C++ INTERVIEW PREP

PRACTICE QUESTIONS WITH ANSWERS

- *What is function inlining? What are the benefits/costs?*
  - Function inlining is a process in which the compiler copy the entire body of a function into the body of the calling function. This has for benefit to, first, negate the cost of the function call thus saving a few instructions, but secondly (and more importantly), it allows the compiler to make additional optimizations that would not have been possible without having the full body of the function but just the function signature (e.g. of optimization could be loop unrolling, move loop invariant outside the loop, vectorization, etc). See this at 18:30.
  - Function inlining however can lead to an increase of the binary/program size. Inlining a function can also put a lot of (maybe unnecessary) pressure one the L1i cache. Moreover, inlining a function may also put undue pressure on the L2 cache - if an inlined function is used an unusually large number of times, the extra code size will increase the likelihood of cache misses, leading to long delays and pipeline stalls as the CPU twiddles its thumbs waiting for the memory bus to do something. See the answer to this thread.
  - It is also good to know that the inline keyword in C++ does not really enforce the compiler to do anything: if you set a function as inlined but the compiler believes it would be better not to inline it, then it will do whatever it believes to be the most performant solution.
  - So if you use inline wrongly (that would actually lead to slower code), the compiler will correctly adjust for that, but it would not do so for a macro.

- *When would inlining not be a good idea?*
  - Inlining is not always a good idea because it trades call overhead for larger code size, and larger code can hurt instruction-cache locality and increase latency on hot paths. In performance-critical code, especially in HFT-style loops where you care about tight, predictable instruction footprints, bloating a hot function with rarely used logic can be worse than paying a small call cost.
  - A classic bad case is when you inline a large logging function inside a branch that is almost never taken. Consider a hot function on your critical path:

```
inline void log_error(const Order& ord) {
    // Hypothetical heavy logging: formatting, time-stamping, locks,
```

```
etc.
    auto ts = std::chrono::high_resolution_clock::now();
    std::ostringstream oss;
    oss << "Order error: id=" << ord.id         << " price=" <<
ord.price         << " qty=" << ord.qty         << " ts=" <<
ts.time_since_epoch().count() << '\n';
    std::lock_guard<std::mutex> lock(global_log_mutex);
    global_log_stream << oss.str();
}

void handle_order(const Order& ord) {
// ultra-hot path
    if (LIKELY(is_valid(ord))) {
        match_order(ord);   // critical work
     }
    else {
        log_error(ord);            // almost never taken
        reject_order(ord);
    }
}
```

- If `log_error` is inlined into `handle_order`, all that logging machinery—string streams, time-keeping, mutex acquisition—turns into a large block of instructions embedded right inside a function you want to keep as small and tight as possible. Even though the error branch is rarely executed, the CPU still has to fetch, decode, and keep those instructions around in the L1 instruction cache and I-TLB as it steps through `handle_order`. That can evict useful instructions for `match_order` or the surrounding hot loop and increase i-cache misses and front-end pressure. In absolute terms, you might only add a few hundred bytes of code, but in a microsecond-sensitive loop this can be enough to degrade your p99/p999 latency.
- In this situation it is better to keep the logging function out-of-line, so the hot function's instruction footprint remains minimal. You might even force that with something like:

```
__attribute__((noinline))
void log_error(const Order& ord) {
    // same heavy logging body
}
void handle_order(const Order& ord) {
    if (LIKELY(is_valid(ord))) {
        match_order(ord);
    } else {
        log_error(ord);    // cold call, but hot code stays small
        reject_order(ord);
    }
}
```

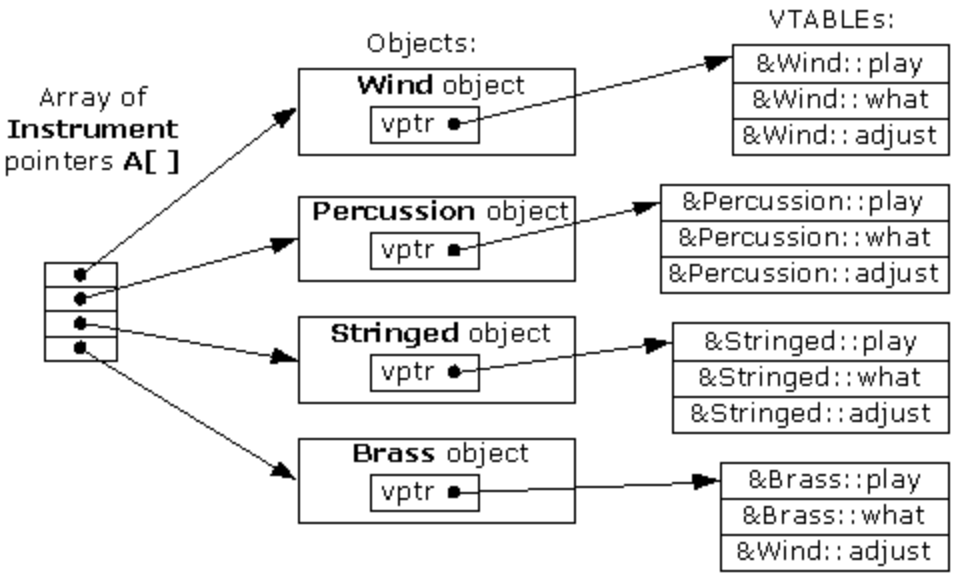
- Now the common path through `handle_order` consists mostly of the validation and matching logic, and the cold logging code is placed in a separate region of the binary. You pay a call/ret on the rare error path, but you protect your instruction cache and keep the hot path short and predictable. In other words, inlining is a bad idea when it causes large, cold, or complex code (like logging, error handling, formatting, or rarely used branches) to bloat a function that lives on a hot, latency-sensitive path.
- See this talk at ~31:50 for more info.

- *Difference between `inline` keyword and macros?*
  - Inline replaces a call to a function with the body of the function, however, inline is just a *request* to the compiler that could be ignored (you could still pass some flags to the compiler to force inline or use *always_inline* attribute with gcc).
  - A macro on the other hand, is expanded by the *preprocessor* before compilation, so it's just like text substitution, also macros are not type checked, inline functions are. There's a comparison in the wiki.

- *What is a `virtual` function? What are they used for?*
  - A **virtual function** in C++ is a member function of a class that you declare with the keyword `virtual` in a base class. This declaration tells the compiler that you intend for this function to be overridden in derived classes and that calls to this function should use dynamic dispatch—that is, the call is resolved at runtime based on the actual type of the object pointed to or referenced, not merely the static type of the pointer or reference.
  - Virtual functions are a cornerstone of runtime polymorphism, enabling you to design flexible and extensible class hierarchies. They allow derived classes to provide specialized implementations of functions that are declared in a common base class, so that when you invoke these functions through a pointer or reference to the base class, the correct, derived-class version is executed. This mechanism is essential for implementing interfaces and abstract base classes, and it is widely used in object-oriented design patterns, such as the Strategy, Observer, and Template Method patterns.
  - See this.

- *Why are virtual function slow (are they)?*

- The first cost of virtual functions is the indirection that comes from the vtable: each polymorphic class (a class that has virtual methods or that is part of an inheritance hierarchy where the base classes have virtual methods) will have its own vtable (and vpointer which increase the size of the class by the size of a pointer which is typically 8 bytes on 64bits systems, hidden) which will contain the different virtual function pointers, arranged by indexes (all vtables for each derived class have the same indexing: e.g. if function f() is at index 1 in the vtable derived class B, it will be at index 1 in the derived class C also. This process is actually quite fast since the compiler knows exactly at which index is each function in the vtable, and can use pointer arithmetic to quickly go to the wanted function (there is no iteration or anything similar going on). This indirection cost depends a lot on the size of the virtual function: the cost will be proportionally much higher for small virtual function that for very large functions.



- The real cost of virtual function actually comes from things that are not directly related to the virtual keyword in itself, but from indirect consequences:
  - Virtual functions cannot be inlined because of dynamic dispatch (the function call is resolved at runtime), thus losing any optimization that could come from inlining (see what are those benefits above).
  - If you have a vector of base class pointer objects that is unsorted (which can often happen), you risk having to consistently switch from one derived object function call (e.g. ABCABCABC), which will lead to a lot of cache misses due to cache eviction.
- See this and this.

- *What is the `explicit` keyword? What does it do?*
  - The `explicit` keyword prevents implicit conversion and copy-initialization (see this)

- *What are the 5 different types of casting in C++? What do they do?*
  1. `static_cast` (see this)
  2. `const_cast` (see this)
  3. `dynamic_cast` (see this)
  4. `reinterpret_cast` (see this)
  5. `std::bit_cast` (C++20) (see this
  6. C-style cast
     - See this for more info.
     - Know that when you use a C-style type, it actually goes through a chain of all the different cast available, from the safest to the most dangerous, until one (or none) works. This chain is defined by the standard and goes as follow:
       - **Attempt a `const_cast`:**
         First, the compiler checks to see if the conversion can be achieved solely by adding or removing cv-qualifiers (e.g., adding or removing `const` or `volatile`). If that works, the conversion is performed as a `const_cast`.
       - **Attempt a `static_cast`:**
         If a simple `const_cast` isn't enough, the cast is then treated as a `static_cast`. This covers many conversions including:
         - Conversions between related pointer types (upcasting/downcasting in an inheritance hierarchy, though downcasting isn't checked at runtime with `static_cast`),
         - Conversions between numeric types,
         - Conversions between enums and their underlying types, and so on.
       - **Combination of `static_cast` and `const_cast`:**
         If the desired conversion involves both a change in cv-qualification *and* a conversion that would normally require a `static_cast`, the C-style cast will perform a `static_cast` followed by a `const_cast` as needed.
       - **Attempt a `reinterpret_cast`:**
         If the conversion still isn't possible, then the C-style cast attempts a `reinterpret_cast`. This is used for low-level, implementation-defined casts (for example, converting between a pointer and an integer or between unrelated pointer types). Note that `reinterpret_cast` does not adjust cv-qualifiers, so if cv-qualification needs to be modified, a subsequent `const_cast` may also be applied.
       - **Combination of `reinterpret_cast` and `const_cast`:**
         Finally, if the conversion requires both a reinterpretation of the type and a change in cv-qualification, the cast may be implemented as a `reinterpret_cast` followed by a `const_cast`.

- For more info, see this

- *What does `static_cast`?*
  - `static_cast` is one of the four C++ cast operators that provide a way to perform explicit type conversions while maintaining type safety. It serves as a compile-time cast that allows you to convert pointers and references within an inheritance hierarchy, perform implicit conversions explicitly, and convert between related types. However, `static_cast` has specific boundaries regarding what it can and cannot do, ensuring that conversions are meaningful and safe within the context of the type system.
  - `static_cast` can be used for *upcasting* by safely converting a pointer or reference from a derived class to a base class. Since every derived class instance inherently contains a base class subobject, this conversion is guaranteed to be safe
  - `static_cast` can also be used for *downcasting*. However, as opposed to `dynamic_cast`, it does not perform runtime checks to ensure the validity of the conversion. Therefore, it should be used only when you are certain of the object's actual type to avoid undefined behavior.
  - You can use `static_cast` to add `const` to a type, such as converting a `T*` to a `const T*`. This is safe and maintains the immutability guarantees.

- *What does `const_cast` ? Why is it dangerous?*
  - `const_cast` allows the removal of the `const` and `volatile` qualifiers from an object. No other C++ cast is capable of removing it (not even `reinterpret_cast`). It is dangerous because it can lead to undefined behaviour. Moreover, removing the 'const' qualifier from an object breaks the previous promise that you made about this object (aka not modifying this object). It is important to note that modifying a formerly `const` value is only undefined if the original variable is `const`; if you use it to take the `const` off a reference to something that wasn't declared with `const`, it is safe. This can be useful when overloading member functions based on `const`, for instance. It can also be used to add `const` to an object, such as to call a member function overload.

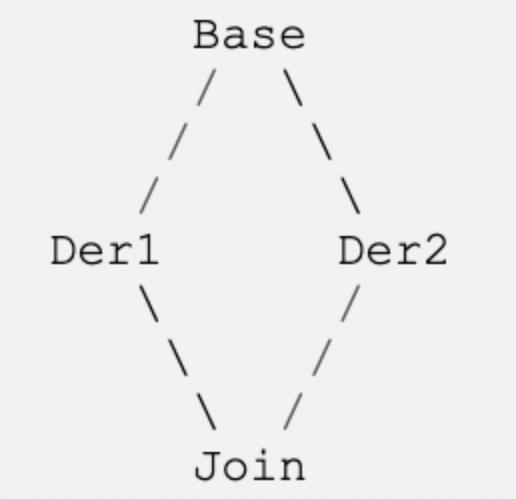
```
int main()
{
    const int a = 10;
    const int* p = &a;
    int* d = const_cast<int*>(p);
    *d = 15; // Undefined behaviour, trying to modify a const object
```

```
    int a2 = 10;
    const int* p2 = &a2;
    int* d2 = const_cast<int*>(p2);
    *d2 = 15; // OK, the original object pointed by p2 was not const
}
```

- *What does `dynamic_cast`? Is there any performance cost?*
  - `dynamic_cast` is a casting operator used for safe downcasting and crosscasting within class hierarchies. It allows you to convert a pointer or reference of a base class to a pointer or reference of a derived class at runtime, while ensuring type safety.
  - `dynamic_cast` can have some cost overhead because of a mechanism called Run-Time Type Identification (RTTI, see this) which allows the type of an object to be determined during program execution. Only three elements in cpp make use of RTTI (note that RTTI implementation is compiler-dependent):
    - `dynamic cast` operator
    - `typeid` operator
    - `type_info` class
  - Moreover, `dynamic cast` is a costlier operation because it can require the traversal of an inheritance hierarchy in order to check the object's type information to ensure that the cast is safe/valid. These type of checks are done at runtime which can increase the overhead.
  - **The Vtable and RTTI -** In many C++ implementations:
    - Each polymorphic class has a vtable which not only holds pointers to the virtual functions but also contains a pointer to a **typeinfo** structure or has a pointer to a structure that includes type information.
    - When you use `dynamic_cast` or `typeid`, the program accesses this typeinfo data in the vtable to determine the actual type of the object at runtime.
    - In gcc, RTTI can deactivated, thus saving some memory from the `typeinfo` struct, using the `-no-rtti` option. However, disabling RTTI kills `dynamic_cast` and `typeid` but has no impact on virtual functions. See this
  - Know that if a `dynamic_cast` fails, it will return either a `nullptr` or throw a `std::bad_cast`, depending whether the new type is a pointer (first case) or a reference (second case).

- *What is the dreaded diamond issue?*

- The “dreaded diamond” refers to a class structure in which a particular class appears more than once in a class’s inheritance hierarchy.
  `cpp class Base { public: // ... protected: int data_; }; class Der1 : public Base {// ...}; class Der2 : public Base {// ...}; class Join: public Der1, public Der2 { public: void method() { data_ = 1; // Bad, ambiguous; see below } }; int main() { Join* j = new Join(); Base* b = j; // Bad, ambiguous; see below }`



- The key is to realize that `Base` is inherited twice, which means any data members declared in `Base`, such as `data_` above, will appear twice within a `Join` object. This can create ambiguities: which `data_` did you want to change? For the same reason the conversion from `Join*` to `Base*`, or from `Join&` to `Base&`, is ambiguous: which `Base` class subobject did you want?
- C++ lets you resolve the ambiguities. For example, instead of saying `data_ = 1` you could say `Der2::data_ = 1`, or you could convert from `Join*` to a `Der1*` and then to a `Base*`. However please, Please, *PLEASE* think before you do that. That is almost always *not* the best solution. The best solution is typically to tell the C++ compiler that only one `Base` subobject should appear within a `Join` object. This is called **virtual inheritance** and that is described here.
- Note: Once a function is declared virtual in a base class, it's automatically virtual in all derived classes, so you don't have to repeat the keyword in the overriding function. However, many developers choose to include the `virtual` keyword (or better yet, the `override` specifier) in derived class functions for clarity and to clearly signal the intention that the function is meant to override a virtual function from the base class.

- *What is virtual inheritance? What problem does it solve?*
  - **Virtual inheritance** is a C++ technique that ensures only one copy of a base class's member variables are inherited by grandchild derived classes. Without virtual inheritance, if two classes `B` and `C` inherit from a class `A`, and a 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`.
- Note that in virtual inheritance, only the most derived class is responsible for creating the virtual base class.
- Below is an example that demonstrates virtual inheritance to address the "diamond problem." In this example, both `Derived1` and `Derived2` inherit from `Base` using virtual inheritance. Then, `MostDerived` inherits from both `Derived1` and `Derived2`, and thanks to virtual inheritance, there is only one shared instance of `Base`:

```
// Base class
class Base {
public:
    Base() { std::cout << "Base constructed\n"; }
    virtual void doSomething() { std::cout << "Base doing
something\n"; }
};

// Derived1 virtually inherits from Base
class Derived1 : virtual public Base {
public:
    Derived1() { std::cout << "Derived1 constructed\n"; }
};

// Derived2 virtually inherits from Base
class Derived2 : virtual public Base {
public:
    Derived2() { std::cout << "Derived2 constructed\n"; }
};

// MostDerived inherits from both Derived1 and Derived2
class MostDerived : public Derived1, public Derived2 {
public:
    MostDerived() { std::cout << "MostDerived constructed\n"; }
};

int main() {
    MostDerived obj;
    obj.doSomething();  // Calls Base::doSomething()
    return 0;
}
```

- **Virtual Inheritance:**
  When `Derived1` and `Derived2` inherit from `Base` using the `virtual` keyword, they

indicate that any further derived classes (like `MostDerived`) should share a single instance of `Base`. This helps avoid ambiguity and redundancy that arises in a diamond-shaped inheritance hierarchy.

- **Construction Order:**
  When `MostDerived` is instantiated, the `Base` constructor is called only once, even though `MostDerived` indirectly inherits from `Base` through both `Derived1` and `Derived2`.
- **Function Call:**
  The call to `doSomething()` in `main()` unambiguously refers to the one `Base` object in `MostDerived`, which is a direct benefit of using virtual inheritance.
- More info here, and here.

- *Should we always use virtual inheritance?*
  - No. An idiom of the C++ standard says that you "should only pay for what you use".
  - Virtual inheritance causes troubles with object initialization and copying. In virtual inheritance, the most derived type is responsible for those operations and has to be familiar with all the details of the base class, leading to more convoluted dependencies between types. It can ultimately lead to unreadable and unmaintainable code.

- *What does `reinterpret_cast`? Why is it dangerous? When would you use it in practice?*
  - As opposed to `static_cast`, `reinterpret_cast` allows you to cast a pointer of type A to a pointer of type B, which can lead to memory corruption
  - It can be dangerous in the following case:

```
int main()
{
    char c;
    int* p = reinterpret_cast<int*>(&c);
    *p = 5;

    /* This code might lead to memory corruption and runtime crash
(but no compiler error), since 'c' originally points to a 1 byte
memory, but after casting, it now have access to 3 additional bytes
that might overlap with adjacent memory. Once the line *p = 5 is
executed, those 3 adjacent bytes will be overridden by the new
integer value, thus potentially leading to corrupted memory or
undefined behaviour */

    // int* p = static_cast<int*>(&c); Compile Error!
```

```
    return 0;
}
```

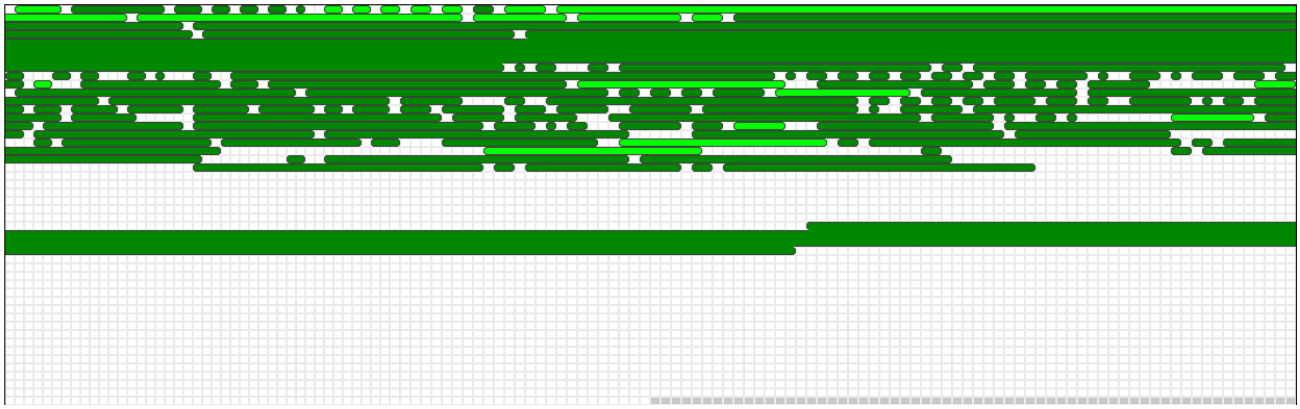
- `reinterpret_cast` is mostly used when dealing directly with bytes, see the following example:

```
struct myStruct
{
    int x;
    int y;
    char c;
    bool b;
};
int main()
{
    myStruct s;
    s.x = 5;
    s.y = 10;
    s.c = 'a';
    s.b = true;
    int *p = reinterpret_cast<int*>(&s);

    cout << *p << endl;  // prints 5
    p++;
    cout << *p << endl; // prints 10
    p++;
    cout << *p << endl; // prints garbage: the next 4 bytes in the
struct is not an int, but a char, a bool, and some padding
    char* c = reinterpret_cast<char*>(p);
    cout << *c << endl; // prints 'a'
    cout << *(reinterpret_cast<char*>(++c)) << endl; // prints 'SOH',
since the next byte in the struct is a bool with value 1 (or true),
which in a ASCII table is represented by the char 'SOH'
    cout << *(reinterpret_cast<bool*>(c)) << endl; // prints '1'

    return 0;
}
```

- *Why is malloc/free slow? Do you know what is memory fragmentation?*
    - One of the reasons why allocators are slow is that the allocation algorithm needs some time to find an available block of a given size. But that is not all. As time progresses it

gets harder and harder to find the block of the appropriate size. The reason for this is called *memory fragmentation*.

- In the above image, each block of memory is 16 bytes in size. If the program requires 32 bytes of memory (which must be contiguous), it will take more time since it has to traverse the entire block of memory until reaching the only available 32-bytes block at position 3 and 4. **As time passes, the *malloc* and *free* functions get slower and slower because the block of appropriate size is more difficult to find**. This problem is called **memory fragmentation**, and it can be better visualized in the following picture:



- **Memory fragmentation is a serious problem for long-running systems**.
- Moreover, malloc and free can be slow in a multithreaded context: In the case of multithreaded programs (and nowadays many programs are multithreaded), *malloc* and *free* must be thread-safe. The simplest way to make them thread-safe is to introduce mutexes to protect the critical section of those functions, but this comes with a cost. **Mutex locks and unlocks are expensive operations on multiprocessor systems, and even though the allocators can quickly find a memory block of appropriate size, synchronization eats away the speed advantage**.
- See this.

- *Is unsigned integer overflow defined or undefined behaviour? What about signed integers?*
  - Unsigned integers overflow is defined by basically restarting back to 0.
  - Signed integers overflow is undefined behaviour.

- *Can you move from const objects?*

- It is not possible to move away from a const value, which is quite logical.
- Know that a std::move is just a cast to a rvalue reference. When you use std::move on a const object, it actually casts to a const rvalue reference using `static_cast` (hence that is why the const qualifier is retained, since there is no `const_cast` taking place) . As a result, since const rvalue references do not bind to rvalue ref, the constructor being called will be the copy constructor (const rvalue ref bind to const lvalue ref), except if a const rvalue ref constructor is available (see this about const rvalue ref). As a result, the std::move actually decays to a copy.
- `static_cast<typename std::remove_reference<T>::type&&>(t)` Observe that we must call first `std::remove_reference` to ensure that we indeed get a rvalue ref in all cases. This is due to reference collapsing rules:
  1. `A& &` becomes `A&`
  2. `A& &&` becomes `A&`
  3. `A&& &` becomes `A&`
  4. `A&& &&` becomes `A&&`

- *What are weak pointers? What are they used for?*
  - A weak pointer is a type of smart pointer.
  - `std::weak_ptr` doesn't own the object it points to. This essentially means that a weak pointer doesn't increase the reference count of the object. The main purpose of weak pointers is to provide a way to access an object that might be owned by multiple shared pointers, without extending the lifetime of that object.
  - A first use of `std::weak_ptr` is to solve the dangling pointer problem. Using simple raw pointers, it is impossible to know whether or not a referenced data was deallocated. Instead, by letting a `std::shared_ptr` manage the data, and supplying `std::weak_ptr` to users of the data, the users can check validity of the data by calling `expired()` or `lock()`.
  - `std::weak_ptr` also solves cyclical references issues. See the following example

```
class B;

class A {
public:
    std::shared_ptr<B> b;
};

class B {
public:
    std::shared_ptr<A> a;
};
```

```
class C {
public:
    std::weak_ptr<A> a; // Use weak_ptr here
};

int main()
{
    std::shared_ptr<A> a = std::make_shared<A>();
    std::shared_ptr<B> b = std::make_shared<B>();
    std::shared_ptr<C> c = std::make_shared<C>();

    // In this first case, neither `a` nor `b` can be deleted because
their reference counts never reach zero.
    a->b = b;
    b->a = a;

    // Here, the reference count for `A` is only incremented by the
`shared_ptr` in `C`. When the `shared_ptr` for `C` goes out of scope,
`C` will be deleted, and the `weak_ptr` in `A` will become invalid.
This breaks the cycle, allowing `A` to be deleted as well.
    a->c = c;
    c->a = a;
}
```

- *What is the difference between private and public inheritance?*
  - **Public Inheritance**:
    - Models “is-a.”
    - Public base members → public in derived.
    - Protected base members → protected in derived.
  - **Protected Inheritance**:
    - Public base members → protected in derived.
    - Protected base members → protected in derived.
    - Usually hides the base class interface from the outside world but allows derived classes of `Derived` to see it.
  - **Private Inheritance**:
    - Used for “implementation” relationships.
    - Public & protected base members → private in derived.
    - You cannot use `Derived` as a `Base` from the outside (no implicit upcasting).
  - Private member stay private in all 3 cases.

- By default, classes use private inheritance while struct use public inheritance.
- Understand well what private inheritance does: if a method is public in the base class and is inherited privately in the derived class, then you can still call that method inside the child class functions. However, since that inherited method is now a private method inside the child class, then you can no longer access this function from OUTSIDE the child class.

- *What comprises a function signature?*
  - A function signature consist of a function name, a parameter count, and the parameters themselves.
  - Note that the return type (nor the const qualifiers) is not part of a function signature. This makes sense since when doing function overloading, you can only overload base on different parameters, not different return types.

- *What is a Covariant Return Type?*
  - In C++, to override a virtual function, the derived class's function must match the base class's signature exactly (e.g. if the base class virtual function returns an `int`, the override function in the derived class MUST also return an `int`)—*except* for certain allowed covariant return types (explained below).
  - The Covariant Return Type of a member function allows an overriding member function to return a narrower type (a.k.a something more specialized in the inheritance hierarchy). See this.

```
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{};
    }
};

int main () {
    SUVFactoryLine sf;
    SUV* car = sf.produce();
}
```

- Basically, in the above example since the `produce()` virtual function in the `CarFactoryLine` class returns a `Car*`, we would expect the same function overridden in the `SUVFactoryLine` class to also return a `Car*` (which can be done using upcasting since the object returned inside the function is a SUV object). However, thanks to covariant return types, we are actually able to return a `SUV*`, which is a derived type of `Car`.

- *What is the size of a class that has not member data (and no virtual function)? What if it has only 1 virtual function?*
  - In C++, every distinct object must occupy a unique address in memory. Even if a class has no data members (i.e., it's "empty"), the compiler still needs to ensure each object has its own address. The simplest way to guarantee that is to give each object **at least 1 byte** of size.
  - If the size were 0, then two different objects of that class type would have the **same address**, which violates the C++ rule that distinct objects must have distinct addresses. By making the size 1, each object has a unique address, and pointer arithmetic works consistently (for example, iterating over an array of such objects).
  - If a class has only virtual functions, then an instance of that class will be either 4 or 8 bytes (depending on the system architecture), the size of the vpointer. Note that the vtable itself is not counted as the size of an object since it is static to the class.
  - Note that as of C++20, there is now a `[[# no_unique_address]]` attribute that allows the compiler to optimize an empty class/struct to occupy no space in memory. See this for more info. Should also know about **Empty Base Class Optimization (ECBO)**.

- *What is the Empty Base Class Optimization?*
  - **Empty Base Class Optimization (EBCO)** is a compiler optimization technique in C++ that efficiently handles inheritance from classes that have no data members, known as empty base classes. Normally, even an empty class would occupy some space in a

derived class to ensure each subobject has a unique address, potentially increasing the size of the derived object unnecessarily. EBCO allows the compiler to eliminate this redundant space by reusing the memory layout of the derived class for the empty base class, effectively making the derived object occupy the same amount of memory as if it did not inherit from the empty class. This optimization not only reduces memory overhead but also enhances performance, especially in scenarios involving multiple inheritances or when using design patterns like the Curiously Recurring Template Pattern (CRTP). By minimizing the memory footprint of objects, EBCO contributes to more efficient and scalable software design.

- See this, this, this and this for more info.

```
struct Empty {}; // Empty could also contain only methods and EBCO would still apply since they do not participate to the size of a class/struct.

struct Derived1 : Empty
{
    char a;
};

struct Derived2 : Empty
{
    Empty e;
    char a;
};

int main()
{
    std::cout << sizeof(Empty) << std::endl; // Returns 1
    std::cout << sizeof(Derived1) << std::endl; // Returns 1, EBCO applies
    std::cout << sizeof(Derived2) << std::endl; // Returns 3, EBCO does not apply since Base object is used as a data member of Derived2. We have 2 bytes for object `a` and `e`, and 1 byte for the inherited byte of the Empty class.
}
```

- Click this link to run the code.

- *Why do we need virtual destructors?*

- A **virtual destructor** ensures that when you delete an object through a pointer to a **base class**, the **derived class's** destructor (and all intermediate destructors in the hierarchy) is called correctly.

- *What is the `final` keyword used for?*
  - In C++, marking a virtual function with `final` **prevents any further overriding** of that function in derived classes. It means:
    - The function is still virtual (i.e., supports dynamic dispatch in the current class).
    - No subclasses can override it again.
  - If the `final` keyword is used after a class definition, it simply means that this class cannot be derived from.
  - Note: Marking a virtual function (or the entire class) as `final` tells the compiler "no further overrides are possible." This can enable certain optimizations such as **devirtualization** or **inlining**. For example, if the compiler sees that a function is `final` and knows the *static type* of the object at compile time, it can sometimes skip the vtable dispatch and call the function directly. In practice, it's usually more about **expressing design intent** ("this function must not be overridden") than about raw performance.

- *What is the size of a `std::vector` (using `sizeof` operator)?*
  - The `sizeof` operator returns the size of the member variables of an object. An `std::vector` contains 3 data members:
    1. A pointer to the beginning of the vector: 8 bytes (on 64bits system)
    2. A `std::size_t` variable used for the number of elements in the vector: 8 bytes
    3. A `std::size_t` variable for the capacity: 8 bytes
  - Hence, the size of an `std::vector` is 24 bytes on a 64 bits system.

- *How does `delete[]` "knows" the size of the operand array?*
  - When memory is allocated on the heap (for example using new), the allocator will keep track of how much memory has been allocated. This is usually stored in a "head" segment just before the memory that get allocated. That way when it's time to free the memory, the de-allocator knows exactly how much memory to free.

- *What is `std::size_t`?*
  - **`std::size_t`** is a type defined in the C++ standard library (technically, it comes from the C standard library) that is:
    1. An **unsigned integer type**.
    2. **Large enough to represent the size** of the largest possible built-in object in memory (on the target platform).
    3. Used primarily for **array indexing**, **memory sizing**, and related scenarios.

- *What is RAII?*
  - **RAII** stands for *Resource Acquisition Is Initialization*. It is a programming idiom in C++ (and other languages) that ensures resources are properly managed by tying their lifetime to the lifetime of an object.

- *Can you throw exceptions from a destructor?*
  - Yes, while technically allowed, it is considered bad practice as it can lead to undefined behaviour, leading to resource leaks or denial-of-service attacks.

  1. **Stack Unwinding Conflict**
     - When an exception is thrown, the runtime starts *unwinding* the stack, calling the destructors of objects with automatic storage duration. If one of these destructors itself throws an exception, you then have *two* active exceptions simultaneously (the original one plus the destructor's).
     - By the C++ standard, this situation immediately calls `std::terminate()`, ending the program. This makes it impossible to handle the second exception normally and often results in a crash or forced termination.
  2. **Difficult Error Handling**
     - Even if there is no other exception in flight, handling an exception from a destructor is hard. The destructor is meant to release resources, and in the event that releasing those resources fails, you typically have limited options to recover safely. If you need to signal errors, it's usually better to do so outside the destructor (e.g., via an explicit `close()` or `cleanup()` method) rather than via exceptions in the destructor.
  3. **Violates RAII Semantics**
     - One of the pillars of modern C++ design is RAII (Resource Acquisition Is Initialization). Under RAII, resource release happens in destructors, which *must not* fail. Throwing from a destructor makes it impossible to rely on deterministic cleanup if an exception is thrown elsewhere. It also complicates the logic of

cleaning up resources since the cleanup method (the destructor) may never complete if it throws.

- **Recommended Approaches**
  - **Never Let the Exception Escape**: If some error occurs inside a destructor, swallow (catch) the exception or handle it in a way that does not rethrow. You might log errors, set flags, or defer handling to non-throwing mechanisms.
  - **Use Separate Error-Throwing Methods**: If there's a meaningful failure mode for closing or releasing a resource, provide an explicit function (e.g., `close()`, `finish()`, or `release()`) that can throw on errors. This way, the destructor can remain `noexcept`, and the user of your class can call the method at an appropriate time (and handle any exceptions that arise).
  - **Ensure `noexcept` Destructors**: In modern C++ (C++11 and later), destructors are implicitly `noexcept` unless declared otherwise. Keeping destructors `noexcept`-compatible avoids surprises during stack unwinding.

- *What is stack unwinding?*
  - As objects are created statically (on the stack as opposed to allocating them in the heap memory) and perform function calls, they are "stacked up".
  - When a scope (anything delimited by `{` and `}`) is exited (by using `return XXX;`, reaching the end of the scope or throwing an exception) everything within that scope is destroyed (destructors are called for everything). **This process of destroying local objects and calling destructors is called stack unwinding.**
  - You have the following issues related to stack unwinding:
    1. avoiding memory leaks (anything dynamically allocated that is not managed by a local object and cleaned up in the destructor will be leaked) - see RAII referred to by Nikolai, and the documentation for boost::scoped_ptr or this example of using boost::mutex::scoped_lock.
    2. program consistency: the C++ specifications state that you should never throw an exception before any existing exception has been handled. This means that **the stack unwinding process should never throw an exception** (either use only code guaranteed not to throw in destructors, or surround everything in destructors with `try {` and `} catch(...) {}`).
  - If any destructor throws an exception during stack unwinding you end up in the *land of undefined behavior* which could cause your program to terminate unexpectedly (most common behavior) or the universe to end (theoretically possible but has not been observed in practice yet).

- *What are the different type of storage duration?*

  Summary Comparison

Storage Duration	Lifetime	Allocation Location	Deallocation	Typical Use Cases
Automatic	Block-scope; ends when block exits	Stack	Automatic (scope exit)	Local variables, temporary objects
Static	Program lifetime	Fixed storage (global area)	Automatic at program exit	Global variables, function-static caches, constants
Dynamic	From allocation until explicitly deallocated	Heap	Manual (`delete` / `free`)	Objects with variable lifetime, large data buffers
Thread Local	Lifetime of the thread	Thread-specific storage	Automatic when thread ends	Per-thread caches, thread-specific counters

  - Note that any static object is stored directly in the binary (in a read-only section of memory).

- Why would you use `{}`-initializer in place of `()`?
  - `{}` prevents narrowing conversion.
  - `()` can lead to parsing issues (a.k.a the most-vexing parse) where the declaration of an object could be misinterpreted with a function signature (e.g. `int i()`)
  - See this for more info.

- *What are the four exception safety guarantee categories in C++?*
  1. *No exception guarantee* — If the function throws an exception, the program may not be in a valid state: resource leaks, memory corruption, or other invariant-destroying errors may have occurred.
  2. *Basic exception guarantee* — If the function throws an exception, the program is in a valid state (but it's unsure what this state will look like). No resources are leaked, and all objects' invariants are intact. This is the state expected by the standard library for all user-defined types.
  3. *Strong exception guarantee* — If the function throws an exception, the state of the program is rolled back to the state just before the function call (for example, std::vector::push_back).

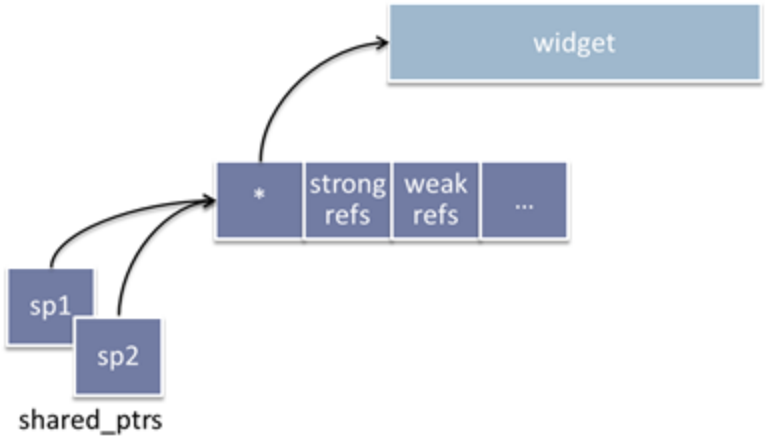
4. *Nothrow (or nofail) exception guarantee* — the function never throws exceptions. Nothrow (errors are reported by other means or concealed) is expected of destructors and other functions that may be called during stack unwinding. The destructors are `noexcept` by default.(since C++11) Nofail (the function always succeeds) is expected of swaps, move constructors, and other functions used by those that provide strong exception guarantee.

- *What are object invariants?*
  - In object-oriented programming, an **object invariant** is a set of conditions or assertions that must always be true about an object throughout its lifetime, ensuring that the object maintains a consistent and valid state, regardless of the operations performed on it; essentially, it defines the rules that govern how an object's data can be structured and accessed, preventing potential data corruption issues.
  - Consider a "BankAccount" class with fields like "balance" and "accountNumber". An invariant could be that the "balance" value is always non-negative. This means that any method manipulating the balance (like "withdraw") should ensure the balance never becomes negative.

- *What is `nullptr`? Why would you use it instead of `NULL`?*
  - `nullptr` is a keyword introduced in C++11 that represents a null pointer constant with its own type, std::nullptr_t. It is type-safe and unambiguously converts to pointer types and pointer-to-member types, ensuring clear function overload resolution. This resolves many issues associated with the traditional `NULL` macro, which is typically defined as the integer literal `0` and can lead to ambiguities, especially in cases where function overloads accept both pointer and integer types. Remember that 0 is a **null pointer constant**.
  - In contrast, `NULL` is an older, pre-C++11 macro that represents a null pointer by using an integer literal, often causing unintentional conversions and ambiguities in the code. Using `nullptr` is recommended for modern C++ code since it provides more clarity and safety in pointer contexts, particularly when dealing with overloaded functions and template programming, where type precision is essential.

- *What is a `std::shared_ptr`? How does it work?*

- *What are the advantages/disadvantages of* `std::make_shared`*?*
  - Using make_shared is more efficient. The shared_ptr implementation has to maintain housekeeping information in a control block shared by all shared_ptrs and weak_ptrs referring to a given object. In particular, that housekeeping information has to include not just one but two reference counts:
    - A “strong reference” count to track the number of shared_ptrs currently keeping the object alive. The shared object is destroyed (and possibly deallocated) when the last strong reference goes away.
    - A “weak reference” count to track the number of weak_ptrs currently observing the object. The shared housekeeping control block is destroyed and deallocated (and the shared object is deallocated if it was not already) when the last weak reference goes away.
  - If you allocate the object separately via a raw new expression, then pass it to a shared_ptr, the shared_ptr implementation has no alternative but to allocate the control block separately, leading to 2 allocations instead of 1, as shown here:



  - Having the control block and the pointee in a single chunk of memory improves locality. The reference counts are frequently used with the object, and for small objects are likely to be on the same cache line, which improves cache performance
  - Having a single block of memory for both the control block and pointee can also be seen as a disadvantage: the pointee could be kept alive by a weak pointer, even though the strong reference count is at 0, thus wasting memory.
  - `std::make_shared` does not allow you to have a custom deleter.
  - Note: If you need to create an object using a custom allocator, which is rare, you can use [allocate_shared] (https://en.cppreference.com/w/cpp/memory/shared_ptr/allocate_shared.
  - See this and this.

- *What are the advantages of* `std::make_unique`*?*

- It is better for exception handling, especially when calling it in function arguments. See this.

- *What is the control block that comes with smart pointers (shared, and weak)? When is it deleted?*
  - See this.

- *Are `constexpr` function inline by default?*
  - Yes, in C++ every `constexpr` function is implicitly declared as `inline`. This means that:
    1. **Inline for ODR (One Definition Rule)**:
       Being implicitly `inline` allows you to define the same `constexpr` function in different translation units without violating the One Definition Rule (ODR). Multiple definitions are permitted as long as they are identical.
    2. **Compiler Inlining Optimizations**:
       Although `constexpr` functions are marked as `inline` in the sense described above, this doesn't guarantee that the compiler will expand them inline (i.e., replace the function call with the function body) during optimization. The decision to actually inline the function is still up to the compiler's optimization heuristics.
  - So, while `constexpr` functions are implicitly inline (in the linkage sense), whether they are inlined for performance purposes depends on the compiler and optimization settings.

- *What are the differences between pointers and references?*
  - **A pointer** is a variable that holds the memory address of another variable. Pointers can be reassigned to refer to different objects, can be null (or uninitialized), and require explicit dereferencing to access the value they point to. They also support pointer arithmetic, allowing you to traverse arrays or other contiguous memory areas, albeit with potential pitfalls like segmentation faults if misused.
  - In contrast, **a reference** acts as an alias to an existing variable, meaning it must be initialized to refer to an object at the time of declaration and cannot later be made to refer to a different object. References cannot be null, and the compiler automatically treats them as the object they alias, eliminating the need for explicit dereferencing. This makes references a safer and often more convenient option for parameter passing and aliasing, but their rigid binding means they lack the flexibility of pointers in

some scenarios, such as representing optional values or performing dynamic reassignments.
- Note that references most of the time (not always) are actually not taking any additional space in memory, it does not have an address (or its address is the same as the referent). The 'not always' is important; in some implementations, references are implemented as **tagged memory address**, acting like a pointer that is automatically dereferenced. See this for more info.

- *What is the `noexcept` keyword? Can an exception be thrown in a `noexcept` function?*
  - The `noexcept` specifier in C++ is used to indicate that a function is guaranteed not to throw any exceptions. This can be applied to both function declarations and definitions, and it plays an important role in both optimization and exception safety:
  - **Optimization and Code Generation:**
    When a function is marked with `noexcept`, the compiler knows that it won't throw exceptions. This information can help the compiler generate more optimized code, especially for move operations and exception handling scenarios. For example, the standard library containers often use move operations that require `noexcept` to ensure strong exception guarantees when reallocating memory.
  - **Exception Safety and Clarity:**
    Declaring a function as `noexcept` communicates the programmer's intent that this function is not expected to throw, which can simplify exception safety analysis. A `noexcept` function can throw an exception, but it must handle this exception within the function scope. If the exception propagate outside the function scope then the program will call `std::terminate()`. This helps in identifying and controlling error-handling behavior at critical points in your code.

- *Can you explain what devirtualization means?*
  - Devirtualization is a compiler optimization that eliminates the overhead associated with virtual function calls by resolving them at compile time rather than at run time. When the compiler can determine the exact type of an object or the most specific override of a virtual function, it can replace the indirect call (using the virtual table) with a direct call to the function. This not only improves performance by removing the extra layer of indirection but also opens the door to further optimizations such as inlining, making the code more efficient while maintaining object-oriented polymorphism.
  - See this for more info.

- *When does the compiler will favor a copy constructor instead of a move constructor?*
  - Compiler will always choose (whenever possible) a move constructor if this one is declared as `noexcept` (there is actually a type trait called `std::move_if_noexcept`). This choice comes from the following fact: the core issue is that it's impossible to offer strong exception safety with a throwing move constructor. Let's imagine that if, in a `std::vector` resize, half way through moving the elements to the new buffer, a move constructor throws. How could you possibly restore the previous state? You can't use the move constructor again because, well, that could just keep throwing.
  - Copying works for strong exception safety guarantee regardless of it's throwing nature because the original state is not damaged, so if you can't construct the whole new state, you can just clean up the partially-built state and then you're done, because the old state is still here waiting for you. Move constructors don't offer this safety net.
  - It's fundamentally *impossible* to offer strongly exception safe `resize()` with a throwing move, but *easy* with a throwing copy. This fundamental fact is reflected everywhere over the Standard library. See this thread and this blog.

- *Does vector requires object to be default constructible?*
  - No, a `std::vector` does not require its element type to be default-constructible in all cases. You can create an empty vector, and operations like `push_back` or `emplace_back` don't require default construction of objects; they directly construct elements with given arguments. However, certain operations—such as calling `resize(n)` to increase the vector's size without providing explicit values—do require the element type to be default constructible, because the vector needs to default-initialize new elements. Same with the `std::vector` constructor that takes an unsigned integer as a parameter, which represents the number of objects that will be added to the vector at initialization (and thus requires default-constructible objects to fill the vector with).

- *What is a particularity of the template specialization `std::vector<bool>`?*
  - `std::vector<bool>` is a specialized template instantiation of `std::vector` that doesn't actually store each element as a separate `bool`, but rather packs the booleans into bits to optimize for space. This compact representation means that, unlike other standard containers, it doesn't provide genuine references to its boolean elements; instead, it uses proxy objects to simulate references, which can lead to some unexpected behavior and performance considerations.

- *What is `std::variant`?*
  - `std::variant` is a type-safe union introduced in C++17 that can hold a value from a set of specified types, but only one at a time. It provides a way to work with heterogeneously-typed data in a safe manner by ensuring that only one of the alternative types is active, and it enforces type correctness at compile time. With functions like `std::visit` and helper utilities such as `std::get` and `std::get_if`, `std::variant` makes it easier to perform operations on the stored value depending on its active type. This feature is especially useful for scenarios where an object might need to store one of several types, similar to a tagged union in other programming languages, while providing the guarantees and safety of modern C++ type systems.
  - However, it is important to note that as opposed to a regular `union`, a std::variant does not have the same size as the largest object in its definition. Instead, its size is the sum of the size of all member data that composes one of its object.
  - It is similar to what `std::any` would do, with some caveats. See this and this for more info.

- *What is `std::optional`?*
  - `std::optional` is a utility template introduced in C++17 that encapsulates an optional value: a value that may or may not be present. It serves as a safer alternative to using pointers or special sentinel values to represent missing data, allowing you to explicitly indicate that a value is optional and require callers to check its status before use. By wrapping a value of a given type, `std::optional` makes it clear when an operation might fail to return a valid result, thereby helping to write more robust and self-documenting code. See this for more info.

- *What are your favorite features from the C++XX standard?*
  - **C++11:**
    - Introduced *auto* type deduction, range-based for loops, and lambda expressions for more concise and flexible code.
    - Added *rvalue references* and *move semantics* to optimize resource management.
    - Provided the *smart pointers* (`std::unique_ptr`, `std::shared_ptr`) for safer memory management.
    - Included *nullptr* for type-safe null pointer representation.
    - Brought in *constexpr* for compile-time evaluation and *static_assert* for compile-time assertions.

    - Enhanced multithreading support with the *thread library* and synchronization primitives.
- **C++14:**
    - Relaxed some of the restrictions of *constexpr* functions by allowing multiple statements and local variables.
    - Introduced *generic lambdas*, enabling lambda expressions to capture parameters with auto type deduction.
    - Improved type deduction in return types and variable declarations, making code more concise.
    - Provided several small language and library improvements for better usability and consistency.
- **C++17:**
    - Added *if constexpr* for compile-time conditional branching.
    - Introduced *structured bindings* to easily unpack tuples, pairs, and structs.
    - Supported inline variables to allow defining variables in header files without violating the ODR.
    - Expanded the standard library with utilities like *std::optional*, *std::variant*, and *std::any* for better type safety and flexibility.
    - Included *std::string_view* for non-owning, efficient string handling and added parallel algorithms to the STL.
- **C++20:**
    - Introduced *concepts* to constrain template parameters and improve template error messages.
    - Added *ranges*, which provide a new way to work with sequences of data in a composable and lazy manner.
    - Brought in *coroutines* to facilitate writing asynchronous code with a sequential style.
    - Enhanced *constexpr* capabilities further, allowing more complex computations at compile-time.
    - Provided the *modules* feature for improved compile times and better code organization compared to traditional headers.
    - Introduced *calendar and timezone* utilities in the standard library.
- **C++23 (Upcoming/Recent):**
    - Continues refining language and library features such as additional improvements to *constexpr* and multithreading.
    - Adds new library components and algorithms aimed at making code safer, more expressive, and easier to maintain.

- Enhances the usability of existing features with smaller, incremental improvements across the board.

- What is 'SSO' in C++?
  - SSO refers to **Small String Optimization**, which is a mechanism used in the `std::string` implementation to store small strings directly within the string object itself, rather than allocating memory on the heap.
  - A standard `std::string` object typically contains a pointer to a dynamically allocated buffer, where the actual string data is stored. This buffer can grow or shrink as needed to accommodate different string lengths, which involves heap allocations and deallocations that can be costly in terms of performance.
  - With SSO, the `std::string` object includes a small internal buffer, usually within the string object itself, to store short strings. If the string length exceeds the capacity of this internal buffer, the string falls back to dynamic allocation. Here's a simplified illustration:

- *Is `char` signed or unsigned by default?*
  - The sign of a plain `char` is implementation-defined in C++. This means that on some systems or compilers, `char` behaves as a signed type, while on others it behaves as an unsigned type. This differs from types like `signed char` and `unsigned char`, which explicitly specify their signedness. If you need a specific behavior, it's best to choose one of those explicit types to avoid portability issues.

- *What is the expected output in the following code snippet?*

```
int f(int a) { return a + 7; }
int g(int b) { return b * 7; }
int h(int c) { return c/2; }
int sum(int a, int b) { return a * b + b; }

int main()
{
    int a = 2;
    int b = 6;

    int sum = sum(f(a), g(h(b)));
```

```
    std::cout << sum << std::endl;
}
```

- Answer: This is unspecified/implementation dependent. As per the standard: "Order of evaluation of any part of any expression, including order of evaluation of function arguments is *unspecified* (with some exceptions, see this). The compiler can evaluate operands and other subexpressions in any order, and may choose another order when the same expression is evaluated again.".
- There is no concept of left-to-right or right-to-left evaluation in C++. This is not to be confused with left-to-right and right-to-left associativity of operators: the expression a() + b() + c() is parsed as (a() + b()) + c() due to left-to-right associativity of operator+, but c() may be evaluated first, last, or between a() or b() at run time.
- In the code snippet above, the order in which the arguments to `sum` (i.e., `f(a)` and `g(h(b))`) are evaluated is unspecified by the C++ standard. This means the compiler is free to evaluate `f(a)` first or `g(h(b))` first, and that might vary between compilers or optimization levels. However, once it decides to evaluate, say, the second argument `g(h(b))`, it will completely evaluate it—that is, it will first call `h(b)` and then pass its result to `g()`. Only after the complete evaluation of that argument will the compiler evaluate the other argument (`f(a)` in this case), if it chooses that order.

- *What is the difference between the stack and the heap?*
  - The **stack** is a memory region that operates in a LIFO (Last In, First Out) manner and is typically used for objects with **automatic storage duration**, such as function parameters and local variables. Allocation on the stack is fast and efficient due to its simple allocation scheme and minimal management overhead, resulting in a predictable memory layout. However, because the stack has a limited size, allocating large amounts of memory there can lead to overflow. Furthermore, stack memory is generally reserved for objects with a short lifetime, making it less flexible for long-lived or dynamically sized data.
  - In contrast, the **heap** is a larger, more flexible memory region used for **dynamic storage duration**, where memory is allocated and managed during runtime. Memory allocated on the heap can persist beyond the scope in which it was created and can be resized or deallocated manually using mechanisms such as `new` / `delete` in C++ (or `malloc` and `free` in C). While heap allocations provide greater flexibility, they incur more overhead in terms of allocation/deallocation time and require explicit memory management to prevent issues like memory leaks, fragmentation, or dangling pointers.

- *Why is it slow to allocate memory on the heap?*
  - Allocating memory on the heap is generally slower than stack allocation due to the additional overhead and complexity involved in managing dynamic memory. Unlike the stack, which operates on a simple Last-In-First-Out (LIFO) principle allowing for rapid allocation and deallocation by merely adjusting a pointer, the heap requires more intricate management to handle arbitrary allocation and deallocation requests. This involves searching for suitable memory blocks, maintaining metadata about allocated and free memory regions, and handling fragmentation. One common mechanism used to manage free memory on the heap is the **free list**, which is a linked list of available memory blocks. When a program requests memory, the allocator traverses the free list to find a block of sufficient size, which can be time-consuming, especially as the free list grows or becomes fragmented. Additionally, allocating and deallocating memory on the heap often involves locking mechanisms to ensure thread safety in multi-threaded environments, further contributing to the slower performance compared to the stack.
  - Moreover, heap allocation can lead to memory fragmentation, where free memory is broken into small, non-contiguous blocks, making it harder to satisfy large memory requests efficiently. This fragmentation necessitates more complex algorithms to manage and optimize memory usage, adding to the allocation time. In contrast, stack allocation benefits from its predictable and linear memory usage pattern, allowing for constant-time operations without the need for searching or managing free blocks. The reliance on free lists and the need to handle fragmentation and concurrency in heap memory management introduce significant overhead, making heap allocations inherently slower than the straightforward and efficient stack allocations.

- *Why do we need the mutable keyword in lambdas?*
  - In C++, lambda expressions are implicitly const by default, meaning that the body of the lambda cannot modify its captured variables (unless they are captured by reference). When you need to modify a captured variable by value within the lambda, you must use the `mutable` keyword. This keyword relaxes the default constness of the lambda's call operator, allowing you to change the state of captured variables. Without `mutable`, attempts to modify by-value captures would result in a compilation error because the lambda operator() is considered a const member function.
  - It may look counterintuitive that we are only able to modify captures-by-reference, but remember that lambdas are function objects and that `const` member functions treat the object (i.e., `*this`) as immutable, meaning that it prevent modification of the object's own data members through the interface of that object. For non-reference (or non-pointer) members, the data itself is part of the object, so a `const` function cannot modify them. However, when a member is a reference, it doesn’t actually hold the

object but rather refers (or aliases) another object. The `const` qualifier on the member function applies to the reference itself (i.e., you can't make it refer to a different object) but not to the object being referenced if that object is not inherently const. Therefore, inside a `const` method, while you cannot change which object the reference is bound to, you can modify the object it refers to if that object is mutable. This is why modifying the referent in a const function is allowed even though modifying a direct member variable is not.

- *What does `throw;` (throwing nothing) do?*
  - In C++, the statement `throw;` is known as a **rethrow**. When used inside a `catch` block, it rethrows the exception that was caught. This allows the exception to propagate further up the call stack, enabling other exception handlers to process it.
    - **Inside a `catch` block:** If you use `throw;` inside a `catch` block, it rethrows the currently caught exception. This is commonly used to log or partially handle an exception and then allow higher-level handlers to process it.
    - **Outside a `catch` block:** Using `throw;` outside of a `catch` block is undefined behavior (UB). Since there is no currently active exception to rethrow, the program may crash or exhibit unpredictable behavior.

- *What is the difference between `enum` and `enum class`?*
  - A traditional `enum` introduces its enumerator names into the surrounding scope, which can lead to name collisions if different enums have members with identical names. Additionally, the values of a traditional `enum` implicitly convert to integers, allowing enumerators from different enums or even unrelated integers to be compared or mixed inadvertently. This lack of encapsulation and type safety can result in bugs that are difficult to trace, especially in large codebases where multiple enums might overlap in their enumerator names or where unintended comparisons occur.
  - On the other hand, `enum class` (introduced in C++11) provides a scoped and strongly typed enumeration that enhances both safety and clarity. The enumerators defined within an `enum class` do not leak into the surrounding scope, preventing name clashes and making the code more readable. Moreover, `enum class` does not allow implicit conversions to integers, which enforces type safety by ensuring that enumerator values cannot be inadvertently compared with integers or with enumerators from different `enum class` types.

- *What happens if a thread is not joined at the end of `main()` but has time to finish its work before the main thread finishes? What if it still hasn't finished the work once `main()` ends?*
  - In C++, if a `std::thread` is not explicitly joined or detached before the end of the `main()` function, the program will call `std::terminate()`, leading to an abrupt termination. This behavior occurs regardless of whether the thread has already completed its work or is still running when `main()` ends. Specifically, even if the thread has finished executing its task before `main()` concludes, the `std::thread` object remains in a joinable state until you either call `join()` to wait for its completion or `detach()` to allow it to run independently. Failing to perform either action means that when the `std::thread` object is destructed as `main()` exits, the C++ runtime detects that the thread is still joinable and invokes `std::terminate()`, causing the program to terminate immediately.
  - On the other hand, if the thread is still actively working when `main()` ends and you have not joined or detached it, the same `std::terminate()` is triggered. This ensures that all threads are properly managed and that resources are correctly released, preventing potential issues like resource leaks or undefined behavior from orphaned threads. To avoid unexpected termination, it is essential to either join all threads to ensure they complete their execution before the program exits or detach them if their execution should continue independently of the main thread's lifecycle. Properly handling thread termination not only maintains program stability but also ensures that all necessary cleanup operations are performed.

- *Why is it impossible to construct a `std::vector<int&>`?*
  - The component type of containers like vectors must be assignable. References are not assignable (you can only initialize them once when they are declared, and you cannot make them reference something else later). Other non-assignable types are also not allowed as components of containers, e.g. `vector<const int>` is not allowed.

- *What is object slicing in C++?*
  - **Object Slicing** in C++ occurs when an object of a derived class is assigned to an object of a base class, resulting in the loss of the derived class-specific attributes and behaviors. This happens because the base class object can only hold the data members defined within the base class, effectively "slicing off" any additional members that the derived class may have. Object slicing undermines the principles of polymorphism, as the derived class's unique functionalities are no longer accessible through the base class object. This typically occurs when passing objects by value,

returning objects by value, or assigning derived class objects to base class variables without using pointers or references.

- Imagine you have a base class `Animal` and a derived class `Dog` that adds an extra member function `bark()`. If you create a `Dog` object and assign it to an `Animal` object, only the `Animal` part of the `Dog` object is copied, and the `bark()` function is lost in the `Animal` object. As a result, attempting to call `bark()` on the `Animal` object will either result in a compilation error or will not exhibit the intended behavior, demonstrating how object slicing removes the derived class's specific features.

```
class Animal {
public:
std::string name;
Animal(const std::string& n) : name(n) {}
void speak() const { std::cout << name << " makes a sound." <<
std::endl; }
};

class Dog : public Animal {
public:
Dog(const std::string& n) : Animal(n) {}
void speak() const { std::cout << name << " barks." << std::endl; }
void bark() const { std::cout << name << " says: Woof!" << std::endl;
}
};

int main() {
Dog myDog("Buddy");
Animal myAnimal = myDog; // Object slicing occurs here
myAnimal.speak(); // Outputs: Buddy makes a sound.
// myAnimal.bark(); // Error: 'class Animal' has no member named
'bark'
return 0;
}
```

- *What is struct padding? Why is it needed? What is a word?*
  - **Struct Padding** refers to the practice of adding unused bytes between the members of a `struct` (or `class`) in C++ to align data in memory according to the architecture's requirements. This alignment ensures that each data member is placed at a memory address that is a multiple of its size or a specified alignment boundary, which can vary depending on the system's architecture (e.g., 4-byte, 8-byte alignment). Padding helps optimize memory access speed and ensures that the CPU can read and write data

efficiently. Without padding, misaligned data could lead to performance penalties or even hardware exceptions on some architectures that strictly enforce alignment.

- In computer architecture, a **word** is the natural unit of data used by a particular processor design. The size of a word is typically equal to the processor's register size, such as 32 bits (4 bytes) or 64 bits (8 bytes) in modern systems. Words are fundamental for the CPU's operation, as they determine the amount of data the processor can handle in a single instruction. For example, a 32-bit word can represent integers, memory addresses, or other data types that fit within 32 bits. Understanding the concept of a word is crucial when dealing with memory alignment and struct padding, as padding often aligns data members to word boundaries to match the processor's word size, thereby facilitating efficient data access and manipulation.

```
struct Example {
char a; // 1 byte
char b; // 1 byte
int c; // 4 bytes
};
```

- In the above example, without padding the integer would overlap between two words (on a 32bits architecture) requiring two CPU cycles for access. Adding padding after the two `char a` and `char b` now removes the overlap, leaving `int c` on its own word in memory. As a result, `int c` can now being accessed in one CPU cycle instead of two. Note that adding padding does not change anything for `char a` and `char b`, both characters can be accessed in one CPU cycle in both cases. In fact, a single byte type can always be accessed in a single CPU cycle as it is always perfectly aligned.
- See this, this AND the link in description! See also this, this and this.

- *What is the difference between a cache line and a word?*
  - A **word** is the standard unit of data that a processor can handle efficiently, typically matching the processor's native register size. For instance, on a 32-bit system, a word is usually 32 bits (4 bytes), and on a 64-bit system, it is 64 bits (8 bytes). Words are the basic building blocks for data operations, such as arithmetic calculations, data manipulation, and memory addressing. They determine the amount of data the CPU can process in a single instruction cycle, directly impacting the system's performance and efficiency. Note that a **word** refers to the fixed size of data a CPU processes at once, while a **CPU register** is a small, high-speed storage location within the CPU that holds that data temporarily, meaning a register is essentially a place to store a word of data during processing; so, a word defines the data size, and a register is a storage location for that data size.

- On the other hand, a **cache line** (also known as a cache block) is a larger unit of data transfer between the main memory (RAM) and the CPU cache. A cache line typically consists of multiple words; for modern processors, a cache line is usually 64 bytes, which would encompass eight 8-byte words in a 64-bit system. The purpose of a cache line is to optimize memory access patterns by taking advantage of spatial locality—the tendency for programs to access memory locations that are near each other within a short time frame. When the CPU requests data, it doesn't just load the single word needed but an entire cache line, anticipating that nearby data will soon be required. This reduces the number of memory accesses needed, as multiple words are fetched in a single operation, thereby improving overall system performance.

- Is `std::shared_ptr` thread-safe?
  - The **control block** of `std::shared_ptr` is thread-safe. The critical thread-safe feature it provides is the **atomic manipulation of the reference count**. This means that multiple `std::shared_ptr` instances can safely share ownership of the same object across different threads without causing race conditions when copying, assigning, or destroying `shared_ptr` instances. The underlying control block, which holds the reference count and other metadata, uses atomic operations to ensure that increments and decrements to the count are performed safely even when accessed concurrently by multiple threads.
  - While the reference counting mechanism of `std::shared_ptr` is thread-safe, **the `std::shared_ptr` instances themselves are not inherently thread-safe for concurrent modifications**. If multiple threads attempt to modify the same `std::shared_ptr` instance—such as resetting it, assigning a new pointer, or swapping—it can lead to undefined behavior unless proper synchronization (e.g., using mutexes) is employed. Additionally, the object being pointed to by `std::shared_ptr` is **not protected**; concurrent access to the managed object must be synchronized externally if the object itself is mutable and accessed by multiple threads.
  - To see a possible implementation of `std::shared_ptr`, see this post and a good video here.

- *Does the following code snippet compiles?*

```
const int& add();

int main()
{
```

```
    auto v = add();
    static_assert(std::is_same_v<decltype(v), const int&>);
}
```

- No, it will lead to a static assertion failure. In C++, the `auto` keyword deduces the type of a variable by **ignoring top-level `const` qualifiers and references** unless explicitly specified otherwise. Here are some examples:

```
const int& add();
int* add2();
const int* const add3();

int main()
{
    decltype(auto) v = add();
    const auto& c = add();
    auto d = add2(); // Note that auto does not ignore pointers,
since a pointer is an actual type (as opposed to references)
    auto e = add3();

    static_assert(std::is_same_v<decltype(v), const int&>); // OK
    static_assert(std::is_same_v<decltype(c), const int&>); // OK
    static_assert(std::is_same_v<decltype(d),  int*>); // OK
    static_assert(std::is_same_v<decltype(e), const int*>); // OK
    static_assert(std::is_same_v<decltype(e), int*>); // NOT OK,
e is deduced as a pointer to const int, which is different than a
pointer to int
    static_assert(std::is_same_v<decltype(e), const int* const>);
// not OK, the const qualifier is dropped for the pointer (remember,
auto ignores the const qualifier to the type which is a pointer, not
the const to the pointed object, a pointer to an int is a different
type than a pointer to a const int)
}
```

- Know that `decltype(auto)` can deduces the **exact** type of an expression, and can thus deduce both the const qualifier and reference. It can be seen as a perfect forward type deduction.

- *What is the Static Initialization Order Fiasco?*
  - The *static initialization order fiasco* refers to the ambiguity in the order that objects with static storage duration in different translation units are initialized in. If an object in one translation unit relies on an object in another translation unit already being initialized, a crash can occur if the compiler decides to initialize them in the wrong order. For

example, the order in which .cpp files are specified on the command line may alter this order. The Construct on First Use Idiom can be used to avoid the static initialization order fiasco and ensure that all objects are initialized in the correct order. Within a single translation unit, the fiasco does not apply because the objects are initialized from top to bottom.

- *Are you able to modify a member data that is a reference to an object outside the class inside a const member function? Why or why not?*
  - When you declare a member function as `const`, you are promising that this function will not **modify the state of the object on which it is called**. This means that within a `const` member function, you cannot alter any of the class's non-`mutable` member variables directly. The `const` qualifier ensures that the `this` pointer inside the function is treated as a pointer to a `const` object (`const ClassName* const this`), preventing modifications to the object's members.
  - However, when a class has a member that is a reference (e.g., `int&`), the `const` qualifier on the member function does not inherently apply to the object being referenced. The reference itself is an alias to another object, and the `const`ness of the member function affects the alias (the reference) but not the referent (the object it points to). As a result, if the referenced object is not `const`, a `const` member function can modify it.
  - This is the reason why you need to set lambdas as 'mutable' if you want to modify an object that captures by value but not if it captures by reference.

- *What does the following code snippet outputs?*

```
struct A {
    virtual void f() { std::cout << "1"};
    A() { f(); }
};

struct B : A {
    virtual void f() override { std::cout << "2"; }
};

int main() {
    B b;
}
```

- When an object of class `B` is created, the constructor of the base class `A` is invoked first. Inside `A`'s constructor, the virtual function `f()` is called. However, during the base class construction, the object's dynamic type is still `A`, not `B`, so the call to `f()` resolves to `A::f()`, which prints `"1"`. Even though class `B` overrides `f()` to print `"2"`, this override isn't used during the base class's constructor execution. Consequently, the program only outputs `1`.
- To be more precise, when an object of type `B` is created (`B b;`), the following sequence occurs:
  - **Base Class Constructor (`A::A()`):**
    - The constructor of `A` is invoked first.
    - Inside `A`'s constructor, the virtual function `f()` is called.
    - **Important:** During the base class constructor execution, the dynamic type of the object is still `A`, not `B`. Therefore, the call to `f()` resolves to `A::f()`.
- Note: during object destruction, this sequence is somewhat reversed: the destructor of the derived class `B` is invoked first, handling any cleanup specific to `B`. After `B`'s destructor completes, the destructor of the base class `A` is called. This order ensures that resources allocated by the derived class are properly released before the base class resources, maintaining a safe and logical destruction sequence from derived to base classes.

- *What does the following code snippet outputs?*

```
struct A {
    virtual void f() = 0;
    A() { f(); }
};

struct B : A {
    virtual void f() override { std::cout << "2"; }
};

int main() {
    B b;
}
```

- When an object of class `B` is instantiated, the constructor of its abstract base class `A` is called first. Inside `A`'s constructor, the pure virtual function `f()` is invoked. However, during the construction of `A`, the object is not yet fully a `B` instance, so the call to `f()` does not resolve to `B`'s implementation but instead attempts to call `A::f()`, which is pure virtual and lacks an implementation. This results in undefined

behavior, typically causing the program to crash or terminate abnormally at runtime without printing anything.

- *Can you iterate over a `struct` members data in C++?*
  - There is no inbuilt function/way to iterate over a `struct` in C++. However, if all members in the `struct` are of the same type, you can achieve it in the following way:

```
struct A {
    int a = 2;
    int b = 4;
    int c = 7;
    int d = 1;
    int e = 9;
    int f = 10;
    int g = 0;
};

int main() {
    A a;
    int* ptr = reinterpret_cast<int*>(&a);

    for (int i = 0; i < sizeof(A)/sizeof(int); ++i)
    {
        std::cout << *ptr << std::endl;
        ptr++;
    }
}
```

  - Be aware though that if the member data are not of the same type, you will have issue due to the `reinterpret_cast`.

- *Do you know what are ref-qualifiers? What are they used for?*
  - **Ref-qualifiers** in C++ are modifiers that can be applied to member functions to control how and when they can be invoked based on the value category (lvalue or rvalue) of the object they are called on.
    - **Lvalue Ref-qualifier (`&`):**
      Indicates that the member function can only be called on lvalue instances of the class. An lvalue refers to an object that has a persistent state and a identifiable memory address.

- **Rvalue Ref-qualifier (`&&`):**
  Specifies that the member function can only be invoked on rvalue instances. An rvalue represents temporary objects or objects that are about to be moved from, typically resulting from expressions like `std::move(obj)` or function returns by value.

```
class Buffer {
private:
    std::string data;

public:
    Buffer(const std::string& str) : data(str) {
        std::cout << "Buffer constructed with data: " << data <<
std::endl; }
    // Function to append data when called on an lvalue (modifiable)
    void append(const std::string& str) & {
        data += str;
        std::cout << "Appended on lvalue. New data: " << data <<
std::endl; }
    // Function to append data when called on an rvalue (movable)
    void append(const std::string& str) && {
        data += str;
        std::cout << "Appended on rvalue. New data: " << data <<
std::endl; }
    // Function to display data
    void display() const {
        std::cout << "Buffer data: " << data << std::endl; } };

int main() {
    Buffer buf1("Hello");
    buf1.display();

    // Calling append on an lvalue Buffer
    buf1.append(", World!");
    buf1.display();

    // Calling append on an rvalue Buffer using std::move
    std::move(buf1).append(" This is an rvalue.");
    buf1.display(); // buf1 is still valid, but was modified through
the rvalue

    // Directly creating an rvalue and calling append
    Buffer("Temporary").append(" Data for rvalue.").display();
}
```

- See this for running the code above and this for more info.

- *What does the following program output?*

```
template <typename T>
void foo(T& x) {
    std::cout << "T = const int? " << std::is_same_v<const int, T> <<
'\n'
              << "x = const int&? " << std::is_same_v<const int&,
decltype(x)> << '\n';
}

template <typename T>
void bar(const T& x) {
    std::cout << "T = const int? " << std::is_same_v<const int, T> <<
'\n'
          << "x = const int&? " << std::is_same_v<const int&, decltype(x)>
<< '\n';
}

int main() {
    const int i{};
    int j{};
    const int& c = j;

    foo(i); // Here T is const int and x is a const int&
    foo(j); // Here T is int and x is int&
    foo(c); // Here T is const int and x is const int&
    bar(i); // Here T is int and x is const int&
    bar(j); // Here T is int and x is const int&
    bar(c); // Here T is int and x is const int&
}
```

  - The program outputs "110011010101". This is due to how type deduction works.
    - Template deduction always remove the reference qualifier of the value being passed (for lvalue ref), so if an `int&` is passed to a function that take `T&`, T will be just and `int`, not `int&`.
    - If the parameter is non-`const` (e.g. `T&`), then T will take the type of the value passed + the `const`-qualifier if any. So if a `const int` is passed to `T&`, then T is deduced as `const int`.
    - If the parameter is `const` (e.g. `const T&`), T will take the type of the value **without** the `const`-qualifier. So if a `const int` is passed to `const T&`, then T is deduced as `int`.
    - Note that the type of the *parameter* itself will be the combination of all the attributes of the parameter. So if the an `int` is passed to a function which has

parameter `const T& x`, then x will have type `const int&`.
    - See this for the above code.
- When a template parameter is passed **by value**, the compiler deduces the type by **stripping away references and top-level `const` qualifiers** from the argument. This means that the deduced type `T` corresponds to the underlying type of the argument, without any reference or `const` modifiers applied directly to it.

```
// Template function with parameter passed by value
template <typename T>
void func(T x) {
if constexpr (std::is_same_v<T, int>) {
    std::cout << "T is int\n";
}
else if constexpr (std::is_same_v<T, double>) {
    std::cout << "T is double\n";
} else {
    std::cout << "T is something else\n"; }
}

int main() {
    int a = 10;
    const int b = 20;
    int& refA = a;
    const int& refB = b;

    func(a);    // T deduced as int
    func(b);    // T deduced as int (top-level const is stripped)
    func(refA); // T deduced as int (reference is stripped)
    func(refB); // T deduced as int (const and reference are
stripped)
    func(30);   // T deduced as int (rvalue)
}
```

Function Signature	Argument Type	Deduced `T`	Parameter Type
`template <typename T> void func(T x);`	`int`	`int`	`int`
`template <typename T> void func(T x);`	`const int`	`int`	`int`
`template <typename T> void func(const T x);`	`int`	`int`	`const int`
`template <typename T> void func(const T x);`	`const int`	`int`	`const int`

Function Signature	Argument Type	Deduced `T`	Parameter Type
`template <typename T> void func(T& x);`	`int`	`int`	`int&`
`template <typename T> void func(T& x);`	`const int`	`const int`	`const int&`
`template <typename T> void func(const T& x);`	`int`	`int`	`const int&`
`template <typename T> void func(const T& x);`	`const int`	`int`	`const int&`
`template <typename T> void func(T&& x);`	`int` (lvalue)	`int&`	`int& &&` → `int&`
`template <typename T> void func(T&& x);`	`int` (rvalue)	`int`	`int&&`
`template <typename T> void func(T&& x);`	`const int` (lvalue)	`const int&`	`const int& &&` → `const int&`
`template <typename T> void func(T&& x);`	`const int` (rvalue)	`const int`	`const int&&`

- *Does the following code snippet compiles?*

```
int main() {
    std::vector<char> delimiters = { ",", ";" };
    std::cout << delimiters[0];
}
```

  - Yes it compiles, but it is undefined behaviour. We are passing two string literals in an initializer-list (not actual char), so the compiler will look for a constructor that can match. Turns out that string literals are in fact `const char*`, which match with the concept of `InputIterators`, hence the `std::vector` constructor called is the following:

```
    template <class InputIterator> vector(InputIterator first,
InputIterator last)
```

  - Now the constructor believes it has been passed two iterators to the same sequence, but it has actually been passed iterators to two completely different sequences, `","` and `";"`. *§[forward.iterators]¶2* says: "The domain of `==` for forward iterators is that of iterators over the same underlying sequence.". So the result of this program is undefined.

- *What does the following code snippet outputs?*

```
struct A {
A() { std::cout << "1"; }
A(const A&) { std::cout << "2"; }
A(A&&) { std::cout << "3"; } };

struct B {
A a;
B() { std::cout << "4"; }
B(const B& b) : a(b.a) { std::cout << "5"; }
B(B&& b) : a(b.a) { std::cout << "6"; } };

int main() {
B b1;
B b2 = std::move(b1); }
```

  - The result is `1426`. Know that the member variable constructors are always called first before the constructor of the object itself.
  - In a non-delegating constructor (...), if a given potentially constructed subobject is not designated by a *mem-initializer-id* (...), then
    — if the entity is a non-static data member that has a default member initializers;
    — otherwise, if the entity is an anonymous union or a variant member;
    — otherwise, the entity is default-initialized.
  - Then, `b2` is initialized with the move constructor (since `std::move(b1)` converts `b1` to an xvalue, allowing it to be moved from). In `B`'s move constructor, `a` is initialized in the member initializer list. Even though `b` is of type rvalue reference to `B` (and bound to an rvalue), the expression `b.a` is an lvalue. Lvalues cannot be moved from, so `b2.a` is copy initialized, printing `2`. Finally, the body of `B`'s move constructor prints `6`.

- *Why is it not possible (or at least not easily) to separate the declaration and definition of a templated file into an .hpp and .cpp files?*
  - In C++, templates are instantiated at compile time, meaning that the compiler needs to see the full definition (both declaration and implementation) of any template in every translation unit where it is used. If you separate the declaration of a templated class into a header file and its definition into a separate cpp file, the compiler won't have access to the implementation when it attempts to instantiate the template with specific types. This leads to linker errors or missing symbols because the instantiation process never occurs. That's why templated classes and functions are typically defined entirely

in header files (or in files included by the header) so that all translation units have access to the complete implementation required for instantiation.

- Templated classes and functions serve as blueprints for the compiler, rather than being concrete functions or classes themselves. When you write a template, you're providing a generic definition that the compiler uses to generate specific instantiations when you use the template with concrete types. This means that until you actually instantiate a template with a particular type, there is no real code corresponding to that template in the final program—it's only a pattern for creating code. This is also why the full template definition must be available in every translation unit that uses it: the compiler needs to see the blueprint to create the actual, type-specific implementations during compilation.

- *What is the lifetime of a string literal?*
  - In C++, string literals have **static storage duration**. This means that they are allocated when the program starts and exist for the entire lifetime of the program, until it terminates. Because of this, you can safely use a pointer to a string literal in your code without worrying that the underlying memory will go out of scope. However, it's important to remember that string literals are immutable—attempting to modify one results in undefined behavior.
  - As a result the following code is possible:

```
// This function should be made constexpr or even consteval since the returned string literal is known at compile time. Not doing so is actually bad optimization.
std::string_view data()
{
    return "hello world!"; // OK, the string literal outlives the function body
}

std::string_view data2()
{
    std::string str = "hello world!";
    return str; // UNDEF BEHAVIOUR, the std::string has automatic storage duration and will be destroyed at the end of this scope. The returned `string_view` thus point to deleted memory
}

int main()
{
    std::string_view sv = data();
    std::string_view sv2 = data2();
```

```
    std::cout << sv; // OK
    std::cout << sv2; // UB
}
```

- *What does the following code snippet outputs?*

```
struct A {
    A() = default;
    A(const A &a) { std::cout << '1'; }
    A(A &&a) { std::cout << '2'; }
};

A a;

int main() {
    [a = std::move(a)]() {
    [a = std::move(a)]() {};
        }();
}
```

  - The outer lambda captures the `a` from the global scope. The global `a` is non-`const`, so when we do `std::move(a)`, the result is a non-`const` rvalue reference. When initializing the capturing member of the closure type, the move constructor of `A` is used and `2` is printed.
  - The inner lambda captures the `a` from the outer lambda, not the `a` from the global scope. This happens inside the function call operator of the outer closure type. That function call operator is `const`, so the access to the `a` member is treated as `const`. When we do `std::move(a)`, the result is a `const` rvalue reference. When initializing the capturing member of the inner closure type, the non-`const` rvalue reference in `A`'s move constructor is not viable, and instead the copy constructor is used, causing `1` to be printed.

- *What about the following?*

```
int main() {
    std::cout << 1["ABC"];
}
```

  - The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))`. In our case `1["ABC"]` is identical to `*(1+"ABC")`. Since the plus operator is commutative, this is identical to `*("ABC"+1)`, which is identical to the more familiar `"ABC"[1]`.

- *What does the following code snippet outputs?*

```
char a[2] = "0";

struct a_string {
a_string() { *a='1'; }
~a_string() { *a='0'; }
const char* c_str() const { return a; }
};

void print(const char* s) { std::cout << s; }
a_string make_string() { return a_string{}; }

int main() {
a_string s1 = make_string(); // RVO is used here, so the temp from
make_string() is not destroyed and directly passed to s1
print(s1.c_str());
const char* s2 = make_string().c_str(); // Here it's different,
make_string() creates a temp, call the c_str() func which pass the
resulting const char* to s2 then, the temp is destroyed
print(s2);
print(make_string().c_str());
}
```

`a_string s1 = make_string();` creates an `a_string`, and `a` is set to `1`. This object is not destroyed (and `a` set back to `0` again) until at the very end of `main()`, after we're done printing.
Then `print(s1.c_str());` is called, printing the value of `a`, which is now `1`.

Next, we do `const char* s2 = make_string().c_str();`. The temporary `a_string` created by the call to `make_string` sets `a` to `1`, but then back to `0` again when it is destroyed. By the time we get to `print(s2);`, it prints `0`. Had this been a real `std::string`, `print(s2);` would cause undefined behavior since `s2` is an invalid pointer!

In `print(make_string().c_str());` the temporary object is instead destroyed *after* the call to `print` returns, so `1` is printed.

- *What is the difference between `new`, placement `new` and `::operator new`?*

- `new`:
  This is the new-expression you write in your code (for example, `new T(args)`). It does two things: it first calls an allocation function (usually the global `::operator new` or an overloaded version) to allocate raw memory, and then it calls the constructor of type `T` to initialize the object in that memory. Use **`new`** when you want the compiler to both allocate memory and construct the object.
- placement `new`:
  This is a special form of new that allows you to construct an object in a pre-allocated block of memory. It is used by writing something like `new (ptr) T(args)`, where `ptr` is a pointer to memory that you've already allocated. Unlike the normal new-expression, placement new does not allocate any memory; it only calls the constructor at the specified location. Use **placement `new`** when you already have memory allocated (perhaps in a buffer or from a custom allocator) and you simply want to construct an object in that memory. Note that as of C++20, there is a new function `std::construct_at` which does exactly the same thing as placement `new`, but also works in constant expressions. Since `std::construct_at` conveys the intend more clearly than placement `new`, it should typically be preferred to the former when available.
- `::operator new`:
  This is the low-level memory allocation function that is invoked by the new-expression to obtain raw memory. It is a function (or can be overloaded as a function) that usually has a signature like `void* operator new(std::size_t size)`. It is separate from the new-expression itself; while the new-expression calls `::operator new` (or a class-specific version) to allocate memory, it then proceeds to call the constructor to create the object. The **`::operator new`** function is usually not called directly by most application code. Instead, it is the underlying allocation function that gets called by new-expressions; however, you might overload it in your own classes or globally to customize memory allocation.

- *What does the following code returns?*

```cpp
void g(int&) { std::cout << 'L'; }
void g(int&&) { std::cout << 'R'; }

template<typename T>
void f(T&& t) {
    if (std::is_same_v<T, int>) { std::cout << 1; }
    if (std::is_same_v<T, int&>) { std::cout << 2; }
    if (std::is_same_v<T, int&&>) { std::cout << 3; }
    g(std::forward<T>(t)); }
```

```
int main() {
    f(42);
    int i = 0;
    f(i);
}
```

- '1R2L' is returned. This is a key part of how forwarding references (or universal references) work. The type deduction rules ensure that:
  - If the argument is an lvalue, `T` is deduced as an lvalue reference type (e.g., `int&` for `int`).
  - If the argument is an rvalue, `T` is deduced as the base type (e.g., `int` for `int&&`).
- This behavior allows the template to forward the exact type of the argument while maintaining reference semantics.
- However, the type of the paramater 't' itself is 'int&&' for 'f(42)' and 'int&' for 'f(i)' due to reference collapsing.

- *What is more efficient: passing a `std::string_view` or a `const std::string&` in a function parameter?*
  - When you call a function that takes a `const std::string&` with a string literal, the compiler creates a temporary `std::string` from that literal. This temporary object is then bound to the reference parameter: even though the reference itself doesn't cause a copy if you already have a `std::string`, in this case there isn't an existing `std::string` to refer to. In contrast, if you call a function that takes a `std::string_view` with a string literal, the `std::string_view` is constructed directly from the literal without creating any temporary `std::string` object. The string view simply holds a pointer to the literal along with its length, making it more efficient in cases where you don't need the additional functionality or ownership semantics of a `std::string`.
  - Note also that you should never pass a `std::string_view&` as this is a unecessary indirection. A `std::string_view` is a pointer itself.
  - See this and this.

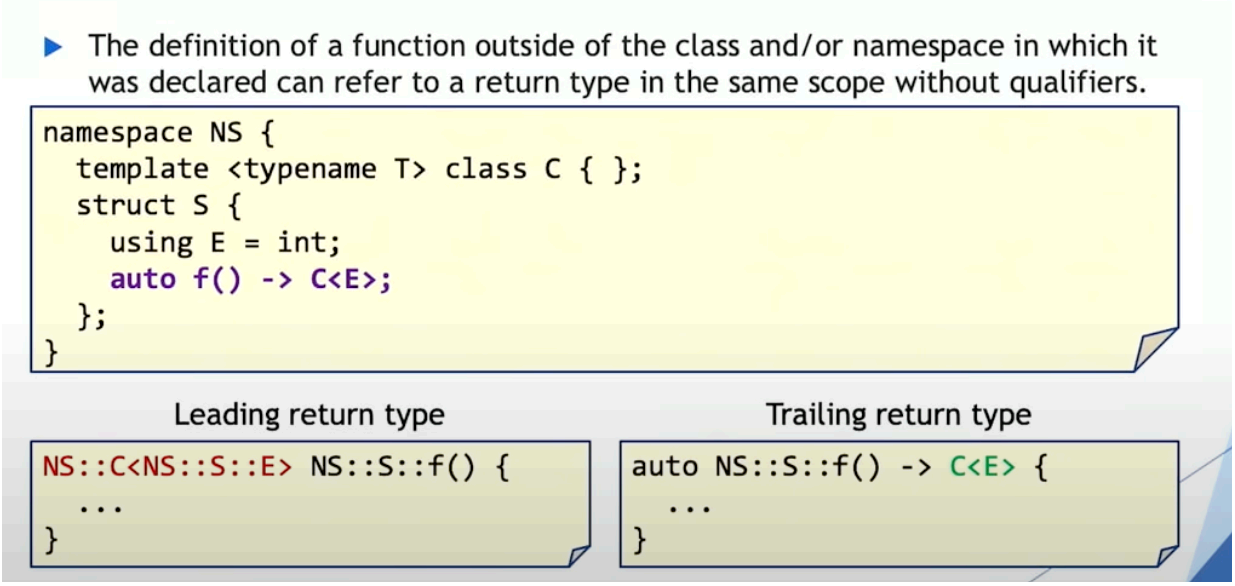
- *What is an allocator? Can you implement one?*
  - See this video for a custom allocator implementation.

- *What happens if you try to add two `bool` in C++?*
  - In C++, arithmetic operations on smaller integer types (such as `short`, `bool`, and `char`) are automatically promoted to `int` due to a set of rules known as integral promotions. These rules are part of the language's implicit conversion mechanisms that ensure that arithmetic operations are performed on a type that is at least as large as an `int`. The reasons for these promotions include:
    1. **Consistency and Simplicity:**
       By promoting smaller types to `int`, the language ensures that all arithmetic operations are carried out on a common type, simplifying the rules for operator overloading and ensuring consistent behavior across different types.
    2. **Performance Considerations:**
       Most hardware architectures are optimized to perform arithmetic on types that are the size of an `int` or larger. Using `int` for arithmetic allows the compiler to generate efficient machine code without needing special handling for smaller types.
  - Note that this conversion also applies to `unsigned` types:
    - When you perform arithmetic (like addition), C++ applies the "usual arithmetic conversions" to ensure that both operands are of a common type. One key rule is that any integral type with a lower conversion rank than `int` is promoted to `int` —if an `int` can represent every value of the original type; otherwise, it is promoted to `unsigned int`.
    - Types such as `unsigned short`, `char`, or `bool` have a lower conversion rank than `int`. That means when you use them in an arithmetic operation, they are automatically promoted to `int` (assuming that an `int` can hold all the values that the smaller type can represent). For example, if `unsigned short` is smaller than `int` on your platform, every `unsigned short` is promoted to `int` before the operation is carried out.

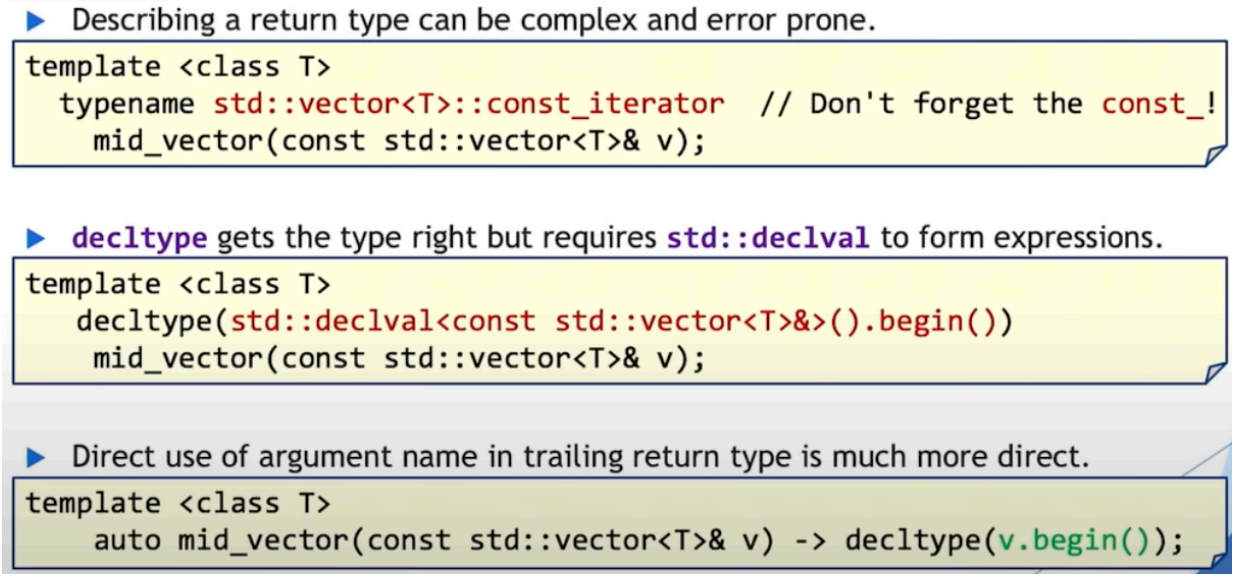
- *What is an aggregate in C++?*
  - An **aggregate** is an array or a class with
    1. no user-declared or inherited constructors,
    2. no private or protected direct non-static data members,
    3. no private or protected direct base classes, and
    4. no virtual functions or virtual base classes.

- *What are trailing return types used for in C++?*

- First it can help you write less. For example:



- Secondly, it is really helpful in metaprogramming, where it can be used with `dcltype()` on a function parameter as shown in the following:



- It is also the main way to specify a return type for a lambda.
- See this for more info.

- *What is RVO? What about NRVO?*
  - **Return Value Optimization (RVO)** and **Named Return Value Optimization (NRVO)** are compiler optimizations in C++ that eliminate unnecessary copying or moving of objects during function returns.
    - **RVO:**
      Return Value Optimization is a compiler optimization that constructs the return value of a function directly in the memory location where the caller expects it, rather than creating a temporary object and then copying or moving it. This can

occur even when the returned object is an unnamed temporary. Because the object is constructed directly in its final

- **NRVO:**
  Named Return Value Optimization is a specific form of RVO that applies when the function returns a named local variable. Normally, returning a named variable might involve copying or moving that object to the caller, but with NRVO, the compiler is allowed (and, in many cases, required by modern C++ standards) to construct that named variable directly in the caller's space. As a result, no extra copy or move operation is needed even though the object has a name.

- Note that certain cases prevent RVO and NRVO. For example, if a function returns an object based on a conditional statement that will be evaluated at runtime, then RVO is impossible.

- *Are arrays and pointers similar in C++?*
  - An array type in C++ allocates a fixed block of contiguous memory for a specified number of elements, and its type inherently includes the size (e.g., `int[10]`). When you declare an array like `int arr[10]`, the compiler reserves enough space to store ten integers, and operations such as `sizeof(arr)` yield the total memory used by the entire array. In many contexts, an array will automatically decay to a pointer to its first element (i.e., an `int*`), but even then, the original array retains its size and layout information, and its name cannot be reassigned.
  - In contrast, a pointer like `int* ptr` is simply a variable that stores a memory address and does not by itself allocate memory for a series of objects. The pointer can be reassigned to point to different memory locations, and `sizeof(ptr)` yields the size of the pointer variable itself, not the size of the memory block it may point to. Although arrays often decay to pointers when passed to functions, they are fundamentally different from pointers, as arrays carry size information and have fixed storage duration, whereas pointers are flexible and can refer to dynamically allocated or varying blocks of memory.

- *What does the following code snippet outputs?*

```
struct X {
    int var1 : 3;
    int var2;
};

int main() {
```

```
    X x;
    std::cout << (&x.var1 < &x.var2);
}
```

- This is a compilation error. The expression `int var1 : 3` declares a bit field of size 3. In C++, you cannot use the address operator `&` (or have a reference) on a bit field because bit fields are not objects that occupy a full, addressable memory location. References require an actual object with a distinct address to bind to, but bit fields are stored as portions of an integral type and typically do not have independent addresses. As a result, the language disallows taking the address of a bit field, which in turn makes it impossible to bind a reference to one.
- **Note that you can only have bit field of integral types (int, bool, enum, etc.)!**

- *What is type punning in C++?*
  - **Type punning** is the practice of treating a piece of memory as if it were an object of a different type than the one it was originally defined as. In other words, it's a way to reinterpret the underlying bit representation of an object as a different type. This technique is often used to inspect or manipulate the low-level binary representation of data—for example, viewing the bits of a floating-point number as an integer or vice versa. Common methods to achieve type punning in C++ include using unions, reinterpret_cast, or (in C++20) std::bit_cast. Each of these methods allows you to bypass the usual type-safety guarantees and access the raw memory of an object.
  - While type punning can be useful for low-level programming, it comes with risks because it can easily lead to undefined behavior if not done carefully. The C++ standard imposes strict aliasing rules, which are intended to ensure that objects are accessed only through pointers or references of compatible types. Violating these rules —by, for example, reading a float through an int pointer—can result in compiler optimizations that break your code unexpectedly. Therefore, when type punning is required, it's important to use well-defined methods like `std::bit_cast` (which has clearly defined behavior) to safely reinterpret data without invoking undefined behavior.
  - See this talk for more info.

- *Can `static` variables be set `constexpr`?*
  - The key point is that "constexpr" and "static" refer to different aspects of a variable's life and use, and they can work together without conflict. Declaring a variable as `constexpr` means that its value is known at compile time and can be used in constant expressions. It tells the compiler that the variable is immutable and its value can be

computed during compilation. On the other hand, `static` refers to the variable's storage duration—it persists for the entire execution of the program. A `static constexpr` variable, therefore, has both properties: its value is a compile-time constant and it is stored in a fixed memory location for the duration of the program.

- This is not contradictory because having a value known at compile time does not mean the variable ceases to exist at runtime. Instead, it means that the compiler can substitute the constant value wherever the variable is used, and if needed, the variable is also available in the program's static storage. For example, a declaration like `static constexpr int value = 42;` allows `value` to be used in compile-time contexts (such as template arguments or constant expressions) while also existing as a read-only value during the program's execution. Thus, the two concepts complement each other rather than conflict.

- *What is the strict aliasing rule?*
  - See this

- *What is the difference between `alignof()` and `sizeof()`?*
  - `sizeof()` determines the size, in bytes, of a data type or variable. This includes any padding added by the compiler for alignment purposes.
  - `alignof()` determines the alignment requirement of a data type. Alignment refers to the memory address where a variable of that type can be stored. It ensures that the variable is stored on an address that is a multiple of its alignment requirement.

```
struct S {
    int a; // Size: 4 bytes, Alignment: 4 bytes
    double b; // Size: 8 bytes, Alignment: 8 bytes
    char c; // Size: 1 byte, Alignment: 1 byte
};
```

  - In the above example, the `sizeof(S)` will likely be 24 bytes due to padding, while the `alignof(S)` will be 8 bytes, which is the largest alignment requirement of its members. This is because the structure `S` needs to be aligned on an 8-byte boundary due to the `double b` member.

- *Can you capture a global variable in a lambda function?*

- No, you can only capture object with automatic storage duration. Moreover, given that global (static) variables can be used in any scope within the translation unit, being able to capture one would be kind of redundant (they can already be used within the lambda without capture).

- *What does the following code snippet outputs?*

```
int foo() { return 10; }
struct foobar {
    static int x;
    static int foo() { return 11; }
};
int foobar::x = foo();

int main() {
std::cout << foobar::x; }
```

  - In the statement `int foobar::x = foo();` we redeclare the variable `x`. Its target scope is a class scope introduced by the declaration of the class `foobar`. The portion after `x` is also included in that scope. So when looking for `foo` we find `foobar::foo` there and don't search any further. This mechanism is called Argument Dependent Lookup (ADL) for more info or Koenig Lookup..

- *What does the following code snippet outputs?*

```
namespace x {
class C {};
    void f(const C& i) { std::cout << "1"; }
}

namespace y {
    void f(const x::C& i) { std::cout << "2"; }
}

int main() {
    f(x::C());
}
```

  - Since the functions `f` are declared inside namespaces, and the call `f(x::C())` is unqualified (not preceded by `x::` or `y::`), this would normally not compile.

    - However, due to argument-dependent name lookup a.k.a. "Koenig lookup", the behavior is well-defined.
    - With argument-dependent name lookup, the namespaces of the *arguments to* a function is added to the set of namespaces to be searched for that function. Since we're passing an `x::C` to `f`, the namespace `x` is also searched, and the function `x::f` is found.

- *What is SFINAE?*
    - SFINAE stands for "Substitution Failure Is Not An Error." It is a compile-time mechanism used in C++ template metaprogramming that allows the compiler to silently discard invalid template instantiations during overload resolution rather than treating them as errors. When the compiler substitutes a type into a template and that substitution leads to an invalid type or expression, instead of generating a compilation error, the compiler simply removes that candidate from the set of overloads. This allows multiple templates or specializations to coexist, and the most appropriate one is selected based on whether the substitution succeeds.
    - A common use of SFINAE is to conditionally enable or disable function templates or class template specializations based on type traits or other compile-time properties. For example, you can use SFINAE with `std::enable_if` to define a function template that only participates in overload resolution if a particular condition (such as the presence of a member type or the convertibility between types) is met. This enables more flexible and robust generic programming, as you can create templates that adapt their behavior depending on the characteristics of the types provided, all while avoiding hard errors when a substitution doesn't match the intended criteria.
    - Note that C++20 Concepts significantly reduced the need to write SFINAE code explicitly and make templates easier to work with.

- *Can you modify a string literal?*
    - Yes, but this is undefined behaviour. It is generally placed in the read-only memory, and 2 string literals might even share the same memory. So it would not be a good idea to modify them.

- *What is the `volatile` keyword?*
    - The `volatile` keyword in C++ is a type qualifier that tells the compiler that a variable's value may change at any time—outside of the control of the program (for

example, by hardware, a different thread, or an interrupt handler). Because of this, the compiler is instructed not to optimize accesses to that variable: it must read from the memory location each time the variable is used, rather than caching its value in a register or assuming that successive accesses yield the same result.
    - This behavior is particularly important in low-level programming scenarios, such as when working with memory-mapped hardware registers, handling signals, or interacting with variables modified by concurrent processes in certain embedded systems. However, it's important to note that `volatile` does not provide any synchronization or ordering guarantees for multi-threaded programming; it merely prevents certain compiler optimizations. For thread synchronization, one should use appropriate synchronization primitives (like mutexes, atomic operations, or C++20's `std::atomic`).

- What is the `register` keyword?
    - The `register` keyword in C++ was originally introduced as a hint to the compiler that the declared variable might be used frequently and should therefore be stored in a CPU register for faster access, rather than in memory. This was intended to help optimize performance by reducing memory access times. However, modern compilers have sophisticated optimization algorithms that automatically determine which variables should reside in registers, making the `register` keyword largely redundant.
    - As a result, modern C++ standards (starting with C++11 and further deprecated in C++17) do not require or even consider the `register` keyword; compilers simply ignore it. Additionally, because variables declared as `register` are not allowed to have their address taken (remember that registers are hardware components that typically do not have accessible memory addresses in the same way that RAM), its use was somewhat restrictive. Today, it's best to let the compiler's optimizer handle storage decisions and avoid using the `register` keyword altogether.

- *What is a C++ `union`? Can you give an example where it is useful?*
    - A **union** in C++ is a special user-defined type that allows different data members to share the same memory location. Only one of the union's members can hold a value at any given time, which means the union's total size is determined by its largest member. This design enables efficient use of memory, especially when you need to store one of several possible types but never more than one at a time. Unions are commonly used for low-level data manipulation and type punning, allowing the programmer to interpret the same bytes in different ways.

- See example here.

- *If I have a function that returns auto, and within that function I have an if-else statement where one branch returns an `int` and the other one returns a `bool`, will the type being returned depend on the branch?*
  - The return type of a function declared with `auto` is deduced at compile time based on all the return statements in the function. All the return expressions must be convertible to a single common type. In your example, one branch returns an `int` and another returns a `bool`. Since `bool` can be implicitly converted to `int` (with `false` converting to 0 and `true` to 1), the compiler deduces the function's return type as `int`. This means that regardless of which branch is taken at runtime, the function returns an `int`. If the return expressions cannot be converted to a common type, the code will be ill-formed.

- *What does `std::hash`? How can you implement your own hash function?*
  - `std::hash` is a function object (or functor) template defined in the `<functional>` header that provides a standard way to compute hash values for objects of various types. It is specialized for many built-in types (such as integers, pointers, and strings) as well as for user-defined types if you provide an appropriate specialization or overload. The hash value it returns is of type `std::size_t`, and this value is used, for example, in unordered associative containers like `std::unordered_map` and `std::unordered_set` to determine where to store objects.
  - When you call `std::hash<T>{}(value)`, it invokes the `operator()` defined for that specialization of `std::hash`, which computes a hash based on the value's internal representation. This standardization of hashing helps ensure consistent behavior across different parts of a program or different implementations, and it allows the standard library to rely on `std::hash` when it needs to hash keys for efficient lookup and storage. If you have a custom type, you can specialize `std::hash` for that type to allow it to be used in unordered containers.
  - Let's say that you want to create your own hash function for a `std::pair`, here is a code sample of how you could do it:

```
struct HashPair
{
    template <typename T, typename S>
    std::size_t operator()(const std::pair<T , S>& p) const
    {
        // Equivalent to boost::hash_combine()
```

```
        return std::hash<T>{}(p.first) ^ (std::hash<S>{}(p.second) <<
1);
    }
}

int main()
{
    std::pair<int, char> p{1, 'a'};
    std::unordered_map<std::pair<int,char>, int, HashPair> map;
    map.insert({p, 2});
}
```

- Remember that the operator() should be const!!

- *What does the following code snippet output?*

```
void f(const std::string &) { std::cout << 1; }

void f(const void *) { std::cout << 2; }

int main() {
  f("foo");
  const char *bar = "bar";
  f(bar);
}
```

- A string literal is not a `std::string`, but a `const char[]`. If the compiler was to choose `f(const std::string&)`, it would have to go through a user defined conversion and create a temporary `std::string`. Instead, it prefers `f(const void*)`, which requires no user defined conversion.

- What does the following code snippet output?

```
int main() {
unsigned short zero = 0, one = 1;
if (zero - one < zero)
    std::cout << "less";
else
    std::cout << "more"; }
```

- This is implementation defined. If we are on an implementation where `short` and `int` have different sizes (`short` size smaller than `int` size), then since `short` has a smaller conversion rank than `int`, the operands are promoted to `int` before

applying the operation. This is a process called as integral promotion, which basically says that the usual arithmetic conversion promotes integral types of conversion rank lower than `int` to `int` if `int` can represent all the values of the source type, and otherwise to `unsigned int`.
- If `short` and `int` have the same size, `unsigned short` instead gets converted to an `unsigned int`, the result wraps around to a large value, and `more` is printed.

- *What does the following code snippet outputs?*

```
int main() {
    try {
        throw std::out_of_range("");
    } catch (std::exception& e) {
        std::cout << 1;
    } catch (std::out_of_range& e) {
        std::cout << 2;
    }
}
```

  - 1 is the output. `std::out_of_range` binds to both catch block arguments. Even though the second catch block seems to be the more specialized, thus the more appropriate, the first catch-block in order of appearance is chosen. According to the standard: "The handlers for a try block are tried in order of appearance.".

- *Can you define a static member data directly within the class declaration?*
  - Before C++17, a static data member declared inside a class is only a declaration—it tells the compiler that a static member exists, but it does not allocate storage for it. The C++ standard requires that there be exactly one definition of a static member (i.e., one place where storage is actually allocated), so you must provide an out-of-class definition in a single translation unit. If you were allowed to define the static variable directly in the class definition, then every translation unit that includes that header would get its own copy, violating the One Definition Rule and causing linkage errors.
  - With C++17, the introduction of inline static variables allows you to both declare and define a static member directly in the class definition by marking it as inline. This tells the compiler that it is allowed to have multiple definitions across translation units as long as they are identical, and the linker will merge them into one. However, without the inline specifier (or for types that aren't constexpr integrals), you cannot define the static member inside the class because doing so would lead to multiple definitions and thus linker errors.

- *What is the One Definition Rule (ODR) in C++?
  - The **One Definition Rule (ODR)** is a fundamental principle in C++ that requires every object, function, class, and template to have exactly one definition in a program, even if it is declared in multiple translation units (i.e., source files). In other words, while you can declare an entity (like a function or a variable) as many times as you like—usually by including headers—the program must contain only a single definition of that entity. This rule helps avoid ambiguity during linking and ensures that every reference to that entity refers to the same implementation or storage.
  - There are some exceptions and special cases under the ODR. For example, inline functions, templates, and inline static variables in C++17 and later can have multiple definitions as long as they are identical across translation units; the linker will merge these definitions into a single one. However, for most non-inline functions or variables, if you inadvertently define them in more than one translation unit, you will end up with multiple definitions, which violates the ODR and usually results in linker errors. Violating the ODR leads to undefined behavior, making it crucial to ensure that each entity is defined only once in the entire program.*

- *What happens if I declare a static variable without giving it a value?*
  - According to the C++ standard, all objects with static storage duration are zero-initialized before any other initialization takes place.

- *What is `std::hash`? Which underlying algorithm is used?*
  - `std::hash` is a function object template provided by the C++ Standard Library (defined in `<functional>`) that computes a hash value for an object of a given type. Its primary purpose is to produce a `std::size_t` value that serves as a numerical representation of an object's content, which is especially useful in hash-based containers like `std::unordered_map` and `std::unordered_set`.
  - In essence, `std::hash` encapsulates the process of converting an object into a number in a way that distributes similar objects across a wide range of values, helping minimize collisions in hash tables. This hash value is then used by the container to quickly locate elements, leading to average constant-time complexity for insertions, deletions, and lookups. By standardizing the hash computation across types, `std::hash` simplifies the design and use of generic hash-based data structures.
  - The hash function uses Fowler–Noll–Vo hash algorithm.

- *Why do we need hash functions? What are their purpose?*
  - A hash function provides a mapping from a key to a bucket of values. For many simple types, the default hash function (for example, `std::hash<int>`) is essentially the identity function or performs a minimal transformation, but it's still used to compute an index (often via a modulus operation) into an array of buckets. This design enables the hash map to efficiently organize and locate elements using average constant-time lookups.
  - Hash maps use buckets to efficiently organize and look up keys while controlling memory usage and maintaining fast average performance. Instead of maintaining a direct, one-to-one mapping for every possible key (which would require an enormous amount of contiguous memory if the key domain is large or sparsely populated), hash maps compute a hash value for each key and then map that hash to a bucket index (typically using a modulus operation). Each bucket can hold one or more keys, which is how the hash map handles the possibility of collisions—situations where different keys produce the same bucket index. See this for more info.
  - To summarize the objectives of a hash function are as follow:
    - Minimize collisions
    - Uniform distribution of hash values
    - Easy to calculate
    - Resolve any collision

- *What is the difference between linear probing and separate chaining? Can you give some advantages/disadvantages of both methods?*
  - **Linear Probing** is a collision resolution technique used in open addressing hash tables. When a key hashes to an index that is already occupied, linear probing checks the next slot (and continues checking sequentially) until it finds an empty one. This means that every key is stored directly in an array slot, and collisions are resolved by "probing" along the array. While linear probing can be cache-friendly due to its contiguous memory accesses, it can suffer from "primary clustering"—where groups of filled slots form and cause longer probe sequences, which may degrade performance as the table fills up.
  - **Chaining**, on the other hand, resolves collisions by having each bucket (or slot in the hash table) hold a collection (often a linked list or a dynamic array) of keys that hash to that bucket. Instead of searching for an empty slot, all keys that collide are stored together in the same bucket. This method avoids the clustering issues seen in linear probing and can handle an arbitrary number of collisions gracefully. However, chaining usually involves additional memory allocation for each node in the chain and can have

less predictable memory access patterns, which might affect performance in certain situations.
- Open-addressing is usually faster than chained hashing when the load factor is low because you don't have to follow pointers between list nodes. It gets very, very slow if the load factor approaches 1, because you end up usually having to search through many of the slots in the bucket array before you find either the key that you were looking for or an empty slot. Also, you can never have more elements in the hash table than there are entries in the bucket array. See this thread for more info.

- *What are different algorithms used in open addressing to handle key collision? Why were they invented?*
  - **Linear probing**, which is the most simple type of open addressing technique, came with an issue: it can suffer from "primary clustering"—where groups of filled slots form and cause longer probe sequences, which may degrade performance as the table fills up. To avoid this, other open addressing algorithms were invented to handle this shortcoming of the linear probing technique:
    - **Quadratic Probing:**
      Uses a quadratic function (i.e., index + 1^2, $+2^2$, $+3^2$, …) for probing. This reduces primary clustering compared to linear probing.
    - **Double Hashing:**
      Uses a second hash function to compute the step size, providing a more uniform distribution of probe sequences and reducing clustering.

- *What does the following code snippet outputs?*

```
int i = 1;

int main()
{
    int i, j = 2;
    std::cout << i;
}
```

  - In this case, the `i` being printed is the one with automatic storage duration in the most innermost scope, which has not been initialized, thus containing some junk value. If you wanted to call the global `i`, you would need to use the scope resolution operator for the global space: `::i`.

- In C++, when two variables have the same name, the one declared in the innermost (or local) scope **shadows** any variable with the same name in an outer (or global) scope. In your example, the local variable `i` declared in `main()` hides the global `i` within that function. This is a result of the scope resolution rules: the compiler always first looks in the current, most immediate scope for a variable before considering any outer scopes. Thus, when you write `std::cout << i;` inside `main()`, it prints the local `i` (which is uninitialized in your example) rather than the global `i`.

- *Let's assume you have a `Vector3` class which is a 3-dimensions vector. If I want to implement the `operator +` or `operator <<`, should they be declared as member functions or non-member functions?*
  - In this kind of circumstances, it's usually best to implement the binary `+` operator as a non-member function. Here's why:
    - **Symmetry & Implicit Conversions:**
      When implemented as a non-member, both the left-hand and right-hand operands can participate in implicit conversions. This is especially useful if you want to allow expressions like `vector + someConvertibleType`.
    - **Non-Modifying Operation:**
      The `+` operator generally doesn't modify either operand but returns a new object representing the result. Making it a non-member clearly indicates that it's a pure function.
    - **Encapsulation:**
      Even if your vector's internal data is private, you can declare the non-member operator as a friend if needed. This gives the operator access while still keeping the implementation separate from the class interface.
  - It is also generally best to declare this kind of operator as `friend` since they often need to access private/protected members of such classes.
  - Note that operators like `operator +=` SHOULD be member functions, since they modify directly the object (and do not return a new object).

- *What is `alloca()` ? How is it different from a C-style array?*
  - Memory allocated with `alloca` is taken from the stack. This memory is automatically reclaimed when the function that called `alloca` returns, just like local variables. This means you don't have to explicitly free the memory, unlike memory allocated on the heap with `malloc` or `new`.

- Unlike typical C-style arrays where the size is fixed at compile time (unless using C99 Variable Length Arrays, which aren't available in standard C++), `alloca` allows you to allocate a block of memory whose size is determined at runtime.
- The lifetime of the memory allocated by `alloca` is tied to the function's stack frame. When the function exits, all memory allocated by `alloca` is automatically freed. This makes it useful for temporary allocations that do not need to persist beyond the function call.
- Note that `alloca` is not part of the C or C++ standards. It is available as a compiler extension in many environments (such as GNU C, MSVC, and others), but its availability is not guaranteed across all platforms and compilers.

- *What is `std::forward` ? What is it used for? What is the difference with `std::move` ?*
  - `std::forward` is a utility function template in C++ used to preserve the value category (lvalue or rvalue) of function arguments in template code, particularly in the context of perfect forwarding. When writing generic code, such as in function templates or constructors, you often need to forward parameters to another function or constructor without inadvertently converting an rvalue into an lvalue. `std::forward` helps achieve this by conditionally casting its argument to an rvalue only if the original argument was an rvalue, thereby ensuring that move semantics are correctly applied. This is essential in scenarios like implementing wrapper functions or factory functions that forward arguments to other functions or constructors while maintaining their original lvalue/rvalue nature.
  - `std::move` and `std::forward` both cast objects to rvalue references, but they serve different purposes:
    - **std::move:**
      It unconditionally casts its argument to an rvalue, signaling that the object can be "moved from." It doesn't actually move anything by itself—it merely allows move semantics to be applied by treating the object as an rvalue.
    - **std::forward:**
      It conditionally casts its argument depending on its template parameter, preserving the original value category (lvalue or rvalue). This is essential in perfect forwarding, where you want to forward an argument to another function while keeping its lvalue or rvalue nature intact.
  - In short, use `std::move` when you want to explicitly treat an object as an rvalue, and use `std::forward` in template code to forward parameters without losing their original value category.
  - See this video.

- *Can you show me a possible implementation of `std::move` and `std::forward`?*

```cpp
// ----------- move -----------

template <typename T>
constexpr typename std::remove_reference<T>::type&& move(T&& t) noexcept
{
    // Remove any reference from T and then cast t to an rvalue
reference.
    return static_cast<typename std::remove_reference<T>::type&&>(t);
}

// ----------- forward -----------

// This overload handles lvalues.
template <typename T>
constexpr T&& forward(typename std::remove_reference<T>::type& t)
noexcept
{
    return static_cast<T&&>(t);
}

// This overload handles rvalues.
template <typename T>
constexpr T&& forward(typename std::remove_reference<T>::type&& t)
noexcept
{
    // The static_assert prevents binding an rvalue to an lvalue
reference type.
    static_assert(!std::is_lvalue_reference<T>::value, "Bad forward:
cannot forward an rvalue as an lvalue");
    return static_cast<T&&>(t);
}
```

  - See this to run the code and this for more info.
  - Here is a step-by-step recap of what is happening in the case `forward<int>(std::move(i))`:
    - **1. Explicit Template Argument:**
      You explicitly specify `T` as `int`. So for the call, $T = int$.
    - **2. Determine Parameter Types Using remove_reference:**
      - In the **lvalue overload**, the parameter type is:
        `typename std::remove_reference<int>::type&`
        Since `std::remove_reference<int>::type` is `int`, this becomes `int&`.

    - In the **rvalue overload**, the parameter type is:
      `typename std::remove_reference<int>::type&&`
      Which becomes `int&&`.
- **3. Evaluate the Argument:**
  The argument is `std::move(i)`, which returns an rvalue of type `int&&`.
- **4. Overload Resolution:**
    - The **lvalue overload** requires an argument of type `int&`. An rvalue (`int&&`) cannot bind to a non-const lvalue reference, so this overload is not viable.
    - The **rvalue overload** accepts an argument of type `int&&`, which perfectly matches the rvalue you provided.
- **5. Static Assertion Check in the rvalue Overload:**
  Inside the rvalue overload, there's a `static_assert(!std::is_lvalue_reference<T>::value, ...)`.
  Since T is `int` (not an lvalue reference), `std::is_lvalue_reference<int>::value` is `false`, so the assertion passes.
- **6. Forwarding:**
  The function then returns `static_cast<int&&>(t)`, effectively forwarding the argument as an rvalue.

- *How would you implement yourself the `sqrt()` function?*
    - Can be done pretty easily using the Newton-Raphson method:
        1. Set up the equation (remember that we are solving for y, not x)

$$y = \sqrt{x}$$
$$f(y) = y^2 - x = 0$$

        2. Find $f'(y)$

$$f'(y) = 2y$$

        3. Plug in the N-R formula $y_n = y_{n-1} - \frac{f(y)}{f'(y)}$

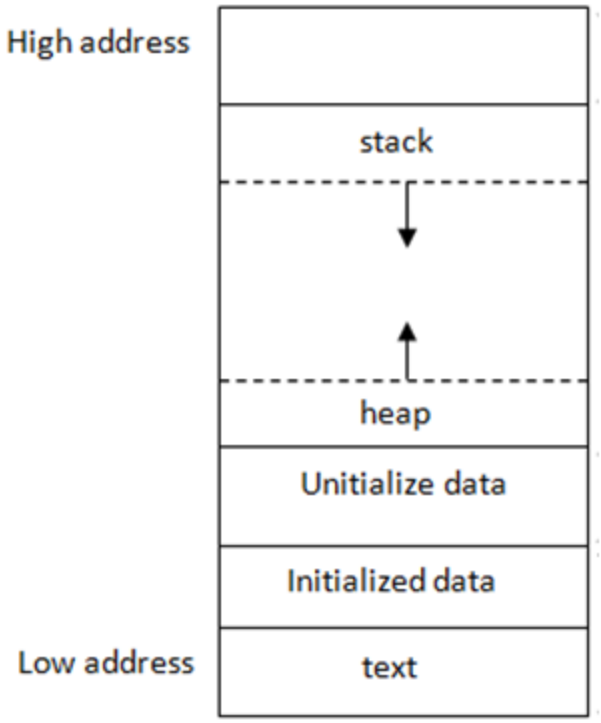
$$y = y - \frac{y^2 - x}{2y}$$
$$y = \frac{2y^2 - y^2 + x}{2y}$$
$$y = 0.5 * (y + \frac{x}{y})$$

4. Iterate several time to improve accuracy (e.g. 5 times)

- See the code here

- *What is a jump table?*
  - **A jump table** is a data structure—typically an array of pointers to code—that the program uses to quickly select one of many possible execution paths. Instead of using multiple conditional branches (like a long chain of if-else statements or a switch-case structure), the program computes an index based on some value (often an integer) and then uses that index to look up the address of the code to jump to in the table. This approach speeds up decision-making, especially when there are many cases, by reducing branch overhead and making use of direct indexing. Jump tables are commonly generated by compilers for switch-case statements when the case values are dense.

- *Which typically has a higher memory address: the stack or the heap?*
  - In general, stack-allocated variables have higher memory addresses than the heap. Additionally, the stack tends to grow downward while the heap grows upward. See the image below and the following link for more info.



- See this video for more info.

- *What is CRTP? Can you show how to implement it?*
  - **CRTP (Curiously Recurring Template Pattern)** is a C++ idiom where a class template takes its derived class as a template parameter. This pattern enables static polymorphism, allowing the base class to call functions defined in the derived class at compile time without the overhead of virtual functions. It enforces an interface and can lead to more optimized code by leveraging compile-time binding. Here is an example:

```
// Base class template using CRTP.
template <typename Derived>
class Base {
public:
    // A non-virtual interface that calls a derived class
implementation.
    void interface() {
    // Calls Derived's implementation() method.
    static_cast<Derived*>(this)->implementation();
}
    // A default method available to all derived classes.
    void defaultInterface() {
        std::cout << "Base default implementation\n";
    }
};

// Derived class inheriting from Base, using itself as the template
```

```
parameter.
 class DerivedClass : public Base<DerivedClass> {
public:
    void implementation() {
    std::cout << "DerivedClass implementation\n";
    }
 };

// Another derived class.
class AnotherDerived : public Base<AnotherDerived> {
public:
void implementation() {
    std::cout << "AnotherDerived implementation\n";
    }
};

int main() {
    DerivedClass d;
    d.interface(); // Calls DerivedClass::implementation()
    AnotherDerived ad;
    ad.interface(); // Calls AnotherDerived::implementation()
    return 0;
}
```

- *What does the following code snippet outputs?*

```
int f() { std::cout << "f"; return 0;}
int g() { std::cout << "g"; return 0;}
void h(std::vector<int> v) {}
int main() {
    h({f(), g()});
}
```

  - The goal of this question is to demonstrate that the evaluation order of elements in an initializer list is specified (as opposed to the arguments to a function call).
  - Within the *initializer-list* of a *braced-init-list*, the *initializer-clauses*, including any that result from pack expansions, are evaluated in the order in which they appear.
  - If `h` took two `int`s instead of a `vector<int>`, and was called like this: `h(f(), g());`
    the program would be unspecified, and could either print `fg` or `gf`.

- *If `class B` is a friend of `class A` and `class C` inherits from `class B`, is `class C` also a friend of `class A`? In other words, can you inherit friendliness?*
  - No it is not possible to inherit friends from a base class.

- *If you have a `const` method in a class, what are the ways to be able to modify member data inside that function (even though it is a bad practice)?*
  - The first way is simply to set the member in question as `mutable`. You will now be able to modify it inside the `const` method.
  - The second way is to take as argument of the function a pointer to the member variable(s) that need to be modify. You can then, within the `const` function, modify the given member data through the dereferenced pointer.

- *When should you pass an object by value or by reference? Should you pass primitive types by value or by reference?*
  - You should always pass an object by (const) reference, except if you plan on making a copy of it within the function. In that case, you should pass by value to avoid the additional memory read that comes from the address read.
  - Primitive types should always be passed by value since it requires only one memory read. If you pass it by reference or pointer, then first the address of the primitive will be read then the value at this address, leading two 2 memory read instead of one.

- *Can you give me 4 types of sorting algorithms? What are their worse case and best case complexity?*
  - **Insertion Sort**
    - **How It Works:**
      Builds the sorted array one element at a time by taking the next element and inserting it into the correct position in the already sorted portion.
    - **Best-Case Complexity:**
      O(n) – when the input is already nearly sorted (each new element is already in its proper place).
    - **Worst-Case Complexity:**
      $O(n^2)$ – when the input is in reverse order, requiring shifting every element for each insertion.
    - **Stable?** Yes.

- **Merge Sort**
  - **How It Works:**
    A divide-and-conquer algorithm that recursively splits the array into two halves, sorts each half, and then merges the sorted halves together.
  - **Best-Case Complexity:**
    O(n log n) – regardless of the initial order, since it always divides and merges in the same way.
  - **Worst-Case Complexity:**
    O(n log n) – the merge process dominates and remains the same irrespective of input order.
  - **Stable?** Yes.
- **Quick Sort**
  - **How It Works:**
    Also a divide-and-conquer algorithm that selects a pivot element, partitions the array so that elements less than the pivot come before it and elements greater come after, and then recursively sorts the partitions.
  - **Best-Case Complexity:**
    O(n log n) – when the pivot splits the array nearly evenly at every step.
  - **Worst-Case Complexity:**
    $O(n^2)$ – when the pivot is the smallest or largest element each time (e.g., sorted or reverse sorted input with a poor pivot choice).
  - **Stable?** Generally not, although some stable variants exist.
- **Bucket Sort**
  - **How It Works:**
    Distributes elements into several buckets (based on a range or hash function) and then sorts each bucket individually (often using another sorting algorithm), and finally concatenates the results. It works best when the input is uniformly distributed over a range.
  - **Best-Case Complexity:**
    O(n + k) – where k is the number of buckets; under ideal conditions (even distribution) the sorting inside each bucket is very fast, leading to nearly linear time.
  - **Worst-Case Complexity:**
    $O(n^2)$ – if the distribution is poor (e.g., most elements fall into one bucket), then that bucket may need to be sorted using a quadratic algorithm.
  - **Stable?** Generally not.

- *As we have seen before, Quick Sort has the same best time complexity as Merge Sort, but a worse time complexity. Does that mean Merge Sort is just a better sorting algorithm?*
  - Quicksort has $O(n_2)$ worst-case runtime and $O(log(n))$ average case runtime. However, it's superior to merge sort in many scenarios because many factors influence an algorithm's runtime, and, when taking them all together, quicksort wins out.
  - In particular, the often-quoted runtime of sorting algorithms refers to the number of comparisons or the number of swaps necessary to perform to sort the data. This is indeed a good measure of performance, especially since it's independent of the underlying hardware design. However, other things – such as locality of reference (i.e. do we read lots of elements which are probably in cache?) – also play an important role on current hardware. Quicksort in particular requires little additional space and exhibits good cache locality, and this makes it faster than merge sort in many cases.
  - In addition, it's very easy to avoid quicksort's worst-case run time of $O(n_2)$ almost entirely by using an appropriate choice of the pivot – such as picking it at random (this is an excellent strategy).

- *Can you allocate memory at compile time, for example using* `new` *?*
  - Yes, with C++20 and later, dynamic memory allocation (using new) is allowed in constant expressions (e.g., within constexpr functions), but with important restrictions. In such contexts, any memory you allocate at compile time must be deallocated before the constant evaluation finishes. This is because the compiler's constant evaluator simulates a "heap" for the purpose of constant expressions, and it must reclaim all allocations before the program is fully compiled. In other words, you can use new in a constexpr function if—and only if—the allocated memory is freed (using delete) within the same evaluation so that there are no lingering allocations, ensuring no memory leak occurs in the compile-time context.
  - See this.

- *What is transient allocation?*
  - **Transient allocation** means that you can allocate memory in a `constexpr` expression, but then the mem block **must be released** at the end of that expression. That way, the compiler can adequately track all the allocations, and I guess it's much easier to control and implement.

- *What is Two's Complement?*

- **Two's Complement** is a method for representing signed integers in binary that simplifies arithmetic by using a single system for both positive and negative numbers. In this system, positive numbers are represented as their ordinary binary values, while negative numbers are represented by inverting (flipping) all the bits of the positive value (the one's complement) and then adding 1, effectively computing $2^N - x$ in an N-bit system. This approach guarantees a unique representation for zero and enables the same hardware circuitry to perform addition and subtraction without needing special rules for negative numbers. To use two's complement, you decide on a fixed bit-width, convert your positive number to binary, invert the bits if the number is negative, add 1, and then use the resulting bit pattern in your calculations.

- *What does the compiler option `-ffast-math`?*
  - See this.

- *What does it mean to 'vectorize' a calculation?*
  - Vectorizing a calculation in C++ means restructuring your code to exploit data-level parallelism—processing multiple data elements with a single instruction—often using SIMD (Single Instruction, Multiple Data) instructions. Here are some details:
    1. **Auto-Vectorization by the Compiler:**
       Modern C++ compilers can automatically convert simple loops into vectorized code if the code is written in a “vectorization-friendly” style. This means:
       - The loop has no complex dependencies between iterations.
       - The data is stored contiguously (like in an array or a `std::vector`).
       - You enable optimization flags (e.g., `-O2` or `-O3` with GCC/Clang) that allow auto-vectorization. The compiler then uses CPU-specific SIMD instructions (such as SSE, AVX on x86 architectures) to process several elements in parallel.
    2. **Manual Vectorization Using Intrinsics:**
       You can also write explicit SIMD code using compiler intrinsics (e.g., `_mm_add_ps` for adding four floating-point numbers simultaneously on SSE-enabled processors). This approach gives you more control but is less portable and more complex. It involves:
       - Including headers like `<immintrin.h>`.
       - Ensuring that your data is properly aligned in memory (often with specialized allocators or alignment directives).

    - Writing code that uses SIMD data types (e.g., `__m128` for SSE) and intrinsics to perform arithmetic operations on these vectors.
3. **High-Level Libraries and Abstractions:**
   Libraries such as Eigen, Blaze, and others provide high-level abstractions for linear algebra that internally use vectorized operations. These libraries allow you to write code in a natural mathematical style, and they handle the underlying SIMD optimizations for you.
4. **Data Alignment and Memory Layout:**
   For vectorization to be effective, data should be laid out contiguously in memory and ideally aligned on boundaries (e.g., 16-byte alignment for SSE, 32-byte for AVX). This helps the processor fetch data in chunks and process multiple elements simultaneously.

- *What is False Sharing?*
    - False sharing occurs in multithreaded programs when multiple threads modify different variables that reside on the same cache line. Although the threads are working on separate data, the fact that these variables share the same cache line means that every write by one thread invalidates the cache line for others, forcing expensive cache coherence traffic. This can significantly degrade performance even though the data isn't logically shared between threads.
    - This concept is closely related to the one of **cache coherence**: cache coherence is a protocol and a set of techniques that ensure that multiple caches, which may store copies of the same memory location in a multiprocessor or multicore system, remain consistent with one another. In such systems, each processor might have its own local cache, so when one processor writes to a shared memory location, its cached copy changes. Without coherence protocols, other processors might continue reading stale data from their caches. Protocols like MESI (Modified, Exclusive, Shared, Invalid) are used to manage these states: they track which caches hold copies of a memory block and coordinate updates or invalidations when one cache modifies that block. This prevents problems such as reading outdated values and ensures that all processors have a consistent view of memory, albeit sometimes at the cost of additional overhead due to cache line invalidations or updates. See this for more info. Should also absolutely read the cache slides in HFT folder!

- *What is type erasure?*

- Type erasure is a technique that hides the concrete type details behind a uniform interface, allowing you to work with different types in a generic way without exposing their specific implementations. For example, consider a simplified version of a type-erased callable wrapper:

```
struct CallableBase {
virtual void call() = 0;
virtual ~CallableBase() = default;
};

template<typename F>
struct CallableImpl : CallableBase {
    F f;
    CallableImpl(F f) : f(f) {}
    void call() override { f(); }
};

class Function {
     std::unique_ptr<CallableBase> callable;
public:
    template<typename F>
    Function(F f) : callable(new CallableImpl<F>(f)) {}
    void operator()() { callable->call(); }
};

int main() {
    Function func = [](){ std::cout << "Hello, World!\n"; };
    func();
}
```

- See this.

- *What are some of the costs of type erasure?*
  - In C++, type erasure trades static information for flexibility, and that comes with several concrete costs. First, there is usually at least one extra level of indirection. A typical type-erased wrapper like `std::function`, `std::any`, or a hand-rolled “erased” interface holds a pointer to some control block or vtable plus a pointer to the actual object. Every call to the erased interface becomes something like “load vtable pointer, load function pointer from it, then indirect call,” which can inhibit inlining and makes branch prediction and instruction-level optimization harder. This can have a noticeable effect in hot loops compared to a templated, statically bound version where the compiler can inline and devirtualize everything.

- Second, there is often heap allocation. Many type-erased abstractions store the underlying object on the heap, because at compile time they do not know its concrete size. This adds allocation and deallocation overhead, potential contention if the allocator is shared, and more pointer chasing at runtime, which affects cache locality. Implementations can mitigate this with small-buffer optimization (SBO), but the general cost model is: large or dynamically sized erased values usually imply dynamic allocation, while the non-erased, templated version can often live entirely on the stack or inside contiguous containers.
- Third, there is a loss of static type safety and the optimizations that come with it. With type erasure you typically only know “this object models some interface” rather than its exact type. That means you cannot use the type system to enforce certain invariants specific to a concrete type, and you must rely on runtime checks or conventions. For example, `std::any_cast` can fail at runtime if you guess the wrong type; erased polymorphic wrappers can only expose what is in the erased interface, so any richer type information is gone and cannot be checked by the compiler. You also lose the ability to use features like `constexpr` or concept-constrained overload resolution on the concrete type at use sites, because those rely on knowing the actual type at compile time.
- So, summarizing in one sentence: type erasure in C++ tends to introduce extra indirection (hurting inlining and cache behavior), often requires heap allocation for the erased object, and reduces static type safety by hiding concrete types behind a minimal erased interface, shifting some errors and constraints from compile time to runtime.

- *What are the advantages/disadvantages of using `std::function` vs a function pointer?*
  - **Advantages of std::function:**
    - **Flexibility:** It can hold any callable object (functions, lambdas, functors, etc.) that matches the specified signature, not just plain function pointers.
    - **Type Erasure:** It hides the concrete type of the callable, allowing for a uniform interface and easier code abstraction.
    - **Copyability and Assignability:** std::function objects can be copied and assigned, making them useful in generic programming and for storing callbacks.
  - **Disadvantages of std::function:**
    - **Overhead:** Due to type erasure and potential dynamic memory allocation, std::function generally incurs more runtime overhead than a raw function pointer.
    - **Performance:** For performance-critical code, the extra indirection and overhead may be a disadvantage compared to the simplicity and potential inline-ability of function pointers.

- **Function Pointers:**
  - **Lightweight:** They are typically a single pointer with minimal overhead.
  - **Fast:** They can be more efficient, particularly in tight loops or performance-critical sections, since there's no extra indirection or dynamic allocation.
  - **Limited Flexibility:** They can only store plain functions with a matching signature and cannot hold lambdas with captures or functors unless converted appropriately.
- See this.

- *What does the following code outputs?*

```
std::vector<int> vec{1, 2};
auto start = vec.begin();
vec.push_back(3);
std::cout << *start;
```

  - This is undefined behaviour and outputs garbage. Once 3 is pushed to the vector, the vector needs to reallocate memory since the size gets bigger than the available capacity on the `push_back` call. Reallocation leads to the invalidation of the `start` iterator, which now points to uninitialized memory.

- *What is memory ordering in C++?*
  - Memory ordering in C++ is all about controlling how and when memory operations (loads and stores) performed by different threads become visible to each other. Since modern compilers and CPUs often reorder instructions for performance, the C++ memory model provides a set of guarantees through atomic operations that let you specify the level of ordering you need. Here are the key concepts in more detail:

    - **memory_order_relaxed:**
      This ordering ensures that the operation is atomic (i.e., it won't be torn or observed in a half-completed state) but does not impose any ordering constraints on surrounding memory accesses. It's useful when you only need a simple atomic counter or flag and you don't care about the sequence in which operations occur relative to other memory accesses.
    - **memory_order_acquire:**
      Typically used with atomic loads, this ordering prevents any subsequent memory operations from being reordered before the load. In essence, once a thread reads

a value with acquire semantics, all the memory writes that happened before a corresponding release (in another thread) are guaranteed to be visible.

- **memory_order_release:**
  Used with atomic stores, this ensures that all previous memory writes in the same thread are completed before the store is performed. When a thread writes with release semantics, it “releases” its prior changes so that another thread reading that atomic variable with acquire semantics can see all those changes.
- **memory_order_acq_rel:**
  This ordering is a combination used for read-modify-write operations. It acts like a release when storing and like an acquire when loading. It prevents the reordering of operations both before and after the atomic operation, ensuring a strong barrier around the modification.
- **memory_order_seq_cst (Sequentially Consistent):**
  This is the strongest ordering guarantee. It enforces a single global order of all sequentially consistent operations, meaning all threads observe these operations in the same order. Although the simplest to reason about, it can be more restrictive performance-wise compared to weaker orderings.

• **Release-Acquire Semantics:**
A common synchronization pattern is to have one thread perform a store with release semantics and another thread perform a corresponding load with acquire semantics on the same atomic variable. This pairing establishes a “happens-before” relationship—ensuring that all memory writes before the release are visible after the acquire. For example, if Thread A writes to several variables and then does a release store, and Thread B later reads that atomic variable with acquire, then Thread B is guaranteed to see all of Thread A’s prior writes.

• **Relaxed Ordering and Performance:**
When you use memory_order_relaxed, you get atomicity without any ordering guarantees. This means that while the operation itself won’t be interleaved with another thread’s operation, other memory accesses may be reordered around it. Relaxed ordering can lead to performance improvements on some architectures, but it places the burden on the programmer to ensure that the lack of ordering won’t introduce subtle bugs.

• **Sequential Consistency:**
Memory_order_seq_cst is intuitive because it forces a single, total order on all operations marked with this memory order. Every thread sees the same order of these operations, which makes reasoning about program behavior easier. However, because it restricts reordering more strictly than other memory orders, it can sometimes incur performance penalties.

• Balancing Performance and Correctness:
The choice of memory ordering is a trade-off. Stronger memory orders (like seq_cst) are simpler to reason about and ensure maximum consistency, but they might limit certain compiler and hardware optimizations. Weaker memory orders (like relaxed) offer performance advantages but require careful design to avoid subtle bugs due to unexpected reordering.

- See this for more info.

- *What does `std::lock` ?*
  - `std::lock` is a utility function provided by the C++ Standard Library (in the `<mutex>` header) that locks multiple mutexes at once while avoiding deadlock. Instead of locking each mutex one by one (which might lead to deadlock if another thread locks them in a different order), `std::lock` takes a variable number of lockable objects (such as mutexes) and attempts to lock them all simultaneously.
  - When locking multiple mutexes manually (e.g., calling `m1.lock()` followed by `m2.lock()`), there's a risk of deadlock if another thread locks them in the opposite order. `std::lock` eliminates this problem by ensuring that the locks are acquired in a safe, deadlock-free manner.
  - `std::lock` employs a deadlock avoidance algorithm. This means that if it cannot acquire all the locks immediately, it will release any locks it has already taken and try again. This prevents the scenario where two threads are each holding one lock and waiting indefinitely for the other.
  - A related construct is the `std::scoped_lock`, which is a convenient RAII wrapper that manages the lifetime of one or more mutex locks, automatically unlocking them when the object is destroyed. It leverages `std::lock` internally when locking more than one mutex.

- *What is a cache line and how do you ensure that an array starts at a specific cache line?*
  - You can ensure that an array is aligned to a specific boundary so that its first element starts at that boundary. In C++11 and later, you can use the `alignas` specifier. For example, if you want the array to be aligned to a 64-byte boundary (common for many CPU cache lines), you can declare it as follows:

```
alignas(64) int myArray[10];
```

  - This guarantees that `myArray[0]` starts at an address that is a multiple of 64, ensuring that the cache line containing `myArray[0]` also contains the following

elements (depending on how many fit in 64 bytes) and does not include any preceding unrelated data.

- For dynamic memory allocation, you can use the aligned new operator (available in C++17) like this:

```
    int* myArray = new (std::align_val_t(64)) int[10]; // Use the
array...
    delete[] myArray;
```

- These techniques let you control the alignment of your array, which can be critical for optimizing performance and preventing issues like false sharing in multithreaded applications.

- *Is it possible to prevent an object to be allocated on the stack?*
  - By making the destructor non-public (e.g., protected or private), you prevent automatic (stack) allocation because the compiler won't be able to generate code to destroy the object when it goes out of scope. Instead, you provide a public static factory function that allocates the object on the heap (using new) and returns a pointer (often wrapped in a smart pointer).

```
class HeapOnly {
public:
    static std::unique_ptr<HeapOnly> create() {
        return std::unique_ptr<HeapOnly>(new HeapOnly());
    }

protected:
    ~HeapOnly() = default; // Protected destructor prevents stack
allocation.
private:
    HeapOnly() = default;
};

int main() {
    // HeapOnly obj; // ERROR: cannot instantiate on the stack.
    auto ptr = HeapOnly::create(); // OK, allocated on the heap.
    return 0;
}
```

- *What happens if you create a `std::vector` whose size is greater than what is available in RAM? What about an `std::array` ?*
    - When you try to allocate a vector (or any dynamic memory) that requires more memory than is available, several things can happen depending on your system and configuration:

    1. **Standard Behavior (Throwing std::bad_alloc):**
       In most C++ implementations, when you request memory using operator new (which std::vector does internally), and if the allocation request exceeds available RAM (or the process's address space limits), the operator new will fail and throw a std::bad_alloc exception. This exception signals that the allocation couldn't be satisfied.
    2. **Memory Overcommitment (OS-specific Behavior):**
       Some operating systems (like Linux with overcommit enabled) may allow an allocation request to succeed even if the total requested memory exceeds the actual physical RAM. In these systems, the OS might overcommit memory and only actually back the allocation when you access it. If you later try to use that memory and there isn't enough physical or swap space, the process might be terminated (e.g., by the OOM killer on Linux) or experience a segmentation fault.
    3. **Performance and Paging:**
       Even if your allocation doesn't immediately fail (or if you're close to the limits), excessive memory usage can lead to heavy paging (swapping data to and from disk) which dramatically slows down your program.

    - An `std::array` is allocated on the stack. If you create an `std::array` whose size is greater than the stack size (typically, a stack is between 1 and 12mb), then you get stack overflow.

- *Does the following code snippet compiles?*

```
auto square(int x) { return x*x; }
int main () {
    Matrix<square(3), square(3)> m;
}
```

    - No, it does not compile. Remember that non-constexpr function calls are evaluated at runtime. Hence, you cannot use a the returned value of a non-constexpr function as a template parameter, since those are evaluated at compile time.

- *When generating random numbers using the `<random>` library, why do we generally use both a random device and a pseudo-random generator like `std::mt19937` together? See*

the code snippet.

```
int main()
{
    std::random_device rd;
    std::mt19937_64 mt{rd()};
    std::uniform_int_distribution unif(0, 10);
    int x =  unif(mt{});
}
```

- `std::random_device` is used to obtain a non-deterministic (or less predictable) seed value, whereas `std::mt19937` (the Mersenne Twister) is a fast pseudo-random number generator. In other words:
  - **std::random_device:**
    Provides high-quality randomness from hardware or OS sources. However, it's relatively slow and may have limited throughput. It's primarily used for seeding.
  - **std::mt19937:**
    Once seeded, this generator produces a deterministic sequence of random numbers quickly. It's much faster than calling std::random_device repeatedly and is suitable for most simulation and general-purpose uses.
- Thus, the common pattern is to use `std::random_device` to generate a seed, and then pass that seed to `std::mt19937`, which then efficiently produces a long sequence of random numbers. See this thread and this article for more info.

- *Why is it better for assignment operators to return an Object reference rather that something else like void (both work)?*
  1. **Enabling Chained Assignments:**
     By returning a reference to the assigned object, you allow statements like `a = b = c;` to work correctly. In this chain, `b = c` returns a reference to `b`, which is then assigned to `a`.
  2. **Consistency with Built-in Types:**
     Built-in types in C++ (such as `int`, `double`, etc.) return the left-hand operand after assignment. Custom types should mimic this behavior for intuitive and predictable use.
  3. **Efficiency:**
     Returning a reference avoids unnecessary copying of objects. It provides direct access to the modified object without the overhead of creating a new object instance.
  4. **Fluent Interface and Expression Compatibility:**
     Returning a reference allows the assignment to be part of a larger expression, making your class more flexible in usage.
     In contrast, if the assignment operator returned `void`, you would lose these benefits,

particularly the ability to chain assignments, which can make the code less natural to write and harder to maintain.

- *What are the differences between fully-specialized templates and partially-specialzied templates? Can you show some examples?*
  - A **fully specialized template** (or complete specialization) provides an implementation for a template when all template parameters are exactly specified. This means that the specialization applies only when the template is instantiated with that exact type or set of types.

```
// Primary template
template <typename T>
struct Printer {
    static void print() {
        std::cout << "General template for type T\n";
    }
};

// Full specialization for int
template <>
struct Printer<int> {
    static void print() {
        std::cout << "Specialized for int\n";
    }
};

int main() {
    Printer<double>::print(); // Uses the general template
    Printer<int>::print();    // Uses the fully specialized template
    return 0;
}
```

  - A **partially specialized template** provides a specialization for a subset of types that match a particular pattern. Unlike full specialization, some template parameters remain as parameters (possibly deduced), making the specialization applicable to a broader range of types. Note that **partial specialization is allowed only for class templates, not function templates.**

```
// Primary template
template <typename T>
struct TypeInfo {
    static void print() {
        std::cout << "General type\n";
```

```
    }
};

// Partial specialization for pointer types
template <typename T>
struct TypeInfo<T*> {
    static void print() {
        std::cout << "Pointer type\n";
    }
};

int main() {
    TypeInfo<int>::print();    // Uses the primary template: "General
type"
    TypeInfo<int*>::print();   // Uses the partial specialization:
"Pointer type"
    return 0;
}
```

- *Why is the following code snippet not correct?*

```
int main()
{
    char a = 'A';
    std::cout << &a;
    std::cout << (char *)&a;
}
```

  - The issue is that when you output a pointer to a char (i.e. a `char*`) using `std::cout`, the stream treats it as a C-style string, not as an address. Since the memory pointed to by `&a` is not necessarily null-terminated, this leads to undefined behavior.
  - Both `&a` and `(char *)&a` have the type `char*`. The stream's overload for `char*` expects a null-terminated string, so it will try to print characters starting at that address until it finds a `'\0'`. In this case, since there's no guarantee of a null terminator after `a`, you might see garbage output or run into a runtime error.

- *Is it possible to initialize a const variable with a switch statement?*
  - By default, no it is not possible. However, you can still do it using an immediately invokable lambda function. See below.

```
const int id = 1;
const std::string name = [](int id)
{
    switch(id):
        case 0: return "Zero";
        case 1: return "One";
        default: return "Unrecognized";
}(id);
```

- *What is the ouput of the following code snippet?*

```
int x = 4, y = 1;
std::cout << x+++++y << std::endl;
```

  - When the compiler sees `x+++++y`, it doesn't interpret it as "x + (++(+y))" or any kind of cleverly nested addition; instead, the lexer breaks it into the tokens `x ++ ++ + y`. This is called the maximum munch principle.
  - Because `++` is a valid operator, it greedily takes two plus signs at a time. That means the parser tries to form `(x++) ++ + y`. Here, it first applies the postfix increment `x++` (which yields a prvalue) and then immediately tries to apply another `++` to that prvalue. Since the result of `x++` is not an lvalue (you can't increment the temporary value returned by `x++`), the compiler issues an error such as "lvalue required as increment operand." In short, the token stream `x ++ ++ + y` leads to an invalid attempt to apply `++` to something that isn't a modifiable lvalue, so the code won't compile.
  - See the compiler output here.

- *How would you check if a number is a power of 2?*

```
bool isPowerOfTwo(unsigned int n) {
  return n > 0 && (n & (n - 1)) == 0;
}

// C++20: std::has_single_bit
bool isPowerOfTwo_cpp20(unsigned int n) {
    return std::has_single_bit(n);
    }

int main() {
    for (unsigned int x : {0u,1u,2u,3u,4u,16u,18u}) {
```

```
        std::cout << x
                  << (isPowerOfTwo(x) ? " is " : " is not ")
                  << "a power of two\n";
        }
    }
```

- If `n` is a power of two (say 10000_2), then `n-1` is all 1's in the lower bits (01111_2), so `n & (n-1)` is zero.
- If `n` has more than one bit set, then `n & (n-1)` will clear the lowest "1" bit but leave something nonzero.

- *In a open addressing hash map, why a lot of implementation use a power of 2 for the table size? How does it compare to choosing a prime number?*
  - Using a power-of-two table size (i.e. $M=2^k$) is very common—especially in languages like Java or C where you want to do `index = hash(key) & (M - 1)` instead of a slower `mod` operation. But it comes with its own trade-offs:
    1. **Fast index computation**
       Masking out the low k bits (`& (M-1)`) is typically several times faster than a full integer division for `mod M`. This is why many implementations that care deeply about constant factors choose power-of-two sizes.
    2. **Reliance on good hash mixing**
       If your hash function only "sprays" entropy into the high bits of its result, then truncating to the low k bits (via bitmask) can yield awful clustering. In contrast, when M is prime, even a mediocre hash will have its low and high bits folded together by the modulo operation, smoothing out bit-patterns.
       - **With primes**:
         `index = hash(key) % M`
         mixes *all* bits of `hash(key)` into the result.
       - **With powers of two**:
         `index = hash(key) & (M-1)`
         uses *only* the lowest k bits. If your hash doesn't randomize those well, you get hot spots.
    3. **Double hashing constraints**
       For double hashing you need `gcd(step, M) = 1` so your probe sequence can visit every bucket.
       - **If M is prime**, any non-zero step < M is automatically coprime.
       - **If $M = 2^k$**, you must ensure your secondary hash function always returns an *odd* number (so it isn't divisible by 2), otherwise the sequence can only cover a subset of the table.

4. **Resizing semantics**
   When you grow a power-of-two table, you typically *double* its size—an O(n) rehash but easy to compute (just bump k→k+1). With primes, you need to pick the "next" prime (e.g. 2^k+1 or something), which is more work but can keep your load factors a bit tighter if you vary your growth factor.

- To summarize:
  - **Use power-of-two** if
    - You already have or can afford a high-quality hash function with full bit mixing (e.g. MurmurHash3, xxHash, CityHash, the built-in Java/C# hashes after their "spread" step).
    - You want the fastest possible index computation and are willing to guarantee the secondary hash is odd (for double hashing) if you go that route.
  - **Use primes** if
    - You're implementing your own simple hash (e.g. `h = key % some_base` or a low-quality polynomial rolling hash) and you can't guarantee uniformity of low bits.
    - You prefer the mathematical simplicity of "any non-zero" step being coprime.
  - In practice, many high-performance libraries choose *power-of-two* sizes but pair them with a bit-scrambling step that guarantees all bits of the raw hash influence the low k bits. If you use that pattern, you get both speed **and** good distribution.

- *Can you show an example of why using a power of two for the size of the table in a open addressing hash table avoid the modulus operation?*
  - Suppose:
    - Our table size is $M = 8 = 2^3$.
    - The mask is $M - 1 = 7 = 0b111$.
  - Now say your hash function produces the 8-bit value:

```
hash(key) = 0b11010110  (214 decimal)
```

1. **Modulo computation**

$$\text{index}_{\%} = \text{hash}(key) \bmod 8 = 214 \bmod 8 = 6 = 0b110$$

2. **Bitmask computation**

```
index_& = 0b11010110
        & 0b00000111
        ------------
```

```
        0b00000110
        = 0b110 = 6
```

- Because 2^3 in binary is `1` followed by three zeros, taking “mod 8” simply keeps the lowest 3 bits of the hash. Masking with `0b111` does exactly that—zeroes out all bits above bit-2, leaving bits 0–2 untouched.

- *Are you allowed to have a virtual method inside a templated class?*
  - Yes—there’s nothing in C++ that forbids you from declaring virtual functions in a class template. In fact a template class works just like any other class in that respect. For example:

```cpp
// A class template with a pure virtual method
template<typename T>
struct Processor {
    virtual ~Processor() = default;
    virtual void process(const T& item) = 0;
};

// A concrete instantiation for
int struct IntProcessor : Processor<int> {
    void process(const int& item) override { std::cout << "Processing
int: " << item << "\n"; }
};
```

  - Here `Processor<T>` is a template, but its member function `process` is virtual (and even pure). You can instantiate multiple specializations (`Processor<int>` , `Processor<std::string>` , etc.), each of which participates in the normal C++ v-table mechanism.
  - What you **cannot** do is declare a *templated* virtual function.

```cpp
struct Bad {
// ERROR: virtual functions cannot themselves be templates
template<typename U>
virtual void foo(const U& u);
};
```

  - That is ill-formed because the language doesn’t support virtual templates. If you need a family of overloads that behave polymorphically, you must either
    1. Write non-templated virtual overloads (e.g. one `foo(int)` , one `foo(std::string)` , etc.), or
    2. Use a single virtual function that takes a type-erased parameter (e.g. `virtual void foo(std::any);`), or

3. Combine templates and runtime polymorphism by templating the class (as shown above), not the virtual method.

- *Can you explain what is a memory pool and why it is useful?*
  - A **memory pool** (or “pool allocator”) is a custom allocator that grabs one large block of raw memory up front, then parcels it out in fixed-size chunks to your objects, rather than calling `new` or `malloc` every time. The pool typically keeps a free-list of those chunks so that both allocation and deallocation are just pointer-hops—constant-time operations with no bookkeeping, no fragmentation, and no calls into the OS.
  - Why use a memory pool?
    1. **Blazing allocation/deallocation speed**
       Standard heap allocators have to maintain metadata, merge adjacent free blocks, handle fragmentation, and ultimately call into the OS when they can’t satisfy a request. In contrast, a pool’s free-list is usually just a singly-linked list of free slots:

       ```
       // allocate:
       Slot* s = freeList;
       freeList = freeList->next;
       return s;
       // deallocate:
       s->next = freeList;
       freeList = s;
       ```

       Two pointer-stores, no loops, no locks (in a thread-local pool), no system calls.
    2. **Predictable, bounded latency**
       In real-time or low-latency systems (like HFT), you cannot afford the jitter that comes from an unpredictable `malloc`. Pools give you hard O(1) timings and you can reserve exactly enough memory at startup to avoid any further heap interactions.
    3. **Avoid fragmentation**
       By carving your block into uniform slots, you never end up with tiny unusable gaps in the heap. You know exactly how many objects you can hold, and you’ll never fail an allocate until your pool is exhausted.
    4. **Better cache locality**
       When you allocate many objects of the same size from a contiguous region, those objects tend to live close together in memory—improving data-cache performance when you scan or iterate them.
  - In short, a memory pool trades a little upfront complexity and fixed memory footprint for extremely fast, predictable allocations—exactly the qualities you need when every microsecond (or nanosecond) counts.

- *Could you give the approximate number of cpu cycles taken to execute common operations (like +, %, division, etc).*
  - Here's a rough "cheat-sheet" of latencies (in clock-cycles) for common operations on modern x86-64 CPUs (e.g. Intel Haswell/Skylake family). These are **latencies** in a dependency chain; actual throughput can be higher for fully-pipelined ops.

Operation	Approx. Cycles	Notes & Sources
Integer		
Add / Sub	1 cycle	Latency of simple ALU ops is 1 cycle on all recent x86 CPUs superuser.com
Multiply (`IMUL`)	3 cycles	Fully-pipelined on many CPUs (throughput = 1 per cycle) superuser.com
Divide / Modulo (`DIV` / `IDIV`)	~30 cycles (32-bit) 42–95 cycles (64-bit)	Integer division is microcoded and slow: Haswell ≈ 30 cycles; Skylake DIVQ ≈ 42–95 cycles forum.nasm.usstackoverflow.com
Bitwise AND/OR/XOR	1 cycle	Same simple-ALU latencies superuser.com
Shift (`SHL` / `SHR`)	1 cycle	Single-cycle barrel shifter superuser.com
Floating-Point / SIMD		
FP Add / Sub	3 cycles	Typical FADD latency ≈ 3 cycles on Haswell; throughput ≈ 0.5–1 cycle superuser.com
FP Multiply	5 cycles (Haswell) 3 cycles (other)	FMUL has ≈ 5 cycle latency on Haswell, though many older designs list ≈ 3 cycles superuser.com
FP Divide	20–40 cycles	Similar scale to integer divide, though exact depends on width and μarch; microcoded and high-latency
Memory & Caches		
Register access	0 cycles (free)	Values in registers are "free" beyond the ALU latency of the op

Operation	Approx. Cycles	Notes & Sources
L1 load	1-3 cycles	L1 d-cache load-use latency on Haswell & newer is ≈ 4 cycles stackoverflow.com
L2 load	4-10 cycles	L2 cache hit ≈ 10 cycles stackoverflow.com
L3 load	30-40 cycles	L3 local hit ≈ 40 cycles stackoverflow.com
DRAM load (RAM)	~200 cycles	Main-memory latency ≈ 60 ns → ≈ 180–200 cycles at 3 GHz stackoverflow.com
Control Flow		
Branch (correctly predicted)	1 cycle	Predicted branches retire in ≈ 1 cycle stackoverflow.com
Branch (mispredicted penalty)	12–20 cycles	Typical mispredict penalty ≈ 12 cycles (Haswell) up to ≈ 20 on older μarches; can be > 10 cycles stackoverflow.comstackoverflow.com

---

- *Are there cases where passing-by-value might be better than passing-by-reference?*
  - Consider the following case:

```
// Argument s is a const reference
std::string str_to_lower(const std::string& s) {
    std::string clone = s;
    for (char c: clone)
        c = std::tolower(c);
    return clone;
}
// Argument s is an rvalue reference
std::string str_to_lower(std ::string&& s) {
    for (char c: s)
        c = std::tolower(c);
    return s;
}
```

  - Here, we define two functions: one for const lvalues, and one for rvalues. However, we could just create one function that takes the argument by value instead, and obtain the same behaviour:

```
std::string str_to_lower(std::string s) {
    for (char c: s)
```

```
        c = std::tolower(c);
    return s;
}
```

- Now, if we perform one of the two following operations:

```
std::string str = std::string{"ABC"}; // #1
str = str_to_lower(str);

std::string str = std::string{"ABC"}; // #2
str = str_to_lower(std::move(str));
```

- When passed a regular variable, shown as above (#1), the content of `str` is copy-constructed into `s` prior to the function call, and then move-assigned back to str when the functions returns. When passed an rvalue, as shown above (#2), the content of `str` is move-constructed into `s` prior to the function call, and then move-assigned back to str when the function returns. Therefore, no copy is made through the function call. Thus, we see that in both cases, we do get the wanted behaviour, without having to declare 2 functions.
- Note however that this pattern does not always work. Consider the following class:

```
class Widget {
    std::vector<int> data_{};
    // ...
public:
    void set_data(std::vector<int> x) {
        data_ = std::move(x);
    }
};
```

- Assume we call set_data() and pass it an lvalue, like this:

```
std::vector<int> v = std::vector<int>{1, 2, 3, 4};
widget.set_data(v); // Pass an lvalue
```

- Since we are passing a named object, `v` , the code will copy-construct a new `std::vector` object, `x` , and then move-assign that object into the `data_` member. Unless we pass an empty vector object to `set_data()` , the `std::vector` copy-constructor will perform a heap allocation for its internal buffer. Now compare this with the following version of set_data() optimized for lvalues:

```
void set_data(const std::vector<int>& x) {
    data_ = x; // Reuse internal buffer in data_ if possible
}
```

- Here, there will only be a heap allocation inside the assignment operator if the capacity of the current vector, `data_` , is smaller than the size of the source object, `x` . In other

words, the internal pre-allocated buffer of `data_` can be reused in the assignment operator in many cases and save us from an extra heap allocation. If we find it necessary to optimize `set_data()` for lvalues and rvalues, it's better, in this case, to provide two overloads:

```
void set_data(const std::vector<int>& x) {
    data_ = x;
}
void set_data(std::vector<int>&& x) noexcept {
    data_ = std::move(x);
}
```

- The first version is optimal for lvalues and the second version for rvalues

---

- *Why is the use exceptions avoided by many programmers? Is there a cost to throwing an exception? What if you never throw, is there still a cost?*
  - Exceptions are avoided by some programmers primarily because of concerns about performance and predictability. These concerns can be broken down into three distinct cases: cost of throwing, cost when not throwing, and binary size.
  - When an exception is **actually thrown and caught**, the runtime cost is significant. Unlike a normal function return or branch, exception handling requires unwinding the stack, destroying local objects with active lifetimes, and transferring control to a handler. This process is relatively slow, non-constant in time, and hard to predict. In contexts with strict real-time constraints, this unpredictability makes exceptions unsuitable, since you cannot guarantee bounded execution time.
  - When an exception is **not thrown**, the performance story is much better. Modern compilers implement exception handling so that the *success path* (the “no exception” path) is essentially free in terms of runtime overhead. In fact, code using exceptions often performs better in the success path than code using error codes, since error-code style requires explicit condition checks and branches in every call site. However, the downside is that even if you never throw, exceptions still increase the **binary size** of the program. The compiler and runtime must emit metadata tables to describe where exceptions can be caught and how stack unwinding should proceed. This violates the zero-overhead principle, since some cost is paid even if exceptions are unused, though in practice this is usually only a matter of program size rather than runtime speed.
  - Thus the short answers are: yes, throwing and catching exceptions has a cost, and that cost is both high and unpredictable. If you never throw, there is still a cost, but it is a compile-time/binary-size cost rather than a runtime cost. The zero-cost abstraction principle is not strictly satisfied, but in most applications the penalty is negligible

compared to the clarity and expressiveness that exceptions provide. For performance-critical or real-time systems, exceptions may still be unsuitable, but in general-purpose programming they perform outstandingly on the success path, with the key guideline being that exceptions should remain rare, used only for truly exceptional situations.

- *What are the advantages of string_view vs a basic string? What if you want to take a substring of a string?*
  - The key advantage of `std::string_view` over `std::string` is that it provides a **non-owning, lightweight view** into an existing character sequence. Unlike `std::string`, which manages its own memory and requires allocations and copies when you create substrings, a `string_view` is just a pair of pointers (or a pointer plus a size). This means you can cheaply refer to any part of a string—or even to a string literal—without copying the data. For operations like parsing, slicing, or passing around read-only text, `string_view` is significantly more efficient because it avoids unnecessary memory management overhead.
  - When you want a substring of a `std::string`, using `substr()` creates a brand new `std::string` object that owns its own buffer, including its own null terminator. By contrast, `string_view::substr()` simply adjusts the start pointer and the length, so it is essentially constant-time and does not allocate or copy. However, because `string_view` is non-owning, you must ensure that the underlying data outlives the view—otherwise you end up with a dangling reference.
  - One subtle point concerns the null terminator. A `std::string` guarantees that its internal buffer is null-terminated, so you can safely call `.c_str()` and use it with C APIs. A `std::string_view` does not guarantee a trailing null character at the end of its view; in fact, it may be viewing just the middle of a larger string. This means you cannot safely treat a `string_view` as a C-style string unless you explicitly ensure that its segment ends with `'\0'`. Instead, you should use its `.data()` and `.size()` members for safe access, treating it as a buffer rather than a null-terminated string.
  - In summary, `string_view` excels in efficiency and convenience when slicing or referencing strings without copying, while `std::string` is the safer choice when you need ownership, mutability, or guaranteed null termination. If you want to take substrings frequently in performance-sensitive code, `string_view` provides a major advantage by eliminating allocations, but you must be careful with lifetimes and cannot rely on a built-in null terminator.

- *Is Meyer's Singleton thread-safe?*

- Yes, Meyers' Singleton is thread-safe in C++11 and later. The C++ standard guarantees that the initialization of a function-local `static` variable is performed exactly once, even when multiple threads attempt to initialize it concurrently. This means that if two or more threads call the singleton's `instance()` function at the same time, the compiler will ensure that the object is constructed only once in a thread-safe manner, and any other threads will block until initialization is complete. For example:

```
class Singleton {
public:
    static Singleton& instance() {
            static Singleton s; // thread-safe initialization since
C++11         return s;     }
private:
    Singleton() = default; };
```

- In this pattern, no additional synchronization (like `std::mutex` or `std::call_once`) is required for construction. However, it's important to note that while the creation of the singleton object itself is thread-safe, the **methods** of the singleton are not automatically thread-safe—you must still handle synchronization if multiple threads will modify shared state within the singleton.

- *What is `std::call_once` ?*
  - `std::call_once` is a C++ standard library utility that ensures a piece of code is executed exactly once, even if multiple threads attempt to run it simultaneously. It works together with a `std::once_flag`, which keeps track of whether the code has already been executed. When multiple threads call `std::call_once` with the same flag, only one thread will run the provided function or lambda, while the others will wait until that execution finishes. This is especially useful for one-time initialization of resources, such as setting up a global object or initializing a library. For example:

```
std::once_flag flag;

void init() {
    std::call_once(flag, [] {
        std::cout << "Initialized once\n";
    });
}

int main() {
    std::thread t1(init);
    std::thread t2(init);
```

```
    std::thread t3(init);
    t1.join();
    t2.join();
    t3.join();
}`
```

- In this example, even though three threads call `init()`, the message “Initialized once” is printed only once. `std::call_once` is a clean, efficient way to perform thread-safe one-time initialization without manually managing locks or checks.

- *What is a byte ? Are bytes always 8 bit?*
  - A **byte** is the smallest *addressable* unit of storage on a machine—the amount of memory you can read/write with one address. In most modern systems, a byte is **8 bits**, which is enough to represent values 0–255.
  - Are bytes *always* 8 bits? Historically, no. Older or specialized machines used 6-, 7-, 9-bit “bytes.” Programming languages and standards reflect this: in C/C++, a “byte” is the size of `char`, and `CHAR_BIT` is **at least 8**, not guaranteed to be exactly 8 (though it is on virtually all mainstream CPUs today). In networking standards, the term **octet** is used when exactly **8 bits** is required.
  - So: practically everywhere today a byte = **8 bits**; conceptually, a byte is the smallest addressable unit and **need not be 8** on all conceivable architectures.

- *Is the following code snippet correct?*

```
char buf[11];
std::memset(buf, ' ', sizeof(buf));
std::string hello = "hello";
std::memcpy(buf, hello.data(), hello.size());
std::memcpy(buf + hello.size() + 1, "world", 5);
std::string str{buf};
```

  - The code builds an 11-byte `char` buffer, fills it with spaces, copies `"hello"` into positions `0..4`, leaves a space at position `5`, and copies `"world"` into positions `6..10`, producing the byte sequence `h e l l o _ w o r l d` with **no null terminator (`'\0'`)** anywhere in the buffer. The final line constructs `std::string str{buf};`, which expects `buf` to be a **null-terminated C string**; because it isn’t, the constructor reads past the end of `buf` looking for `'\0'`, invoking **undefined behavior**. To be correct you must either append a terminator (and have room for it) or construct the `std::string` with an explicit length. Here is a corrected version:

```
char buf[11];
std::memset(buf, ' ', sizeof(buf));
std::string hello = "hello";
std::memcpy(buf, hello.data(), hello.size());
std::memcpy(buf + hello.size() + 1, "world", 5);
std::string str(buf, sizeof buf);  // use explicit length; no '\0' required
```

- (Alternative: make `buf[12]` and set `buf[11] = '\0';` then keep `std::string str{buf};`.)

- *In C++, are string literals null-terminated?*
  - Yes— `std::string("Hello")` is perfectly fine. In C++, a string literal like `"Hello"` has type `const char[6]` and **always includes a terminating null character** (`'\0'`) at the end; that's why `sizeof("Hello") == 6` while `std::strlen("Hello") == 5`. When you pass `"Hello"` to the `std::string` constructor, the array decays to `const char*` and the constructor reads characters up to the first `'\0'`, producing a `std::string` with value `"Hello"`. This is exactly the case the constructor is meant for, so it's safe and idiomatic.

- *Can you use `free()` to free the memory allocated by `new`? What about `::operator new`?*
  - No—**never mix them.** In C++, the allocator and deallocator must **match**:
    - Memory obtained by a **`new` expression** (e.g., `new T` / `new T[n]`) must be released with **`delete`** / **`delete[]`**, respectively. Using `free()` on such pointers is **undefined behavior** (and the reverse— `delete` on `malloc` —is also UB).
    - The `new` expression actually does two steps: it calls an **allocation function** (`::operator new` or `::operator new[]`) to get raw storage, then **constructs** the object(s). The matching **deallocation function** (`::operator delete` / `::operator delete[]`) frees that storage after destructors run.
    - If you call the **global allocation function directly**— `void* p = ::operator new(n);` —you must free it with the **matching global deallocation**: `::operator delete(p);`. (If you then place-construct an object there with placement new— `T* q = new(p) T(args);` —you must first call `q->~T();` and then `::operator delete(p);`.)
    - Match the family exactly: `new` ↔ `delete`, `new[]` ↔ `delete[]`, aligned forms (`std::align_val_t`) with their aligned deletes, and class-specific `operator`

`new/delete` with their partners.

- Bottom line: **`free()` is only for `malloc` / `calloc` / `realloc` / `aligned_alloc`. `delete` / `operator delete` are only for memory from `new` / `operator new`.** Mixing them is undefined behavior even if it "seems to work" on a given platform.

- *Is it safe to call `delete` on a `nullptr`?*
  - Yes — calling `delete` (or `delete[]`) on a `nullptr` is **completely safe** and explicitly allowed by the C++ standard.
  - When you write something like

    ```
    int* p = nullptr;
    delete p;
    ```

  - the `delete` expression simply has **no effect**. The runtime checks whether the pointer is null before attempting to free memory.
  - This behavior exists so that cleanup code can safely call `delete` on a pointer without needing to check `if (p) delete p;`. It makes destructors and error-handling paths simpler and less error-prone.

- *Should you pass smart pointers by value or by reference?*
  - You *should* pass shared pointers by reference, not value, in most cases. While the size of a `std::shared_ptr` is small, the cost of copying involves an atomic operation (conceptually an atomic increment and an atomic decrement on destruction of the copy, although I believe that some implementations manage to do a non-atomic increment).
  - In other cases, for example `std::unique_ptr` you might prefer to pass by value, as the *copy* will have to be a move and it clearly *documents* that ownership of the object is transferred to the function (if you don't want to transfer ownership, then pass a reference to the real object, not the `std::unique_ptr`).
  - In other cases your mileage might vary. You need to be aware of what the semantics of copy are for your smart pointer, and whether you *need* to pay for the cost or not.
