(): SpecialException\n
```

Let's have a look at the code once again. There is a new exception type declared (1). In function `a()` we throw it (2) and then right there we catch a quite generic `std::exception` by value (3). After logging it, we rethrow the exception (4). In `main()`, we catch our custom exception type by `const` reference (5):

```
#include <iostream>
#include <exception>

class SpecialException : public std::exception { // 1
 public:
  virtual const char* what() const throw() {
    return "SpecialException";
  }
};

void a() {
  try {
    throw SpecialException(); // 2
  } catch (std::exception e) { // 3
    std::cout << "exception caught in a(): " << e.what() \
<< '\n';
    throw; // 4
  }
}

int main () {
  try {
    a();
  } catch (SpecialException& e) { //5
    std::cout << "exception caught in main(): " << e.what\
() << '\n';
  }
}
```

When we caught and logged a standard exception by value (3), we lost some of the information. Even though originally a `SpecialException` was thrown (2), in order to squeeze it into an `std::exception` variable, the compiler had to get rid of some parts of that exception. In other words, it got sliced. Had we caught it by reference, we would have kept its original type.

Yet, when you rethrow an exception simply by calling `throw;`, it will rethrow the original exception which is `SpecialException` in our case. If you catch an exception by value, the exception that was the source of the copy will be rethrown. Hence any modification - including the slicing - is lost.

So in the above case, the original exception is rethrown (4), not the one we use within the `catch` block, but the one that left the `try` block. We keep that narrower `SpecialException`.

A general rule of thumb is to always catch exceptions by reference to avoid superfluous copies and to be able to make persistent changes to the caught exception.

Question 125: What are the differences between references and pointers?

Both references and pointers can be used to pass even local variables from one function to another. If you use any of them, the object referenced/pointed will not get copied and changes made to their value/state in the called function will be visible in the caller function as well.

Both pointers and references can also be used to gain performance by saving the resources needed to copy big objects, e.g. when the objects are passed into a function or when they are returned.

Despite these similarities, there are also some differences between references and pointers.

1. Once a reference is created, it cannot be later changed to reference another object; it cannot be reseated. On the other hand, this is often done with pointers.

2. References cannot be `NULL`/`nullptr`. Pointers are often pointing at the `nullptr` to indicate that they are not pointing to any valid thing. Though, it's worth noting that since C++17 there is also `std::optional` in the standard library if we need to indicate that an object is "not there".

3. A reference must be initialized when declared. There is no such restriction with pointers.

Due to the above limitations, references in C++ cannot be used for implementing data structures like linked lists, trees, etc.

While the above restrictions prevail, at the same time references are safer and easier to use:

1. Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location, they are called dangling references.

2. Easier to use: References don't need the dereferencing operator (`*`) to access the value. They can be used like normal variables. The address of (`&`) operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator (`.`), unlike pointers where arrow operator (`->`) is needed to access members.

Question 126: Which of the following variable declarations compile and what would be the value of `a`?

1. `unsigned int a = 42;`
2. `unsigned int a{42};`
3. `unsigned int a = -42;`
4. `unsigned int a{-42};`

1-2) That's fairly simple. We declare an `unsigned int` in both cases and we initialize them with a positive number. They both compile and if you print the value of `a`, it'll be 42 in both cases, there is nothing to see here, move along.

3. This is getting more interesting. We initialize an `unsigned int` with a negative number. But an unsigned integer is supposed to represent a positive number. The signed -42 will be converted into an unsigned number. While the value might depend on your platform, but in general the size of an (unsigned) int is 32 bits, so the max value of an unsigned number is 4294967295 (2**32-1, as we start from zero).

If you imagine a number line where there are no negative numbers, left to zero will be the largest number so the value of -42 will be the maximum value of an unsigned integer minus 41. Why 41 and not 42? Because -1 in fact equals the highest number that we can represent. From there we have to step 41 to the left to get to -42, which will be exactly `std::numeric_limits<unsigned int>::max()-41</unsigned>`.

By the way, no matter what platform you use `std::numeric_limits<unsigned int>::max()</unsigned>` will give you the actual highest value for an `unsigned int` and as you could guess you can pass in other types as a template parameter.

4. Finally something that doesn't compile. The uniformed or {}-initialization introduced by C++11 doesn't allow narrowing. Depending on your compiler and its version, you either get an error or a warning, but if you follow the industry recommendations, you will treat warnings as errors, so it shouldn't matter.

Again, just to emphasize the fact. Initializing with = allows narrowing conversions hence -42 became 4294967254, while if you `{}`-initialize, your compiler will complain with either a warning or an error.

Question 127: What will the line of code below print out and why?

```
1 #include <iostream>
2
3 int main(int argc, char **argv) {
4   std::cout << 25u - 50;
5   return 0;
6 }
```

The answer is not `-25`. Rather, the - possibly surprising - answer is 4294967271, assuming 32 bit integers.

But why?

In C++, if the types of two operands differ from one another, then the operand with the "lower" or "smaller" type will be **promoted** to the type of the "higher" or "larger" type operand implicitly.

This promotion which is a special type of implicit conversions, uses the following type hierarchy (from highest type to lowest type):

- long double
- double

- float
- unsigned long int
- long int
- unsigned int
- int

So when the two operands are, as in our example, 25u (`unsigned int`) and 50 (`int`), the 50 is promoted to also being an `unsigned int` (i.e., 50u).

The result of the operation will be of the type of the operands. Therefore, the result of 25u - 50u will itself be an `unsigned int` as well. So the result of -25 converts to 4294967271 when promoted to being an `unsigned int`.

In fact, the result is the maximum value of an unsigned integer - (50 - 25 + 1). The plus one is there, because, between 1 and the maximum value of an unsigned int, there is zero as well that we have to take into account.

```
1 #include <iostream>
2 #include <numeric>
3
4 int main() {
5   auto a = std::numeric_limits<unsigned int>::max()+1 - 2\
6 5;
7   auto b = 25u - 50;
8   std::cout << std::boolalpha << (a == b) << '\n';
9 }
```

By the way, this is a nice instance of an integer underflow which is considered a vulnerability.

Question 128: Explain the difference between pre- and post-increment/decrement operators

C++ has a nice syntactical sugar to add or subtract 1 from a variable. It's so renowned that it appears even in the name of the language. It's the `++` (increment) and the `--` (decrement) operators.

Both can be used as a prefix or a postfix operator.

```
1 int main() {
2   int a=5;
3   a++; // postfix increment; a is 6
4   ++a; // prefix increment; a is 7
5   a--; // postfix decrement; a is 6
6   --a; // prefix decrement; a is 5 again
7 }
```

So what's the difference between the prefix and postfix operators?

The difference between the meaning of pre and post depends upon how the expression is evaluated and the result is stored.

In the case of the pre-increment/decrement operator, the increment/decrement operation is carried out first, and then the result is passed to an lvalue.

Whereas for post-increment/decrement operations, the lvalue is evaluated first and then increment/decrement is performed.

To view this from another perspective, the prefix operator returns a reference of the variable (`T& T::operator++();`), while the postfix a copy (`T T::operator++(int);`).

This implies that while in the language name, there is a postfix increment (C++), in our old raw for loops we got used to writing `i++` to increment the index, actually it's better to use the prefix operators by default unless we have a specific need for the postfix and therefore a copy.

In most cases, you can only gain negligible performance with the prefix operators, but in certain cases, when you have big objects overloading these operators, you might gain a more significant amount of time, especially when the returned value is not used.

Question 129: What are the final values of a, b and c?

Consider the following small program. What are the final values of a, b and c?

```
int main(){
  int a, b, c;
  a = 9;
  c = a + 1 + 1 * 0;
  b = c++;
  return 0;
}
```

This short test covers both simple operator precedence and basic knowledge about the increment operator that we discussed just yesterday. The difficulty of this exercise comes from the line `b = c++;`, you have to understand the difference between prefix and postfix increments (`++c` vs `c++`).

Correct answer:

```
a = 9 // was not modified
b = 10
c = 11
```

In case you guessed that b is 11, let's get back to the postfix increment. What happens is that first, `c`'s value is copied into `b` (both `b` and `c` is 10 at the moment) and just then `c` is incremented and becomes 11. But that doesn't affect `b`'s value. If instead, we had the line `b = ++c`, both would become 11.

Another mistake could be that you guessed that c is either 0 or 1.

This will occur if there is an error in the evaluation of precedence. Because the multiplication operator (`*`) has higher precedence than addition (`+`), and the order of operation is otherwise left to right, this equation translates to the following:

```
c = ((a + 1) + (1 * 0))
c = ((9 + 1) + 0)
c = 10 // correct (it will be incremented later)
```

If the correct order of precedence is not followed then one possibility is:

```
c = a + 1 + 1 * 0
c = 9 + 1 + 1 * 0
c = 10 + 1 * 0
c = 11 * 0
c = 0 // this is wrong
```

Question 130: Does this string declaration compile?

Will the following code compile? If not, why not? If it does, why?

There is no trick, this is the whole application, `foo` is nowhere declared as a global variable.

```
#include <iostream>

int main() {
  std::string(foo);
}
```

Whether this code compiles or not depends on whether you treat warnings as errors or not. Let's assume you don't handle warnings as errors.

This code does exactly one thing, it creates an empty string and assigns it to a variable called `foo`.

The following two lines mean the same:

```
std::string(foo);
std::string foo;
```

Why? Because according to the C++ standard whatever can be interpreted as a declaration, it is interpreted as a declaration. It's also called the [most vexing parse](#).

Why does this matter? Who writes anything like that in real life?

We only have to complexify this example a little bit. Think about a `mutex`.

```
#include <mutex>

static std::mutex m_mutex;
static int shared_resource;

void increment_by_42() {
  std::unique_lock<std::mutex>(m_mutex);
  shared_resource += 42;
}
```

What is happening here?

At the beginning of this mail, you might have thought about that okay, we create a temporary unique_lock, locking mutex `m_mutex`. Well. No. I think you can tell on your own what's happening there. It creates a lock on the type of a mutex and called that lock `m_mutex`. And nothing got locked.

But if you express your intentions by naming that lock, or if you brace initialization it'll work as expected.

```
#include <mutex>

static std::mutex m_mutex;
static int shared_resource;

void increment_by_42() {
  std::unique_lock aLock(m_mutex); // this works fine
  // std::unique_lock<std::mutex> {m}; // even this would\
 work fine
  shared_resource += 42;
}
```

By the way, using `-Wshadow` compiler option would have also caught the problem by creating a warning. Treat all warnings as errors and be safe!

Question 131: What are Default Member Initializers in C++?

Default member initialization is available since C++ 11.

It lets you initialize class members where they are declared not in the constructors.

```
class T {
 public:
  T()=default;
  T(int iNum, std::string iText) : num(iNum), text(iText)\
 {};

  private:
   int num{0}; // Default member initialization
   std::string text{}; // Default member initialization
};
```

In the above example, the members `num` and `text` are initialized with 0 and with an empty string right where they are declared. As such, we can keep the default constructor simpler.

It has at least two advantages.

- If you consistently follow this practice you will not have to worry that you forgot to initialize something
- And you'll not have to scroll anywhere else to find the default value of the variable.

We can still override those values in any constructors. In case, we initialize a member both in-place and in a constructor, the constructor wins.

There is no performance overhead, the constructor will not reinitialize the member for a second time.

You might ask if it means that the members will first be assigned to their default value and then reassigned with the values from the constructor. The compiler is smart enough to know which value to use and it avoids the extra assignments. The C++ Core guidelines also encourages us to use default member initialization for initialization data members instead of the default constructor.

Question 132: What is the most vexing parse?

The most vexing parse is a specific form of syntactic ambiguity resolution in the C++ programming language. The term was used by Scott Meyers in Effective STL. It is formally defined in section 8.2 of the C++ language standard.

It means that whatever that can be interpreted as a function declaration, will be interpreted as a function declaration.

Take the following example:

```
1 std::string foo();
```

Probably this is the simplest form of the most vexing parse. The unsuspecting reader might think that we just declared a string called foo and called its default constructor, so initialized it as an empty string.

Then, for example, when we try to call `empty()` on it, and we have the following error message (with gcc):

```
main.cpp:18:5: error: request for member 'empty' in 'foo'\
, which is of non-class type 'std::string()' {aka 'std::_\
_cxx11::basic_string<char>()'
```

What happened is that the above line of code was interpreted as a function declaration. We just declared a function called foo, taking no parameters and returning a string. Whereas we only wanted to call the default constructor.

This can give a kind of headache to debug even if you know about the most vexing parse. Mostly because you see the compiler error on a different line, not when you declare your ~~varibale~~ function, but when you try to use it.

This can be fixed very easily. You don't need to use parentheses at all to declare a variable calling its default constructor. But since C++11, if you want you can also use the {}-initialization. Both examples are going to work just fine:

```
std::string foo;
std::string bar{};
```

Now let's have a look at a bit more interesting example:

```
#include <iostream>
#include <string>

struct MyInt {
  int m_i;
};

class Doubler {
 public:
  Doubler(MyInt i) : my_int(i) {}
  int doubleIt() {
    return my_int.m_i*2;
  }
```

```
14
15  private:
16   MyInt my_int;
17 };
18
19 int main() {
20   int i=5;
21   Doubler d(MyInt(i));
22
23   std::cout << d.doubleIt() << std::endl;
24 }
```

There are 3 ways to fix it:

- Declare the MyInt object outside of the call, on the previous line, but then it won't be a temporary anymore.
- Replace any or both of the parentheses with the brace initialization. Both `Doubler d{MyInt(i)};` or `Doubler d(MyInt{i})` would work, just like `Doubler d{MyInt{i}}`. And this third one is inconsistent at least in how we call the constructors. The potential downside is that this only works since C++11.
- If you are using an older version of C++ than C++11, you can add an extra pair of parentheses around the argument that is meant to be sent to the constructor: `Doubler d((MyInt(i)))`. This also makes it impossible to parse it as a declaration.

Question 133: Does this code compile? If yes, what does it do? If not, why not?

Does this code compile? If yes, what does it do? If not, why not?

```
1 class MyObject {
2  public:
3   void doSomething() {}
4  private:
5   // ...
6 };
7
8 int main() {
9   MyObject o();
10  o.doSomething();
11 }
```

The code doesn't compile and the error message is this:

```
main.cpp: In function 'int main()':
main.cpp:11:4: error: request for member 'doSomething' in\
 'o', which is of non-class type 'MyObject()'
   11 |  o.doSomething();
      |    ^~~~~~~~~~
```

So `o` in of non-class type `MyObject()`. In other words, `o` is a function taking no parameters and returning a `MyObject`.

This is the infamous most-vexing parse that we discussed a month ago. It is a specific form of syntactic ambiguity resolution in the C++ programming language. The term was used by Scott Meyers in Effective STL. It is formally defined in section 8.2 of the C++ language standard. It means that whatever that can be interpreted as a function declaration, will be interpreted as a function declaration.

Hence the line `MyObject o();` is interpreted not as a variable declaration, but a function declaration.

What would happen if we called `o`?

```
class MyObject{
 public:
  void doSomething() {}
 private:
  // ...
};

int main() {
 MyObject o();
 o();
}
/*
main.cpp:(.text.startup+0x5): undefined reference to `o()'
collect2: error: ld returned 1 exit status
*/
```

Before we saw an error at compilation time, now we see it at linking time. As said, we declared a function, but it is nowhere defined, hence the undefined reference linking error.

The simplest way to fix the error is to simply omit the braces: `MyObject o;`. Since C++11 we can also use aggregate or`. The potential downside is that it only works since C++11, but that's less and less an error.

Question 134: What is `std::string_view` and why should we use it?

`std::string_view` has been introduced by C++17 and a typical implementation needs two information. The pointer to a character sequence and its length. The character sequence can be both a C++ or a C-string. After all, `std::string_view` is a non-owning reference to a string.

This type is needed because it's quite cheap to copy it as it only needs the above-mentioned copy and its length. We can effectively use it for read operations, and besides it received the remove_prefix and remove_suffix modifiers.

As you might have guessed, the reason why `string_view` has these two modifiers, because it's easy to implement them without modifying the underlying character sequence. Either the start of the character sequence or its length should be modified.\n \nIf your function doesn't need to take ownership of the `string` argument and you only need read operations (plus the above mentioned two modifiers) use a `string_view` instead. In fact, `string_view` should replace all `const std::string&` parameters.

There is one drawback though. As under the hood you can either have a `std::string` or a `std::string_view`, you lose the implicit null termination. If you need that, you have to stick with std::string (const&)

The above-mentioned fact that it's quite cheap to copy, and that it has one less level of pointer indirection than a `std::string&` can bring a significant performance gain. If you're interested in the details, please read this article.

Question 135: How to check if a string starts or ends with a certain substring?

Unlike Python or other languages, C++'s string type didn't have `starts_with/ends_with` functions that could easily tell if a string starts with a certain prefix or ends with a given suffix.

This has changed with C++20, where `starts_with` and `ends_with` were added both to `std::sting` and `std::string_view` and performing such checks became extremely simple and readable!

```
#include <iostream>
#include <string>

int main() {
  std::string s{"what a string"};
  std::cout << std::boolalpha;
  std::cout << s.starts_with("what") << '\n';
  std::cout << s.ends_with("not") << '\n';
}
/*
true
false
*/
```

Before C++20 it was still easy to perform similar tests, though a bit less readable and more verbose.

In order to check if a string starts with a prefix, one could use `std::string::find` and check if the returned position is one of the first characters, `0`.

To verify the suffix is a bit more complex and less readable. We can use `std::string::compare`. After making sure that the string to be checked is at least as long as the suffix we want to verify, we have to pass the starting position of the potential suffix, its length and the suffix itself.

Wrapping it to a function it looks like this:

```
#include <iostream>
#include <string>

bool ends_with(const std::string& str, const std::string&\
 suffix) {
  if (str.size() >= suffix.size()) {
    return str.compare(str.size() - suffix.size(), suffix\
.size(), suffix) == 0;
  }
```

```
  return false;
}

int main() {
  std::string s{"what a string"};
  std::cout << std::boolalpha;
  std::cout << (s.find("what") == 0) << '\n';
  std::cout << (s.find("a") == 0) << '\n';
  std::cout << ends_with(s, "string") << '\n';
  std::cout << ends_with(s, "what") << '\n';
}
```

If you have access to boost, you might use `boost::algorithm::starts_with`/`boost::algorithm::ends_with`:

```
#include <iostream>
#include <string>
#include <boost/algorithm/string/predicate.hpp>

int main() {
  std::string s{"what a string"};
  std::cout << std::boolalpha;
  std::cout << boost::algorithm::starts_with(s, "what") <\
< '\n';
  std::cout << boost::algorithm::starts_with(s, "a") << '\
\n';
  std::cout << boost::algorithm::ends_with(s, "string") <\
< '\n';
  std::cout << boost::algorithm::ends_with(s, "what") << \
'\n';
}
```

References:

- Boost Header
- C++ Reference: compare
- C++ Reference: starts_with
- C++ Reference: ends_with
- Find out if string ends with another string in C++ - Stackoverflow

Question 136: What is RVO?

RVO stands for return value optimization.

It is a compiler optimization that involves eliminating the \ntemporary object created to hold a function's return value. RVO is particularly notable for

being allowed to \nchange the observable behaviour of the resulting program by the C++ standard.

In general, the C++ standard allows a compiler to perform any optimization, provided the resulting executable exhibits the same observable \nbehaviour as if (i.e. pretending) all the requirements of the standard have been fulfilled. This is commonly referred \nto as the “as-if rule”. The term return value optimization refers to a special clause in the C++ standard \nthat goes even further than the “as-if” rule: an implementation may omit a copy operation resulting from a \nreturn statement, even if the copy constructor has side effects.

```
#include <iostream>

struct C {
  C() = default;
  C(const C&) {
    std::cout << "A copy was made." << std::endl;
  }
};

C f() {
  return C();
}

int main() {
  std::cout << "Hello World!" << std::endl;
  C obj = f();
}
```

When the compiler sees a variable in the calling function (that will be constructed from the return value), and a variable in the called function (that will be returned),\nit realizes it doesn’t need both variables. Under the covers, the compiler passes the address of the calling \nfunction’s variable to the called function.

To quote the C++98 standard, “Whenever a temporary class \nobject is copied using a copy constructor … an implementation is permitted to treat the original and the copy as two \ndifferent ways of referring to the same object and not perform a copy at all, even if the class copy constructor or \ndestructor have side effects. For a function with a class return type, if the expression in the return statement is \nthe name of a local object … an implementation is permitted to omit creating the temporary object to hold the

\nfunction return value…” (Section 12.8 [class.copy], paragraph 15 of the C++98 standard. The C++11 standard has \nsimilar language in section 12.8, paragraph 31, but it’s more complicated.)

References:

- Abseil.io
- Wikipedia: copy elision

Question 137: How can we ensure the compiler performs RVO?

Let’s remind us from yesterday about what RVO is!

It is an optimization done by the compiler that involves eliminating the temporary object created to hold a function’s return value. RVO is particularly notable for being allowed to change the observable behaviour of the resulting program by the C++ standard.

In practice, it means that when RVO is applied, fewer copies of an object will be made, the copy constructor will be invoked fewer times. The gain can be considerable for some big objects.

In order to have RVO applied the returned object has to be constructed on a return statement. A function can have multiple exits, it’s not a problem as long as the construction happens on the return statement.

```
1 SomeBigObject f() {
2   if (...) {
3     return SomeBigObject{...};
4   } else {
5     return SomeBigObject{...};
6   }
7 }
```

What is NRVO and when can it happen?

N in NRVO stands from Named, so we speak about Named Return Value Optimization. Copies of temporary objects can be spared even if the returned

object has a name and is therefore not constructed on the return statement. The requirement for an NRVO is to have one single object to return even if there are multiple exit points.

```
SomeBigObject f() {
  SomeBigObject result{...};
  if (...) {
    return result;
  }
  //...
  return result;
}
```

So NRVO won't happen here:

```
SomeBigObject f(...) {
  SomeBigObject object1{...}
  SomeBigObject object2{...};
  if (...) {
    return object1;
  } else {
    return object2;
  }
}
```

References:

- Abseil.io
- Fluent C++

Question 138: What are the primary and mixed value categories in C++?

A C++ expression - such as an operator with its operands, a literal, a variable name, etc. - is always characterized by two independent properties: a type and a value category. Each expression belongs to one of the three primary value categories:

- lvalue,
- prvalue,
- xvalue.

lvalue

An lvalue is an expression whose evaluation determines the identity of an object, bit-field, or function whose resources cannot be reused. lvalues could only appear on the left-hand side of an assignment operator.

The address of an lvalue may be taken by the built-in address-of operator. A modifiable lvalue may be used as the left-hand operand of the built-in assignment and compound assignment operators. An lvalue may be used to initialize an lvalue reference; this associates a new name with the object identified by the expression.

Some examples to give you the idea:

- the name of a variable, a function, member. Even if the variable's type is an rvalue reference, the expression consisting of its name is an lvalue expression;
- a function call or an overloaded operator expression, whose return type is lvalue reference
- all built-in assignment and compound assignment expressions (`a = b;`, `a*=b;`, etc)
- cast expressions to lvalue reference type
- …

prvalue

A prvalue ("pure" rvalue) is an expression whose evaluation either

- computes the value of the operand of an operator or is a void expression (such prvalue has no result object), or
- initializes an object or a bit-field (such prvalue is said to have a result object). With the exception of decltype, all class and array prvalues have a result object even if it is discarded. The result object may be a variable, an object created by new-expression, a temporary created by temporary materialization, or a member thereof;

A prvalue cannot be polymorphic: the dynamic type of the object it denotes is always the type of the expression. A non-class non-array prvalue cannot be cv-qualified. A prvalue cannot have an incomplete type. A prvalue cannot have abstract class type or an array thereof.

Some examples:

- literals, except for string literals that are lvalues
- a function call or an overloaded operator expression, whose return type is non-reference
- all built-in arithmetic and logical expressions
- lambda expressions
- …

xvalue

An "eXpiring" value is an expression whose evaluation determines the identity of an object, bit-field, or function whose resources can be reused. (Unlike lvalues whose resources cannot be reused.)

Some examples:

- a function call or an overloaded operator expression, whose return type is an rvalue reference to an object, such as `std::move(x);`
- a cast expression to rvalue reference to object type, such as `static_cast<char&&>(x)`
- …

Reference:

- C++ Reference: value categories

Question 139: Can you safely compare signed and unsigned integers?

No, you cannot. You should avoid comparing signed and unsigned integers in order to avoid bad results.

```cpp
int x = -3;
unsigned int y = 7;

// unsigned result, possibly 4294967286
std::cout << x - y << '\n';

// unsigned result: 4
std::cout << x + y << '\n';

// unsigned result, possibly 4294967275
std::cout << x * y << '\n';
std::cout << std::boolalpha;
std::cout << "-3 < 7: " << (x < y) << '\n'; // false
std::cout << "-3 <= 7: " << (x <= y) << '\n'; // false
std::cout << "-3 > 7:  " << (x > y) << '\n'; // true
std::cout << "-3 => 7: " << (x >= y) << '\n'; //true
```

The reason behind is that `x` which is an `int` is cast into an `unsigned int` Due to the conversion, `x` becomes - depending on your platform - 4294967293 (2^32 -3).

You should avoid comparing signed with unsigned integers unless you use C++20 and you have access to `std::cmp_equal` and to its friends. For more details, check out the references.

References:

- C++ Core Guidelines: ES.100: Don't mix signed and unsigned arithmetic
- Modernes C++: Safe Comparisons of Integrals with C++20

Question 140: What is the return value of main and what are the available signatures?

Though in some cases you might see `void`, it is not correct. The return type of the `main` function of a C++ program is `int`.

If the program finishes successfully `main` returns `0`, otherwise a non-zero number. Based on this return type the OS knows if the program succeeded or not. Though there is no standard on what different integer values would mean.

Even though the return type of `main` is `int`, in this case, a `return` can be omitted and it will be automatically treated as `0`, so it will be considered a successful return.

Main has two valid signatures:

- `int main()` where there is no arguments passed in
- `int main(int argc, char **argv)` or equivalent such as `int main(int argc, char *argv[])`

In this latter case, `argc` represents the number of command-line arguments passed into C++ and in `argv` you can find the program name and the arguments. When `argc` is bigger than 0, argv[0] will be the program name and the rest are the arguments.

As an example, if you execute: `./myProg foo bar 'b a z'`, `argc` will be 3 and `argv` will be `["myProg", "foo", "bar", "b a z"]`.

References:

- Core Guidelines: F46
- Stackoverflow: What should main() return in C and C++?
- CPlusPlus.com

Question 141: Should you prefer default arguments or overloading?

In most cases, default arguments should be preferred. There is no technical or performance reason behind it, it's more practical. While it's possible that one overload simply calls the other with the "default" as in the following example, but there is no guarantee that this will be respected and we won't end up with duplicated code, or worse with diverging behaviours when the intention was to have the same behaviour.

"default": values with overloading

```
1 int foo(int a, int b) {
2   //...
```

```
3 }
4
5 int foo(int a) {
6   return foo(a, 0); // 0 acts as default value for b
7 }
```

A real default argument:

```
1 int foo(int a, int b=0) {
2   // ...
3 }
```

Though if you are a library maintainer, you might have to take binary compatibility into consideration. Adding a new parameter to a function, even if it has a default argument, breaks binary compatibility. On the other hand, adding a new overload does not break binary compatibility.

Therefore if you are a library maintainer and binary compatibility is something you care about, prefer the overloads and take a note to merge them once you plan to release a new major version.

References:

- Core Guidelines: F51
- KDE Community Wiki: Policies/Binary Compatibility Issues With C++

Question 142: How many variables should you declare on a line?

One and only one. In most cases. Syntactically it's fine to declare just as many as you want, but declaring only one will increase readability and avoid mistakes.

Consider such a line:

```
1 char *p, c, a[7], *pp[7], **aa[10];
```

It's pretty difficult to know what goes on and easy to make mistakes. When you start adding initializations, it becomes more of a mess.

```
1 int a, b = 3;
```

How is `a` initialized?

It's not initialized, but inexperienced colleagues might think, that it should be 3.

If you initialize each variable on its own line, you won't have such a misunderstanding, plus it's much easier to add meaningful comments to the code if you want to explain the intention of the variable.

When I see multiple declarations on the same line, most often initialization comes line by line, variable by variable a few lines below, which is a complete waste of assignment.

Declare and if possible initialize one variable per line to spare some assignments and to boost readability.

Reference:

- Core Guidelines: ES10

Question 143: Should you prefer a switch statement or chained if statements?

Prefer the switch statement. There are multiple reasons.

A switch statement is more readable, even if you take into account all the necessary `break` statements.

Besides, usually, a `switch` statement can be better optimized.

But here comes the most important reason. It might be that you `switch` over an `int`, but most probably it can be turned into an `enum`. If you do so, and you avoid having a `default` case, the compiler will emit a warning in case you don't cover all the different cases in the `enum`.

This is also a good reason not to have a `default` case, just to get you covered. Imagine that one day, you add a new case to the `enum` and you forget to update all the switch statements in your codebase. If you don't use `default`s and you treat warnings as errors, your code simply won't compile.

```
enum class Color { Red, Green, Blue, Yellow };

// ...
Color c = getColor();
switch (c) {
  case Color::Red: break;
  case Color::Green: break;
};
/*
main.cpp:9:12: warning: enumeration value 'Blue' not hand\
led in switch [-Wswitch]
main.cpp:9:12: warning: enumeration value 'Yellow' not ha\
ndled in switch [-Wswitch]
*/
```

References:

- Core Guidelines: ES70
- Correct by Construction by Matt Godbolt

Question 144: What are include guards?

Include or header guards are there to prevent that the same header file is included multiple times. If a header is included in multiple files, then the same entities would be defined multiple times which is a clear violation of the one definition rule. As such, the code wouldn't compile.

In order to understand this better, we have to remind ourselves that before the compilation the preprocessor replaces all the `#include` statements with the textual copy of the included file.

Hence, we must use include guards in all our header files:

```
#ifndef SOME_UNIQUE_NAME_HERE
#define SOME_UNIQUE_NAME_HERE

// your declarations (and certain types of definitions) h\
ere
```

```
6
7 #endif
```

At the first inclusion as `SOME_UNIQUE_NAME_HERE` is not defined yet, it will be defined and the content of the header file will be copied. At the second inclusion, `#ifndef SOME_UNIQUE_NAME_HERE` will be evaluated to false and such the preprocessor jumps right after `#endif`, so at the end of the file.

With most modern compilers you can simply start your header file with `#pragma once` and avoid the above syntax to get the same results.

References:

- Core Guidelines: SF8
- LearnCpp: Header guards
- Wikipedia: include guard
- One Definition Rule

Question 145: Should you use angle brackets(<filename>) or double quotes("filename") to include?

As almost always in life, the answer is it depends!

For files that are in the same project, files that exist at the relative path to the including file, one should use the double-quotes.

For files from the standard library, or in fact, from any library that you depend on, the angle brackets form should be preferred.

```
1 // foo.cpp:
2 // From the standard library, requires the <> form
3 #include <string>
4
5 // A file that is not locally relative,
6 // included from another library; use the <> form
7 #include <some_library/common.h>
8
9 // A file locally relative to foo.cpp
10 // in the same project, use the "" form
```

```
11 #include "foo.h"
12
13
14 // A file locally relative to foo.cpp
15 // in the same project, use the "" form
16 #include "foo_utils/utils.h"
```

If you use double quotes, it will first look up the file in the local relative path first and then if it failed to file a match, it will look for it anywhere else it's possible.

This also means that if you use double quotes to include a file from another library and a file with the same name is created in your local project at the same relative path, you will have a bad surprise while compiling.

These are the commonly followed best practices, for exact information, it's worth checking your compilers implementation.

References:

- Core Guidelines: SF8
- Stackoverflow: What is the difference between #include <filename>and #include "filename"?</filename>

Question 146: How many return statements should you have in a function?

While some would insist that you should only have one return statement for better understandability, it's counter-productive.

It's true that with multiple return statements you'll have multiple exit points that you have to consider when you try to understand what a function does. But this is only a problem if your function is too big. If you write short functions, let's say what fits a screen (use a nice, big font size) then this is not a problem.

Besides, with multiple return statements, it's easy to create some guards on top of the function:

```
1 void foo(MyClass* iMyClass) {
2   if (iMyClass == nullptr) {
3     return;
4   }
5   // do stuff
6 }
```

With multiple return statements, you can avoid unnecessarily convoluted code and the introduction of extra state variables.

References:

- Core Guidelines: NR2