# cherno-cpp

A repository for working through [the Cherno’s C++ YouTube series](https://www.youtube.com/playlist?list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb).

## Overview

Why write in C++? Because we care about things like: - Efficient Memory Management - Runtime Performance - Hardware Optimization

If you don’t care about these things, don’t choose C++!

## Contents

This tutorial series will cover the following topics:

* Introduction - How C++ Works
  + 1. [Welcome to C++](#X4b9c9663e31f330f4ccab39824ad800de3c0e66)
    2. [How C++ Works](#video-5---how-c-works)
    3. [TODO: How the C++ Compiler Works](#video-6---how-the-c-compiler-works)
    4. [TODO: How the C++ Linker Works](#video-7---how-the-c-linker-works)
* C++ Fundamentals
  + 1. [Variables](#video-8---variables-in-c)
    2. [Functions](#video-9---functions-in-c)
    3. [Header Files](#video-10---header-files)
    4. [Conditions and Branching](#video-12---conditions-and-branching-in-c)
    5. [Control Flow](#video-15---control-flow-in-c)
    6. [Pointers](#video-16---pointers-in-c)
    7. [References](#video-17---references-in-c)
    8. [Classes vs. Structs](#video-19---classes-vs-structs-in-c)
    9. [static](#video-21---static-in-c)
    10. [Enums](#video-23---enums-in-c)
    11. [Destructors](#video-25---destructors-in-c)
    12. [Inheritance](#video-26---inheritance-in-c)
    13. [Virtual Functions](#video-27---virtual-functions-in-c)
    14. [Interfaces (Pure Virtual Functions)](#X4502d34ae42e989a9578abbcdf757d1f1ca845c)
    15. [Visibility](#video-29---visibility-in-c)
    16. [const](#video-33---const-in-c)
    17. [mutable](#video-34---the-mutable-keyword-in-c)
    18. [Member Initializer Lists](#video-35---member-initializer-lists-in-c)
    19. [Ternary Operators](#video-36---ternary-operators-in-c)
    20. [How to Create/Instantiate Objects](#X3cf4d0d3cbd7e883b32cd31932234793bd58d7d)
    21. [new](#video-38---the-new-keyword-in-c)
    22. [Implicit Coversion and the explicit Keyword](#Xce337322db4612215bf2c477525aa9f88ccace5)
    23. [Operators and Operator Overloading](#X05926bfd8e4fc5d3a7c17416378cbdfb70fbd89)
    24. [this](#video-41---this-keyword-in-c)
    25. [Copying and Copy Constructors](#Xe335e2c9f3941cdd062e2227b2427288f259534)
    26. [The Arrow -> Operator](#video-45---the-arrow-operator-in-c)
    27. [Local Static](#video-48---local-static-in-c)
    28. [auto](#video-56---the-auto-keyword-in-c)
* Data Structures
  + 1. [TODO: Arrays](#video-30---arrays-in-c)
    2. [Dynamic Arrays (std::vector)](#video-46---dynamic-arrays-in-c-stdvector)
    3. [Optimizing the Use of std::vector](#Xb5369da535cf2442e3933439c21f8d900874e68)
    4. [Static Arrays (std::array)](#video-57---static-arrays-stdarray-in-c)
    5. [Multidimensional Arrays (2D Arrays)](#X11d9d51ae8cb449cb0a062fce60bfa79d4f4f2f)
    6. [Unions](#video-67---unions-in-c)
    7. [Iterators](#video-93---iterators-in-c)
    8. [Maps (std::map and std::unordered\_map)](#Xad4dc9d0f948e461565fcf6414ef35edc8b4fec)
* Memory Management in C++
  + 1. [Object Lifetime (Stack/Scope Lifetime)](#Xa488fc559ae54135dd6e104aa6755eab1a2f7cd)
    2. [Smart Pointers](#video-43---smart-pointers-in-c)
    3. [Stack vs. Heap Memory](#video-54---stack-vs-heap-memory-in-c)
    4. [Safety in Modern C++ and How to Teach It](#X5e42ce66bc65a4f321900e142a280285c6d616f)
    5. [Track Memory Allocations the Easy Way](#X918c4e225ec01b3ce5b589b61d2bf411a629ac2)
    6. [Weak Pointers (std::weak\_ptr)](#X20297e087995f99e985a0d9c0569be977b528af)
* C++ Advanced Topics
  + 1. [How to Deal with Multiple Return Values](#Xcc942548b2a900ebac079d8fb6c419c36ab6985)
    2. [Templates](#video-53---templates-in-c)
    3. [Macros](#video-55---macros-in-c)
    4. [Function Pointers](#video-58---function-pointers-in-c)
    5. [Lambdas](#video-59---lambdas-in-c)
    6. [Namespaces](#video-61---namespaces-in-c)
    7. [Sorting](#video-65---sorting-in-c)
    8. [Type Punning](#video-66---type-punning-in-c)
    9. [Virtual Destructors](#video-68---virtual-destructors-in-c)
    10. [Casting](#video-69---casting-in-c)
    11. [Dymamic Casting](#video-73---dynamic-casting-in-c)
    12. [Structured Bindings](#video-75---structured-bindings)
    13. [Singletons](#video-82---singletons-in-c)
    14. [Argument Evaluation Order](#Xe160fd9c1ed49de82ed290e6763f188ea604cb8)
    15. [TODO: Binary and Bitwise Operators](#X6bd565c99e7f90cd47d038a57247510047de95e)
    16. [TODO: Bitwise AND, OR, XOR and NOT](#X263149c4d39811046b03fd8d7bf12e17cf9d112)
    17. [What Exactly is NULL?](#video-101---what-exactly-is-null)
    18. [Conversion Operators](#video-103---conversion-operators-in-c)
* Performance & Benchmarking
  + 1. [Threads](#video-62---threads-in-c)
    2. [Timing](#video-63---timing-in-c)
    3. [Benchmarking](#video-74---benchmarking-in-c)
    4. [How to Make C++ Run Faster with std::async](#Xe0d8827bc5f80e67ff150a3f9e2c966e3b1ccbe)
    5. [TODO: How to Make your Strings Faster](#X4819f574ad5cd36267a92c424b0c0b204b983c9)
    6. [TODO: Small String Optimizations](#X2c566a121ca8259a50ae68166ff2debf92b8cf5)
* Storing Multiple Types of Data
  + 1. [How to Deal with Optional Data](#X1bd9bcf2a3f444639a625250a87693e2a42f44a)
    2. [Multiple Types of Data in a Single Variable](#X55f5e8e6d79560493d5ee96d09f7a4e5d6b5167)
    3. [How to Store ANY Data](#video-78---how-to-store-any-data-in-c)
* Move Semantics
  + 1. [l-values and r-values](#video-85---l-values-and-r-values-in-c)
    2. [Move Semantics](#video-89---move-semantics-in-c)
    3. [std::move and the Move Assignment Operator](#Xcc75e7236a28ec1009dad9fa1a53fffc6bccdf4)
* Workflow and Debugging
  + 1. [Using Libraries](#video-49---using-libraries-in-c)
    2. [TODO: Using Dynamic Libraries](#video-50---using-dynamic-libraries-in-c)
    3. [TODO: Making and Working with Libraries](#X012d7f6dfb2014460167199e727eaa3c0d58b1b)
    4. [TODO: Conditional and Action Breakpoints](#X7f5fa7634aaa396829471a9cbae1b8cfff0cc56)
    5. [TODO: Precompiled Headers](#video-72---precompiled-headers-in-c)
    6. [TODO: Continuous Integration](#video-86---continuous-integration-in-c)
    7. [TODO: Static Analysis](#video-87---static-analysis-in-c)
* Writing Our Own Data Structures
  + 1. [Custom Array](#Xf499b701adecf78dd4eac04aadf32507e677e5b)
    2. [Dynamic Array (Vector)](#X9efef512ab1673e86155445bd30873da2d19c45)
    3. [Writing an Iterator](#video-94---writing-an-iterator-in-c)

## Introduction - How C++ Works

### Video #1 - Welcome to C++ (What We’ll be Learning)

* How C++ actually works.
* Memory and pointers.
* Memory “arenas”, custom allocators, smart pointers, move semantics.
* Templates: *“If you know how to use templates well they’re extremely powerful and will make your life a lot easier.”* - Cherno
* Data structures (and how to make them faster than the standard STL data structures).
* Low-level optimization via “compiler-intrinsics” and assembly.

### Video #5 - How C++ Works

* What we want to know: How do we go from source code (.cpp files) to an executable binary?
* Preprocessor statements (# statements, e.g. #include) happen before compilation.
  + #include statements *literally* pre-pend the contents of another file into your file.
* Every C++ application needs an **entry point**, typically this is main().
* Operators are just functions! Think of operators as functions.
* Header files do not get compiled. The contents of included files get compiled as part of the cpp files that they’re included in.
* The Compiler creates an **object file** (.obj) for *every* cpp file (each cpp file is referred to as a **translation unit**).
* The **linker** “glues” the object files into an executable.
  + A linker error can happen when a symbol (e.g. a function name) that was “promised” to exist (e.g. via forward-declaration) cannot be resolved or found.
  + This error is referred to as an *“unresolved external symbol”*.
* There are a few key differences between a **compiled** language like C++ and an **interpreted** language like Python:
  + C++ is a **statically typed** language, i.e. variable types are checked/known at **compile time**. Python is **dynamically typed**, i.e. variable types are evaluated at **runtime**.
  + The C++ compiler translates source code into bytecode prior to execution, whereas the Python interpreter performs this same translation on-the-fly at runtime, typically leading to slower execution speeds.

### Video #6 - How the C++ Compiler Works

* The compilers job is simple: its job is to translate our code into either **constant data** or **instructions**.
* The output of the compiler is an *intermediate format* file called an **object file**. Object files are passsed from the **compiler** to the **linker**.
* The compilation process happens in three steps: **pre-processing** (#include statements and macros), **tokenizing**, and finally the creation of the **abstract syntax tree**.

### Video #7 - How the C++ Linker Works

* TODO:

## C++ Fundamentals

### Video #8 - Variables in C++

* TODO:

### Video #9 - Functions in C++

* Functions prevent code duplication.
* Each time a function is called in C++, the compiler generates a “call instructions”\*. A call instruction creates a *stack frame* for that function, meaning we have to push the function parameters and return address onto the *stack*. Additionally, we have to jump to a different part of our binary to execute a function, and then return to where it was called from. All of this is “expensive”.
* \*: this only happens if the chooses not to “inline” your function.

### Video #10 - Header Files

* Header files are used to store function and/or class **declarations** (**not** definitions).
* #pragma once: Only include this file once into a single “translation unit”. A translation unit is a single cpp file.

### Video #12 - Conditions and Branching in C++

* if statements consist of two parts:
  + The evaluartion of the conditional and the branch(es) that can be jumped to (jumping to a different part of memory).
  + “Optimized” code may attempt to avoid branching altogether to prevent jumping around in memory.

### Video #15 - Control Flow in C++

* continue, break, return

### Video #16 - Pointers in C++

* *“Possibly the most important episode in the series!”* - Cherno
* raw pointers, not discussing *smart pointers* today.
* A pointer is an integer that represents a memory address.
* Being “invalid” is a perfectly acceptable state for a pointer.
* 0, NULL and nullptr mean the same thing when it comes to pointers.
* Note that dereferencing should always be done prior to any operations on the underlying object, e.g. if we wanted to increment an integer value via an int\* to value, we’d do (\*ptr)++ instead of \*ptr++. The latter will increment the pointer and then attempt to derefernece it (most likely leading to a crash).
* Pointers are *ContiguousIterators* (of an array). You can use ++ to go to the next item that a pointer is pointing to, and +4 to go to the 5th element.

### Video #17 - [References in C++](https://www.youtube.com/watch?v=IzoFn3dfsPA&list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb&index=17)

* References are really just an extension of pointers.
* References are “pointers in disguise”, i.e. references are just syntacial suger on top of pointers.
* References **must** “reference” existing variables, whereas pointers can be created as new variables.
* Unlike pointers which have a unique identity, i.e. a distinct memory address and an amout of space that can be measured with sizeof, references are **not** new variables (i.e. they don’t occupy memory or have their own memory address), they are simply an **alias** for an existing variable.
* A reference **cannot** be reassigned.
* You cannot create a collection (e.g. std::vector) of references.
* A pointer can be assigned to nullptr, whereas a reference must be bound to an existing object.
* The following two ways of incrementing an int are effectively the same: ``` void increment\_via\_pointer(int\* ptr) { // Dereference pointer before incrementing underlying value (\*ptr)++; }
* void increment\_via\_reference(int& value) { value++; } ```

### Video #19 - Classes vs. Structs in C++

* Visibility of member variables:
  + struct: public by default.
  + class: private by default.

### Video #21 - Static in C++

* **Context 1**: static keyword when used outside of a class or struct.
  + static effectively allows you to mark a variable as “private” for a specific translation unit (cpp file).
  + This prevents any other translation unit (cpp file) from finding that static variable in the linking process via the extern keyword.
* **Context 2**: static keyword when used inside a class or struct.
  + A static member variable of a class/struct is the same across all instances of that class/struct, i.e. there will be only **one instance** of the static member variable.
  + A static method of a class does not have access to the object itself, i.e. cannot access the this keyword.
  + static methods (and also variables?) cannot access non-static variables because static methods do not actually have a “class instance”.
* See more on static from [this](https://stackoverflow.com/questions/15235526/the-static-keyword-and-its-various-uses-in-c) StackOverflow question.

### Video #23 - Enums in C++

* Enums provide a way to give a name to a value and organize groups of names and values that make sense together.
* enum vs. Enum class?

### Video #25 - Destructors in C++

* The destructor applies to both **stack** and **heap** allocated objects.
* If you allocate an object via the new keyword, when you call delete on that object, the destructor will be called.
* When a stack-based object goes out of scope, the desctrutor will be called.
* Why write a destructor?
  + Any manually-allocated memory on the heap needs to be manually cleaned up in the destructor or you will create memory leaks.

### Video #26 - Inheritance in C++

* A note on Constructors: every constructor in the inheritance hierarchy gets called in the order of base-class -> derived-class.
* Destructors get called in the *reverse* order!
* **Polymorphism**:
  + Achieved either via function overloading or operator overloading, e.g. you can use the + operator to add two ints, two floats or even concatenate two strings.
  + All members of the base class are part of the derived class. However, the derived class can only access members that are public or protected - private members of the base are **always** inaccessible to the derived class.
  + A derived class can inherit from a base class in the following ways: public, protected, or private

| Base Class Member Access Modifier |  | Visibility to Derived Class |  |
| --- | --- | --- | --- |
| *inheritance type* | public | protected | private |
| public | public | protected | private |
| protected | protected | protected | private |
| Private | inaccessible | inaccessible | inaccessible |

* + A **pure abstract class** (*OOP factory*) is a means of implementing polymorphishm such that users obtain a *unique pointer* to a lightweight or abstract base class, the implementation details are in the derived class that overrides its virtual member functions.
* From the [C++ Style Guide](https://google.github.io/styleguide/cppguide.html):
  + “Limit implementation inheritance. Prefer **interface inheritance** (via pure virtual functions) or use **composition** instead.”
  + **Interface Inheritance**: The use of inheritance from exclusively abstract classses, where *no values or code* are inherited from the parent.
* When dealing with **derived classes**, the way you consume them (**by-reference** or **by-value**) has significant implications, especially concerning polymorphism and **object slicing**.
  + **Consuming an object of a derived class by-value creates a copy of the object**. The copied object will be of the base-class type, and it will be stripped of any of its derived-specific data members or member functions leading to what’s known as **object slicing**.
  + **Consuming a derived-class object by-reference (or pointer) does not result in a copy**. The function operates on the original object, and its type is preserved. If the derived class overrides any virtual functions of the base class, the correct derived-class function will be called.
* class Base {  
   public:  
   virtual void print() const { std::cout << "Base" << std::endl; }  
  };  
    
  class Derived : public Base {  
   public:  
   void print() const { std::cout << "Derived" << std::endl; }  
  };  
    
  void print\_by\_ref(const Base& base) { base.print(); }  
    
  void print\_by\_value(Base base) { base.print(); }  
    
  Derived b;  
  print\_by\_ref(b); // prints "Derived"  
  print\_by\_value(b); // prints "Base"

### Video #27 - Virtual Functions in C++

* Virtual functions allow us to override methods in sub-classes. A method marked as virtual in the parent class can be overridden in the child-class.
* Virtual functions implement “dynamic dispatch” via a “v-table”. The v-table contains a mapping of all the virtual functions in the base-class to their overridden functions in the child-class.
* Marking a function as virtual tells the compiler to create a v-table for that function.
* If you want to override a function in the child-class, you **must** mark the corresponding function in the parent-class as virtual.
* The override keyword should be used to mark the child-class method to indicate that it is overriding a virtual method of the parent class.
* Virtual functions come at the expense of creating a v-table, but in reality, the performance difference is negligible.

### Video #28 - Interfaces in C++ (Pure Virtual Functions)

* virtual void fcn() = 0;. The = 0; is what makes it *pure* virtual.
* The concept of a *pure virtual function* in a base class allows us to define a base-class member function that does not have an implementation, thereby forcing sub-classes to implement that function.
* This is similar to an abstractmethod in python.
* An *interface class* (or *factory*) consists entirely of *pure virtual methods* and cannot actually be instantiated.
* A sub-class of an interface class can **only** be instantiated if it implements **all** of the pure virtual methods defined by the parent-class.

### Video #29 Visibility in C++

* private: these member variables and functions can (and should) only be accessed by the class itself.
* protected: these member variables and functions can (and should) only be accessed by the class itself and any sub-class.
* public: these member variables and functions can (and should) only be accessed by any other class of function.

### Video #30 - Arrays in C++

* TODO

### Video #33 - Const in C++

* **NOTE**: *This is where I picked up after the interview. I’m taking my time getting through the content now.*
* const is a “promise” that something will not be changed (not actually strictly enforced).
* The following table is created from Cherno’s explanation:

|  |  |
| --- | --- |
| Keyword Declaration | Interpretation |
| const int\* ptr | Value at pointer location cannot be changed, but the pointer can be re-assigned to a new address. |
| int\* const ptr | The pointer cannot be assigned to a new address, but the value at the pointer location can be changed. |
| const int\* const ptr | The pointer cannot be re-assigned to a new address and the value at the pointer location cannot be changed. |
| void MyClass::func() const | Guarantees that this function will not modify any member variables of MyClass, i.e. this function is “read-only”. |

* Rule of thumb: Mark any methods that are intended to be read-only as const, e.g. void MyClass::func() const.
* The mutable keyword allows const methods to modify member variables.
* The following table ([source: SO](https://stackoverflow.com/questions/1143262/what-is-the-difference-between-const-int-const-int-const-and-int-const)) summarizes the use of the const keyword. Remember to read the keyword delaration backwards (as driven by the [Clockwise/Spiral Rule](https://c-faq.com/decl/spiral.anderson.html)).

|  |  |
| --- | --- |
| Keyword Declaration | Interpretation |
| int\* | *Pointer to an int* |
| int const \* | *Pointer to a const int* |
| int \* const | *Const pointer to an int* |
| int const \* const | *Const pointer to a const int* |

* Note that the first const can occur on either side of the type, so
  + int const \* == const int\*
  + int const \* const == const int \* const

### Video #34 - The Mutable Keyword in C++

* Two (*very different*) use cases: with const and with **lambdas**.
  + The mutable keyword allows const methods to modify member variables.
  + **lambda** (quick definition): a throw-away function that you can assign to a variable.

### Video #35 - Member Initializer Lists in C++

* A way to initialize class member function via the contstructor.
* Member initializer lists are the most memory/time efficient way to construct an object of a class that consists of non-primitive type members, i.e. members of other class types. For an example of this, see app/member\_init.cpp that demonstrates how the Entity’s Logger is effectively created twice (and thrown away once) when member initialization is used improperly.

### Video #36 - Ternary Operators in C++

* Effectively syntatic sugar for an if statement.
* <var> = <conditional> ? <value if true> : <value if false>
* e.g. std::string rank = (level >= MASTER\_LVL) ? "Master" : "Novice"
* Using a ternary operator to do something like this is actually *faster* because of something called *Return Value Optimization*.
* In the example below, an empty string object called rank is created and then overriden with either “Master” or “Novice” once the conditional is evaluated. Using the ternary operator prevents rank from being created as an empty string object before being re-assigned. std::string rank; if (level >= MASTER\_LVL) rank = "Master"; else rank = "Novice";
* Worth noting that ternary operators can be nested, but this can quickly become confusing to read and isn’t recommended.

### Video #37 - How to Create/Instantiate Objects in C++

* All objects that are created must occupy some memory.
* Memory is divided into two main areas: the **stack** and the **heap**.
* **Stack** objects have a pre-determined lifespan that is defined by the *scope* that they are created in. Once that variable/object goes *out of scope* , then the local *stack frame* gets destroyed, i.e. that memory is freed.
* So, when should you create an object on the Stack? Whenever possible! Creating an object on the stack is typically the fastest and most “resourced managed” way to create an object in C++.
* The Stack is much smaller than the heap. The size of the Stack is platform-dependent, but is typically 1-2MB.
* **Heap** objects don’t behave in the same way.
* The new keyword creates an object on the Heap and returns a pointer to that object, e.g. Entity\* e = new Entity(). This object does **not** have a pre-determined lifespan and must be deleted manually using delete.
* Stack vs. Heap overview:
  + Performance: Heap allocation takes longer than Stack allocation.
  + Memory Allocation: Heap allocation requires *manual* freeing of that memory - freeing memory isn’t nicely managed the same way that it is for objects on the Stack. This can lead to *memory leaks*.
* So when should you create an object on the Heap?
  1. Is your object really, really big? (larger than the size of the Stack)
  2. Do you want to manaually control the lifespan of your object?
* If no to both 1. and 2. then create your object on the Stack!
* *Smart Pointers* (which we’ll get into later) provide a means to allocate an object on the Heap but with the memory-mangement advantages that come with Stack allocation.

### Video #38 - The new Keyword in C++

* new finds a block of memory (via the **free list**) that is large enough to accommodate our needs, and returns a pointer to that block of memory.
* new does two things:
  + Allocates the memory required to store the object.
  + Calls the constructor for that object.
* new is actually an operator, in the same sense that the +, == or << (stream insertion) are operators. That means that new can be **overloaded**.
* So what does new do under the hood? It calls malloc(). The following two lines are *almost* equivalent aside from the fact the new calls the Entity constructor, whereas the second line purely allocates memory and returns a pointer to that memory. This is just for instructional purposes, you don’t want to be doing anything that looks like the second line.

Entity\* e1 = new Entity;  
Entity\* e2 = (Entity\*)malloc(sizeof(Entity));

* new and delete are an inseparable pair. **All** newly allocated memory must eventually be deleted. This is handled in an automated way by *smart pointers*.
  + new and delete for standard objects.
  + new[] and delete[] for arrays.

### Video #39 - Implicit Conversion and the explicit Keyword in C++

* Implicit Conversion allows C++ to convert between one type and another, so long as only a “single step” conversion exists between those two types.
* explicit **disables** implicit conversion.
* Since C++11, explicit can be applied to more than just constructors - it’s now valid when applied to conversion operators as well.
* In the following example, marking the Foo constructor explicit would prevent implicit conversion from taking place in the call to bar(42);. ``` struct Foo { // A “converting constructor” allows implicit conversion. Foo(int x) {} };
* // It looks like we have to pass a Foo, but do we? void bar(Foo foo) {};
* int main() { // This is allowed because of implicit conversion. bar(42); }  
  ```
* Also note that an explicit constructor will prevent *copy-initialization*, e.g. Foo foo = 7. Only *direct-initialization*, Foo foo{7} is allowed for a constructor marked explicit.
* See [cppquiz.org #131](https://cppquiz.org/quiz/giveup/131) for further explanation.
* Probably won’t find myself using this very often, but good to know it exists and what it does.

### Video #40 - Operators and Operator Overloading in C++

* Operators: =, ==, << (stream insertion), &, &&, etc.
* new, delete and () are also operators.
* “Operators *are* functions!” - Cherno

### Video #41 - this keyword in C++

* this is a pointer to the current object instance that the member-method belongs to.
* Deferencing (getting the value or object stored at the pointer location) the this pointer looks like this->x\_ which is equivalent to (\*this).x\_.
* Can be used to pass the current object (or a pointer to the current object) to another function, e.g. func(\*this) or func(this).
* Don’t ever delete this.

### Video #44 - Copying and Copy Constructors in C++

* Copying is expensive and should be avoided if unnecessary.
* The **Copy Constructor** is supplied by default by C++ and is implemented as a **shallow copy**. This will create problems for any objects that have heap-allocated member pointers because the pointer itself will be copied.
  + This results in pointer stored by the copied object pointing to the same memory address as the original object. When that memory is freed by the original object, attempting to free that same memory by the copied object will cause *hard-to-diagnose* errors.
* In order to implement a **deep copy** of an object, we must rewrite the Copy Constructor ourselves: <Type>(const <Type>& other){}
* **Best Practice**: Prefer passing objects by const reference (const <type>&) to prevent unnecessary copying.

### Video #45 - The Arrow Operator in C++

* The arrow operator -> is used to *dereference* a pointer.
* The following lines are equivalent (the () are required in the first line because of operator precedence, i.e. dereferencing must take place before accessing the member method): (\*ptr).member\_method(); ptr->member\_method();
* Because the -> is an operator, it can be overloaded. This can useful if we choose to write our own scoped pointer class, e.g. ScopedPtr.
* Bonus functionality: The arrow operator can be used to determine the offset in memory of a particular member variable, e.g: ``` struct Vec3 { // Floats are 4 bytes in length float x, y, z; }
* int main() { // Starting at zero (nullptr) give me the offset of the member y int offset = (int)&((Vec3\*)nullptr)->y; std::cout << offset << std::endl; }
* Terminal Output: 4 ```

### Video #48 - Local Static in C++

* Can declare a variable as static in a local scope - this is different from the other two use cases of static that we’ve seen already.
* Declaring a variable as static within a local scope (e.g. within a function) restricts access to that variable to that local scope, but extends its lifetime to the lifetime of the program. ``` void f() { static int i = 0; i++; std::cout << “i: << i << std::endl; }
* int main() { f(); f(); f(); }
* Terminal Output: i: 1 i: 2 i: 3 ```
* Often use of local static variables are discouraged.
* One possible use case is for *Singleton* classses, i.e. a class that should only have one instance in existance.

### Video #56 - The auto Keyword in C++

* *Type deduction* (use of the auto keyowrd) allows C++ to become a *psuedo weakly-typed* language.
* When to use auto?
  + For very long types that are annoying to write out and you don’t want to using to create an alias.
  + For iterators in for loops: ``` std::vector strings;
  + // This is ugly and long for (std::vector::iterator it; strings.begin(), it != strings.end(), it++) { // do something }
  + // This is much cleaner and easier to read for (auto it; strings.begin(), it != strings.end(), it++) { // do something } ```
* If you need a reference, use auto&.
* From the [Google C++ Style Guide](https://google.github.io/styleguide/cppguide.html):
  + “Use type deduction only to *make the code clearer or safer*, and do not use it merely to *avoid the inconvenience of writing an explicit type*.”
  + ✅: auto foo = std::make\_unique<Foo>(); - it’s obvious that foo is of type Foo.
  + ❌: auto foo = myWdigetFactory(); - the type of foo is unclear.

## Data Structures in C++

### Video #46 - Dynamic Arrays in C++ (std::vector)

* Time to get accustomed with the standard template library (STL)!
* std::vector is actually a dynamic array (list) - *dynamic* in the sense that it can be resized, e.g. extended, appended to, etc.
* When you exceed the allocated size of a particular std::vector instance, it creates a new array in memory, copies the contents of the original vector into the new vector and deletes the original vector. In practice, this re-allocation can occur quite often and can result in performance losses.
* Dynamic arrays store their memory contiguously (in line, not fragmented in memory). This is optimal for any operation that requires iterating over the vector.
* When this can become non-optimal is when you anticipate that your vector will need to be resized frequently because this will require copying the objects themselves over and over. *Moving* instead of copying largely solves this issue, but not entirely because there is still some copying involved.
* **Question:** Should I be storing pointers to heap-allocated objects in my vectors (lists), or should I store the stack-allocated objects themselves?
  + **Answer:** It depends. The primary consideration is that it is technically more optimal to store the objects themselves in the list because storing the objects themselves requires that the memory allocated for those objects is inline (contiguous). A vector of pointers can be optimal in the case when that vector may need to be resized frequently.
* **Best Practice:** Prefer passing dymanic arrays by const reference to avoid uncessary copying.
* The sizeof(std::vector<T>) is independent of the number of elements currently stored in the vector.
  + It’s usually a small, fixed size, often 24 or 32 bytes on 64-bit systems.
  + This is because std::vector is **implemented as three pointers**: one to the beginning of the vector, one to the end, and one to the current capacity of the allocated storage.
* std::array allocates its memory on the stack, whereas std::vector allocates its memory on the heap.

### Video #47 - Optimizing the use of std::vector in C++

* We can use std::vector.reserve(n) to allocate enough memory for n objects *without* actually wasting time constructing those objects before we’re ready to push them onto the vector.
* We can also use std::vector.emplace\_back() to construct the object being added to the vector *in place* (at the location in memory allocated by the vector) as opposed to creating it in the local stack frame and then having to copy it to the memory location allocated by the vector as is done by std::vector.push\_back().

### Video #57 - Static Arrays (std::array) in C++

* Static arrays have a pre-defined size and their size **cannot** be changed.
* Very similair to C-style arrays: int arr[10]; can be rewritten as a static array like std::array<int, 10> arr;.
* **Question:** How do we take a std::array as a function argument if we don’t know the size? ``` // Template that doesn’t assume type or size template<typename T, size\_t s> void f(std::array<T, s> arr) { for (const auto& item : arr) { // do stuff } }
* // Template that assumes the type (and int array), but not the size template void f(std::array<int, s> arr) { for (const int& item : arr) { // do stuff } } ```
* std::array allocates its memory on the stack, whereas std::vector allocates on the heap.
* **Best Practice:** Prefer std::array to *old* C-style arrays.
  + std::array supports STL features like size() and iterators (begin() and end()).
  + Can apply STL algorithms like std::sort to std::array.
  + No additional performance cost associated with std::array since the size is stored as a template argument.
* The following table is an overview of the differences between std::vector and std::array.

|  |  |  |
| --- | --- | --- |
| Feature | std::vector | std::array |
| Allocation | **Heap** for elements, stack for container object | **Stack** (usually), or static storage |
| Size | Dynamic, can change at **runtime** | Fixed at **compile time** |
| Resizing | Possible | Not possible |
| Memory Overhead | Higher, due to managing heap allocation | Low, only stores elements |
| Access Speed | Slightly slower to to possible indirection | Fast, direct access to elements |
| Use Cases | Variable-size data, flexibilty required | Fixed-size data, performance-critical code |

### Video #64 - Multidimensional Arrays (2D Arrays) in C++

* n-dimensional arrays (when to use them, and when **not** to use them!)
* An array is actually just a pointer to the beginning of the array. Extending that concept to a 2D array would mean that a 2D array is actually just an array of pointers, where each pointer is the starting location of a single array of the larger 2D array.
* Allocating a 2D array might look something like int\*\* arr\_2d = int\*[50];. We can read int\*\* as (int\*)\* or *“a pointer to a collection of integer pointers”*. Each element of arr\_2d will be an integer pointer, so we can do something like arr\_2d[idx] = nullptr;
* Deleting a multidimensional array isn’t trivial either. Say we declare the following 5x10 2D array then attempt to delete it: int\*\* arr\_2d = new int\*[5]; for (int i = 0; i < 5; i++) { arr\_2d[i] = new int[10]; } delete[] arr\_2d; That will only delete the array of pointers pointing to each of the 50 arrays of integers. The integers themselves will not be deleted and become a memory leak. What we have to do to delete a 2D array without leaking memory is: for (int i = 0, i < 5; i++) { delete[] arr\_2d[i]; } delete[] arr\_2d;
* In the example above, there is **no guarantee** that each of the 5 blocks of 10 integers will be contiguous in memory. Which can make our array slower to iterate over than an array that had all 5 blocks of 10 integers allocated contiguously in memory.
* **Best Practice:** Because of this issue, it may not be a good idea to use 2D arrays for things like images or textures where we want access to each pixel to be fast. So instead, prefer to store an image as a 1D array and be smart about how you iterate over it.

### Video #67 - Unions in C++

* A Union is a bit like a class or struct type, but it can only occupy the memory of one member at a time. In other words, … (these are kind of confusing).
* “Put differently, a union of multiple members places each member at the same starting address. This greatly saves on memory used, but the downside is that you can only use one member at a time because they all start at the same address. As you can imagine these were enormously helpful in the 90’s when memory was limited. They still have use cases today, but you have to be careful because it can feel like you’re working with separate unique variables when in reality you’re working with different variables all occupying the same starting address. Unions can tend to trip you up if you’re not careful leading to really wacky bugs but they can be a powerful tool if used seldomly and correctly.”
* Useful for when:
  + We want to give two different names to the same variable, e.g. it may be useful to think of a three-element vector (x, y, z) as a color (RGB) where x maps to R and so forth.

### Video #93 - Iterators in C++

* Iterators are used to traverse data structures, and if we’re writing our own data structures, we probably want to support functionality like idexing and iteration.
* *Range-based* for loops (available since C++11) are made possible by iterators. std::vector implements both the begin() and end() functions that each return an *iterator* that points to a particular position (the beginning and the *past-the-end* element) in the vector. std::vector<int> values = { 1, 2, 3, 4, 5 }; for (int& val : values) { std::cout << "value: << val << std::endl; }
* Traversing a container by using its iterator explicitly is also possible, but less common because a range-based iteration is essentially shorthand for this. But in some situations, e.g. erasing or inserting new elements, you may want to manipulate the iterator. for (std::vector<int>::iterator it = values.begin(); it != values.end(); it++) { // Dereference the iterator (bc it's a pointer) to get the value std::cout << \*it << std::endl; }
* See app/93\_iterators.cpp for more examples, including iteration over non-indexable types, e.g. an unordered map (dictionary), via [structured bindings](#video-75-structured-bindings).

### Video #100 - Maps (std::map and std::unordered\_map) in C++

* Maps allow us to associate a *key-value* pair (a dictionary in Python).
* std::map is an *ordered* map that is a “self-balancing binary search tree,” typically a “red-black” tree.
  + In a tree data structure, elements are sorted via comparison (typically using a less than operator). That way, when you iterate over a map, you’re iterating over the elements in a sorted order.
* std::unordered\_map is a *hash-table*. “It uses a *hash function* to hash the key and generate an index to figure out what which”bucket” your value is in.” - Cherno. Because it is unordered, value retrieval has the *potential* to be faster than with std::map.
* **Best Practice:** Because of this performance difference, prefer using std::unordered\_map over std::map unless you need your elements to be sorted.
* Requirements of the key type choice:
  + std::unordered\_map: The key must be *hashable*. Note that pointers are *always* hashable because a pointer is simply a 64-bit integer, i.e. T\* == uint64\_t.
    - A hash function is *not* required to return a unique hash. Two things with the same hash will be stored in the same bucket, which is inefficient for lookups, but ut hash collisions will not compromise the map.
  + std::map: The key type must implement the less than operator, operator<. The less than operator not only plays the role of comparison for an ordered map, it also defines a unique key within the map. So if we attempt to add a new element that evaluates as “equal to” an existing element in the map, that new element will not be added.
* Note that the index [] operator for a map works strictly as an insertion operator. There is no const version of the index operator, i.e. [] works *only* as a *setter*, and not as a *getter*. ``` std::unordered\_map<std:string, CityRecord> cities;
* // The following will return a reference to the value stored by the “Berlin” key // if it exists, otherwise, it will create the “Berlin” key with a possibly // non-initialized CityRecord instance. CityRecord& berlin\_data = cities[“Berlin”] ```
  + This can actually be useful in terms of creating an object *in-place* rather than having to create it in the local stack frame and then copy it into the map.
  + If we want to retrieve data without inserting it, we need to use at().
  + **Best Practice**: Prefer at() for retreiving map elements because it works for both non-const and const maps.
* **Iteration**: recall that std::vector stores its data in *contiguous* memory and this improves the efficiency of iterating over a vector. Iterating over the elements of a vector will *always* (typically by an order of magnitude or more) be faster than iterating over the elements of a map.
  + Note that it is *not* guaranteed that the elements of a std::unordered\_map will be kept in the order in which they were inserted.

## Memory Management in C++

### Video #42 - Object Lifetime in C++ (Stack/Scope Lifetimes) in C++

* Will be taking a look at the lifetime of stack-based variables.
* Each time a new scope is *entered*, a new *stack frame* is pushed onto the stack. The stack frame consists of any variables declared within that scope (and possibly other data?) When the scope is exited, that stack frame is deleted and the memory on the stack is freed.
* What is meant by *scope*? Basically anything declared within {}.
* All stack-based variables/objects have a scope-based lifetime, i.e. once a stask-based variable/object goes out of scope, it’s memory is freed.
* **Common mistake**: Attempt to create a stack-based variable within a function and then return a pointer to that variable. Once the function returns and that variable goes out of scope, that variable no longer exists and the pointer that is returned now points to a freed memory location that doesn’t contain the data we expect it to. int\* create\_array(const int size) { // This creates an array on the stack (which is scope-based) int array[size]; return array; }
* How can take advantage of the lifespan of stack-based (scope based) variables? Yes. “Scoped” classes like *smart pointers* and *scoped locks* take advantage of this.
* A Smart Pointer is effectively a wrapper around a raw pointer that heap-allocates some memory on creation and then deletes that pointer upon destruction (when the smart pointer itself goes out of scope?)
* Mutex Locking: In the context of threading, a scoped mutex lock allows us to “lock” a function upon entry (and unlock at exit) such that mutliple threads cannot access the function (and also the data manipulted by that function?) at the same time.

### Video #43 - Smart Pointers in C++

* std::unique\_ptr, std::shared\_ptr, std::weak\_ptr
* new allocates memory on the heap and delete us used to free it.
* Smart pointers are a way to abstract the the new/delete paradigm away. Some programmers even go so far as to say you should *never* use the new and delete keywords.
* Smart pointers are effectively wrappers around raw pointers.
* When you *make* a smart pointer, it will call new and allocate memory, and then (based on which type of smart pointer you use) that memory will automatically be freed when the smart pointer goes out of scope.
* A std::unique\_ptr is a *scoped pointer* that cannot be copied.
* A std::shared\_ptr stores a reference count and the object will only be deleted when that reference count goes to zero. Each time a new shared\_ptr is made to an existing object, the reference count increases by one.
* A std::weak\_ptr does not increase the reference count. Having a weak\_ptr to an object is basically a way of saying “I want to know if this object exists, but I don’t want to be the *reason* that it exists or continues to exist.”
* The rules for applying const to smart pointers and their underlying types looks similar to those same rules for [raw pointers](#video-33-const-in-c).

|  |  |  |
| --- | --- | --- |
| Smart Pointer Declaration | Raw Pointer Declaration | Analogous Declaration |
| shared\_ptr<T> | T \* | - |
| const shared\_ptr<T> | const \* T | - |
| shared\_ptr<const T> | T const \* | const T \* |
| const shared\_ptr<const T> | T const \* const | const T \* const |

### Video #54 - Stack vs. Heap Memory in C++

* The Stack has a much smaller pre-defined size (~2MB), whereas the Heap is much larger. **Both** exist in RAM, however the Stack may be *hot* in the *cache* because it is being accessed more frequently.
* Each program/process has its *own* Stack and Heap.
* Each thread will create its own stack, but the heap is shared among threads (hence the need for thread-safety in multi-threaded applications).
* Stack vs. Heap allocation: ``` // Allocate an int and an int array on the stack int value = 10; int array[10];
* // Allocate an int and an int array on the heap int\* heap\_value = new int; *heap\_value = 10; int* heap\_arr = new int[10];
* // Must manually free heap-allocated memory delete heap\_value; delete[] heap\_arr; ```
* **Reminder:** The lifetime of a stack-allocated variable is scope-based - whenever a scope is exited, all stack-allocated memory within that scope is freed.
* *Freeing* memory on the stack is the same thing as resetting the stack pointer back to the beginning of the stack.
* Heap allocation via the new keyword effectively calls malloc() under the hood and returns a pointer to a free portion of memory that is maintained by the *free list*.
* Heap memory == *dynamic* memory.
* **Takeaway:**
  + Allocating memory on the stack is effectively one CPU instruction, whereas allocating on the heap is *much* more expensive: call new -> call malloc -> consult the \_free list\_\_ -> update the *free list* -> … -> eventually delete the memory.
  + The performance difference *is* the allocation. Access *after* allocation is approximately equivalent (*cache misses* for heap-allocated memory can be the difference here).

### Video #71 - Safety in Modern C++ and How to Teach It

* *Safe* programming aims to prevent things like crashes, memory leaks (forgetting to free heap-allocated memory) and access violations. This video will focus on pointers and heap allocation.
* Why do we care? Because we want to write real-time, performance-critical production C++ code.
* With C++11, *smart pointers* were introduced to support this goal. In reality, the entire goal of smart pointers is to automate the use of delete.
* Note: shared\_ptr is **not** thread safe. Why?

### Video #84 - Track Memory Allocations the Easy Way in C++

* TODO:

### Video #105 - [Weak Pointers in C++ (std::weak\_ptr)](https://www.youtube.com/watch?v=M0GLQEfplxs&list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb&index=105)

* *“Weak pointers - the pointers for those of you who are just not quite strong enough to use proper pointers.”* - Cherno
* Weak pointers, std::weak\_ptr, are intended to be used with shared pointers, std::shared\_ptr.
* Shared pointers (std::shared\_ptr) refresher:
  + Recall that a shared pointer is a wrapper around a raw pointer to a heap-allocated object that implements a *reference counter*. ``` // Create a new heap-allocated Object managed by a raw pointer Object\* obj = new Object();
  + // Instead, use a shared pointer to automate the lifetime management (scope) of the Object instance. std::shared\_ptr obj1 = std::make\_shared(); ```
  + The real *magic* of std::shared\_ptr is that the object that the shared pointer manages will not go out of scope (be deleted) untill *every* reference to that shared pointer has gone out of scope.
  + Shared pointers maintain a **strong reference** to the object that they’re managing, i.e. the reference is *strong enough* to keep the underlying object instance alive (until the reference counter reaches zero).
* One strong use case for weak pointers is in the case of cyclical references, i.e. when you have 2 or more classes that maintain references to instances of the other class(es). If we choose to use std::shared\_ptr to maintain those cyclical references then the instances of A and B that we create will never go out of scope (their destructors will never be called): ``` // Forward declare B struct B
* struct A { std::shared\_ptr ptr\_b ~A() { std::cout << “A destroyed!” << std::endl; } }
* struct B { std::shared\_ptrptr\_a ~B() { std::cout << “B destroyed!” << std::endl; } }
* int main() { std::shared\_ptr a = std::make\_shared(); std::shared\_ptr b = std::make\_shared();
* a->ptr\_b = b;  
  b->ptr\_a = a;
* } ```
* (from YouTube comments) *“Writing your own smart pointers is probably one of the best learning exercises for C++ in my opinion… it teaches you a lot about* ***templating****,* ***object lifetime*** *and* ***operator overloading****, just to scratch the surface.”*
* (from YouTube comments) *“The ability to write your own smart pointer should be a fundamental skill that one learns when learning C++. Knowing how to do this teaches you* ***RAII****, which is useful for many things besides memory management. It also forces you the better understand the overhead of managing memory. If you can write a smart pointer for memory, you can do the same for managing any other resource as well.”*
* See app/105\_weak\_pointers.cpp for a more detailed example.

## C++ Advanced Topics

### Video #52 - How to deal with Multiple Return Values in C++

* How to deal with *tuples* and *pairs*.
* In C++, a function can return only *one* value.
* **Option 1:** One way to get around this is to have your function return void and instead pass in references to the objects you want to assign and set them via reference instead of actually returning anything. ``` void f(std::string& out\_str1, strd::string& out\_str2) { out\_str1 = “one”; out\_str2 = “two”; }
* int main() { std::string a, b; f(a, b) } ```
* **Option 1b:** As an alternative to passing by reference, we can pass a pointer, and this allows us the addional option of passing nullptr when we don’t actually want to set that value. ``` void f(std::string\* out\_str1, strd::string\* out\_str2) { if (out\_str1) { out\_str1 = “one”; } if (out\_str2) { out\_str2 = “two”; } }
* int main() { std::string a, b; f(nullptr, &b) } ```
* **Option 2:** We can return a heap-allocated std::array. Not really great option. Cherno isn’t a fan of this method, or of using std::array. ``` #include
* std::array<std::string, 2> f() { return std::array<std::string, 2>(“one”, “two”); }
* int main() { std::string\* result\_array = f(); } ```
* **Option 3:** Using *Tuples* and *Pairs*.
  + Ugly because accessing members of a tuple can only be done via std::get<index>(tuple\_var) or tuple\_var.first and we may want to be more explicit that this and actually name the members of the tuple for the sake of code readabilty.
* #include <tuple>  
    
  std::tuple<std::string, std::string> f() {  
   return std::make\_pair("one", "two")  
  }  
    
  int main() {  
   std::tuple<std::string, std::string> result = f();  
   // Now get the first (index zero) string from the tuple (2 options)  
   std::string first = std::get<0>(result);  
   std::string first = result.first;  
  }
  + Note that std::get<T>(std::tuple) can only be used on a tuple which has exactly one element of type T in it.
* **Option 4:** (Cherno’s favorite) create a struct (a *named tuple* in Python) that excplicitly contains the items that we want to return. ``` struct FileInfo { std::string directory; std::string filename; std::string extension; }
* FileInfo f() { // Take advatage of “implicit” conversion to create the FileInfo object return {“local\_dir”, “my\_file”, “txt”}; }
* int main() { FileInfo file\_info = f(); filename = file\_info.filename; } ```
* Refer to [Structured Bindings](#video-75-structured-bindings) for a better/cleaner/more modern way of dealing with multiple return values.

### Video #53 - Templates in C++

* A template allows us to get the compiler to write code for us based on a set of rules.
* A template is **not** a class or a function. A template is a **pattern** that the compiler uses to generate a *family* of classes or functions.
  + [Why can’t I separate the definition of my templates class from its declaration and put it inside a .cpp file?](https://isocpp.org/wiki/faq/templates#templates-defn-vs-decl)
* In template<typename T> the keywords typename and class can *almost* be used interchangeably. But typename is preferred.
* At compile-time, a templated function only gets created (and linked) for the particular types that it’s actually called within the source code.
* If a templated function is *never* called in your code, the the compiler never actually creates any versions of the function, and it’s possible to have errors in a templated function that can go undetected.
* The STL (Standard *Template* Library) is a collection of standarized templated classes.
* **Best Practice:** When to use templates:
  + Logging systems, buffers than need to contain various types.
* **Best Practice:** When not to use templates:
  + …

### Video #55 - Macros in C++

* Macros allow us to use the *pre-processor* to automate, “macro-ize”, some aspects of our code.
* All # statements are known as *preprocessor directives*. The Preprocessor step comes before compilation.
* Templates v. Macros:
  + Templates get “evaluated” at compile-time.
  + Macros are evaluated in the preprocessor step and strictly consist of “pure text replacement” which comes before compilation.
* Cherno doesn’t like overusing macros.
* **TODO:** Finish adding a command line CMake variable to enable/disable macros in my code.

### Video #58 - Function Pointers in C++

* This video will focus on C-style *raw* function pointers.
* Function pointers allow us to assign a function to a variable. Allows us to apply *logic* to functions and pass functions as arguments to other functions.
* Since functions are effectively just a set of instructions stored somewhere within our binary (executable), the start of those instructions will have a memory address. Thus, we can create a function pointer like:
* **Lambdas** are functions that are *anonymous* functions that are declared inline that can be useful when you don’t want to define a separate function.

### Video #59 - Lambdas in C++

* Anywhere that you use or require a function pointer in C++, you *can* use a lambda instead. But when is it good idea to do so?
* Declaring a lambda can look like: auto lambda = [ captures ]( params ) { body }; auto lambda = [ captures ] { body };
* *Captures:*
  + [a, &b] captures a by copy and b by reference.
  + [this] captures the current object (\*this) by reference.
  + [&] captures all *automatic* variables used in the body of the lamda by reference and the current object by reference if it exists.
  + [=] captures all *automatic* variables used in the body of the lamda by copy and the current object by reference if it exists.
  + [] captures nothing.
* Lambda are also known as a *closure*. Closures are unnamed function objects. Hence, we need auto to deduce the type of a closure. We don’t know its type but the compiler does.
* std::function is a general-purpose polymorphic function wrapper.
  + Lambdas closures as function objects can be converted to their respective std::function objects: std::function<void(int, int)> f = [](int some, int some2) { // do something }
* Lambdas can also be declared like: auto L = [](auto flag) -> auto { return flag ? Foo{"a"} : Fpp{"b"}; }; where the return type is inferred from the type of the operance of its return statement, in this case, the return type is Foo.

### Video #61 - Namespaces in C++

* Standard C does not support namespaces and so often times in C and C++ compatible libraries, like OpenGL, you’ll see function names that have prepended the library name, e.g. gl\_get\_vertex()
* Namespaces exist to avoid naming conflicts. That’s really it.
* :: is the *scope resolution* operator.
* Classes are themselves namespaces.
* You can create an alias for a namespace that may be too long to use efficiently, e.g. using rlln = ReallyLongLibraryName;
* Namespaces can be nested, e.g. outer\_ns::inner\_ns::func()
* Can do namespace ns\_name::name since C++17.
* **Best Practice:**
  + Avoid using namespace std at all costs!
  + If you *must* using namespace try to confine it to as small a scope as possible. Only as a last resort, use it at the top-level of a file.
  + NEVER using namespace in a header file! This is an easy way to create naming conflicts.

### Video #65 - Sorting in C++

* std::sort is C++’s built-in sorting algorithm.
* See app/sorting.cpp for an example using std::sort.

### Video #66 - Type Punning in C++

* Even though C++ is a *strongly-typed* language, *type punning* is just a fancy way of getting around the type system of C++.
* Something that C++ is really good at is raw memory operations (memory maipulation), and we can take advantage of that to do some kind of ridiculous operations. See app/type\_punning.cpp for an example.
* This can get us into trouble most of the time, but it can also be incredibly powerful. For example, type punning is used in Quake’s notoriously fast inverse square root function - it converts a float into a long in order to use bit manipulation.

### Video #68 - Virtual Destructors in C++

* Virtual destructors are useful when dealing with *polymorphism*.
* Remember the order of operations when it comes to instantiating and destroying an instance of a derived class - what we’ll notice for the derived class is that the Base constructor is called first, followed by the Derived constructor. When the Derived instance is destroyed, the reverse order occurs.
* In order to *guarantee* that the Base destructor is called, we can mark the Base destructor as virtual (see app/virtual\_destructors.cpp for an example).
* **Best Practice:** When creating an class that we intend to be extended, i.e. derived from or sub-classed, we **need to** declare its destructor as virtual. Otherwise, we cannot *safely* extend that class in any case where we want to treat the derived class as if it were an instance of the base class, e.g. when passing a derived object to a function that may delete the derived instance via a base class pointer.
* **Best Practice:** [When should we **not** use a virtual destructor?](https://stackoverflow.com/questions/300986/when-should-you-not-use-virtual-destructors)
  1. When we have no intention on deriving from the class.
  2. The class will never be instantiated on the heap.
  3. No intention to delete an derived instance via a base-class pointer. (But how can know that the codebase won’t be extended in some way in the future in such a way that would delete a derived instance via a base class pointer?) **Guideline #4:** (Herb Sutter) “A base class destructor should be either public and virtual, or protected and nonvirtual.”

### Video #69 - Casting in C++

* C-style vs. C++ style casting. C style casts are simple and look something like: double a = 5.25; double b = (int)a + 5.3 // b = 10.3
* C++ style casts are different and include static\_cast, reinterpret\_cast (i.e. type punning), dynamic\_cast and const\_cast. C++ style casts don’t actually do anything that C-style casts can’t achieve, but they are more or less syntactical sugar on tope of C-style casting.
* static\_cast helps to avoid removing constness that can be easily done through C-style casting.
* Say we have multiple derived classes from a single base class, and we’re using a base class pointer to deal with the derived objects. A dynamic\_cast can help us determine which derived type we’re actually working with: ``` class Base { … } class Derived : public Base { … } class AnotherDerived : public Base { … }
* int main() { Derived\* derived = new Derived() Base\* obj = derived;
* // 'another' will evaluate to NULL because obj is not a AnotherDerived instance  
  Derived\* another = dynamic\_cast<AnotherDerived>(obj);
* } ```

### Video #73 - Dynamic Casting in C++

* Dynamic casting is a C++ style cast that acts as somewhat of a “safety net” which ensures that the casting we’re doing is “valid”.
* dynamic\_cast can only be used when Runtime Type Information (RTTI) is enabled.
* If RTTI is enabled, dynamic casting happens at runtime, not at compile time which means it comes with some performance cost.
* Other *managed* langagues, like Python, have built-ins like isinstance() to achieve the same behavior.
* See example from [Video #69](#video-69-casting-in-c) README for how the failure of a dynamic\_cast can be useful.

### Video #75 - Structured Bindings

* Structed Bindings are a new feature to C++17 that allow us to handle [multiple return values](#X2c28d4289d0288e998f8468907920bb9334db51) in a cleaner way.
* Structured bindings allow us to “cleanly” return things like Tuples and Pairs.
* In [Video #52](#X2c28d4289d0288e998f8468907920bb9334db51), Cherno said that he preferred returning an instance of a struct that contains the members he wanted to return, but his opinion has changed somewhat to prefer returning tuples and/or pairs via *structured bindings*.
* Recall that when returning a std::tuple (or std::pair), accessing the tuple members was a bit ugly, and we had to use std::get and an index that wasn’t very human-readable: ``` std::tuple<std::string, int> create\_person(std::string name, int age) { return { name, age }; }
* std::tuple<std::string, int> person = create\_person(“Cherno”, 24); std::string& name = std::get<0>(person); int age = std::get<1>(person); ```
* std::tie offers a slightly cleaner implentation that we didn’t touch on before, but returning a struct with named members is still probably a better idea than this. std::string name; int age; std::tie(name, age) = create\_person("Cherno", 24);
* This is where *structured bindings* come to the rescue! auto [name, age] = create\_person("Cherno", 24);
* The advantage with using structured bindings is that we don’t necessarily need to create all sorts of unnecessary struct types that may only be used in a handleful of places. This allows us to declutter our namespaces and remove unnecessary types.
* See app/93\_iterators.cpp and app/100\_maps.cpp for examples of using structured bindings.

### Video #82 - Singletons in C++

* A *singleton* (a type of [design pattern](https://refactoring.guru/design-patterns/cpp)) is a class (or struct) that you intend to only ever have a single instance of.
* Examples of types of singleton classes:
  + A random number generator: Typically, we instantiate a random number generator once with a seed and then use that single instance (and seed) to generate sequences of random numbers.
  + A renderer: In a rendering or scene generation application, typically a single renderer will render all renderable objects (point sources, meshes, etc.) in the scene for each frame.
* Singleton classes really just behave like a namespace, i.e. a single class is just a set of “global” variables and static functions that may (or may not) act upon those global variables.
* Worth noting one important functional difference between a namespace and a singleton class. In a namespace, all of the data is initialized (loaded into memory when the program first starts). This might be undesirable sometimes when we don’t want all the data initialized until we actually need it. In contrast, singleton instances are only loaded during the first call to the Get() method.
* See tamasdemjen4242’s comment for even more detail on the functional difference between singleton classses and namespaces, including thread-safe vs. non-thread-safe behavior.
* See app/singleton.cpp for an example.

### Video #88 - Argument Evaluation Order in C++

* Consider the following simple example. What do we think will be printed? It turns out that in C++, this toy scenario results in *undefined behavior*, i.e. the behavior will be different from compiler to compiler. ``` void print\_sum(int a, int b) { std::cout << a << ” + b << ” = ” << a + b << std::endl; }
* int main() { int val = 0; // Any of the following will produce “undefined behavior”. print\_sum(val++, val++); print\_sum(++val, ++val); } ```
* Note that the C++ compiler can evaluate certain expressions at compile-time, e.g. the C++ compiler is smart enough to replace int a = 1 + 2; with int a = 3; at compile-time rather than do the sum operation at runtime.
* In C++17, the C++ standard added rules for the evaluation of *postfix-expressions*, e.g. the post-increment operator ++, that state that multiple post-fix expressions must be evaluated sequentially (rather simultaneously at compile-time) but this rule doesn’t actually lead to a deterministic evaluation of print\_sum(val++, val++);.

### Video #96 - Intro to Binary and Bitwise Operators in C++

* TODO

### Video #97 - Bitwise AND, OR, XOR and NOT (&, |, ^, ~) C++

* TODO

### Video #101 - [What Exactly is NULL?](https://www.youtube.com/watch?v=PksUUwvq-po&list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb&index=101)

* In *managed languages* like C# or Java, if you try to use an object that’s NULL, you’ll get a NullReferenceException (C#) or a NullPointerExeption (Java). That’s because these languages do some “hand holding” to help you from fucking up too badly. But what about C++?
* What is the *value* of NULL in C++?
  + void\* value = nullptr;
  + If we take a look at the *value* of value, we’ll see that it’s 0x00000000, i.e. 8 bytes of all zeros.
  + Why 8 bytes? Because this example was built on a 64 bit system, and **any** pointer (or memory address) on a 64 bit system will **always** be 8 bytes. On a 32 bit system, pointers are 4 bytes.
* What about NULL (as opposed to nullptr)?
  + NULL is from C, however, it’s totally acceptable to use it in C++ too.
  + Let’s look at the definition of NULL: #ifndef NULL #ifdef \_\_cplusplus #define NULL 0 #else #define NULL (void \*)0) #endif #endif
  + So really, NULL (in C++) is just an integer defined as 0.
  + But be careful, because 0 and 0x00000000 are **not** the same thing. One is an integer (not a valid memory address) and the other is a memory address (a pointer).
* Let’s look at an interesting example that will teach us more about the role of the C++ compiler. Consider the code from the app/101\_null.cpp app. ``` class Entity { public: Entity() = default;
* const std::string& name() const { return name\_; }  
   void print\_type() { std::cout << "Entity\n"; }  
    
  private:  
   Entity\* parent\_;  
   std::string name\_;
* }; ```
  + The code above is what we’re used to in C++. But to understand the role of the compiler better, it’s useful to understand how this class would be implemented in C (where classes do not exist).
  + In C, since there are no classes, we would create EntityData as a struct and then implement its member functions separately because structs in C cannot have member functions.
  + Surprisingly, this separation of data and member functions is exactly what the C++ compiler creates for us!
  + **In C++, the compiler converts member functions of a class to “regular” free-floating functions (that exist outside of the class) that take in an instance of the class as their first argument**. That instance is referred to as the this keyword! ``` const std::string& name(const EntityData\* self) { return self->name\_; }
  + void print\_type(EntityData\* self) { // Note that if self == nullptr, this has no effect below std::cout << “Entity”; }  
    ```
  + So now we understand that when we do the following, we’re not *actually* calling a function that exists *within* with Entity class. We’re actually calling out to a *stand-alone* function at some location within our compiled binary, and calling that particular function is totally valid and will not crash since self is never dereferenced by the print\_type() member function. Entity\* entity = nullptr; entity->print\_type();

### Video #103 - [Conversion Operators in C++](https://www.youtube.com/watch?v=OK0G4cmeX-I&list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb&index=103)

* Let’s consider a simple example. ``` struct Orange { operator float() { return 5.5f; } }
* void print\_float(float value) { std::cout << value << “”; }
* int main() { Orange orange;
* // Conversion operators allow for the following to work  
  print\_float(orange)  
  std::cout << float(orange) << std::endl;
* } ```
* (from YouTube comments) *“Pretty much 99% of conversions operators should be marked explicit (even boolean ones) to prevent weird bugs and unexpected conversions from happening.”*
* (from YouTube comments) *“Like with single argument constructors, the default should be to add the explicit keyword and only omit if absolutely necessary.”*
* See app/103\_conversion\_operators.cpp for more detailed examples.

## Performance & Benchmarking

### Video #62 - Threads in C++

* Parallelization!
* Use std::thread to start some process, e.g. call a function, in a new thread.
* Use std::thread::join to wait for a worker thread to complete before resuming execution of the current thread, i.e. the thread that the worker thread was kicked off from.
* See app/threading.cpp for an example.

### Video #63 - Timing in C++

* Since C++11, we have std::chrono to help us with timing, i.e. understanding how much time has elapsed between various lines of code.
* See app/threading.cpp to see how the Timer struct is used as a scope-based (lifetime based) timer to function profiling.
* The topic of *benchmarking* will come later and will be more in depth. Will discuss *instrumentation*, i.e. modifying source code to contain profiling tooling.

### Video #74 - Benchmarking in C++

* Performance differences between std::make\_shared vs. std::shared\_ptr<type> (new type) vs. std::make\_unique.
* Important to perform this benchmarking in Release mode and **not** Debug.
* Note that we expect std::make\_shared to be faster than std::shared\_ptr because std::make\_shared performs one heap-allocation, whereas calling the std::shared\_ptr constructor performs two. But as per the results of app/benchmarking.cpp, this isn’t always true.

### Video #79 - How to Make C++ Run Faster with std::async

* How can we take advantage of parallel processing, i.e. use multiple CPU cores?
* MSVS supports an interesting visualization called “Parallel Stacks”.2
* The following is an example snippet from Cherno’s game engine that will asynchronously load a list of textures in parallel. ``` #include #include
* static std::mutex s\_meshes\_mutex;
* // Note that filepath is passed by copy because there’s a possibility that the // ‘mesh\_filepaths’ variables from ‘EditiorLayer::LoadMeshes()’ may have gone // out of scope while this function is still being called asynchonously. static void load\_mesh(std::vector<Ref>\* meshes, std::string filepath) { auto mesh = Mesh::load(filepath)
* // Q: Can we actually 'push\_back' onto a vector concurrently? No, this is  
  // where "mutuex" and "locks" come into play in order to "lock" our resource  
  // (the vector) while one thread accesses/modifies it to prevent another  
  // thread from doing the same.  
    
  std::Lock\_guard<std::mutex> lock(s\_meshes\_mutex);  
  meshes->push\_back(mesh);  
    
  // Note that std::Lock\_gaurd is scope-based and will "release" the lock on  
  // the s\_meshes\_mutex once the lock goes out of scope (the end of this   
  // function).
* }
* class EditorLayer : Layer { public: enum class PropertyFlag { None = 0, ColorProperty = 1 };
* EditorLayer();  
   virtual ~EditorLayer();  
   ...  
    
  private:  
   std::vector<std::future<void>> m\_futures;
* }
* void EditiorLayer::LoadMeshes() { std::ifstream stream(“data/models.txt); std::string line; std::vector mesh\_filepaths; while (std::getline(Stream line)) { mesh\_filepaths.push\_back(line); }
* // Load each mesh in serial  
  for (const auto& filename : mesh\_filepaths) {  
   m\_meshes.push\_back(Mesh::load(file));  
  }  
    
  // Load each mesh in parallel  
  for (const auto& filename : mesh\_filepaths) {  
   // std::async actually returns a std::future object, and we need to store  
   // that "future" object. Why? ...  
   m\_futures.push\_back(  
   std::async(std::launch::async, load\_mesh, &m\_meshes, file));  
  }
* } ```

### Video #80 - How to Make Your Strings Faster in C++

* TODO:

### Video #83 - Small String Optimizations in C++

* TODO:

## Storing Multiple Types of Data

### Video #76 - How to Deal with Optional Data in C++

* New to C++17 is std::optional.
* std::optional<T> **cannot** be used with references. However,boost::optional<T> can handle references. This is very helpful if you want to check for existence of a bigger type in a map like boost::optional<const Element&> GetElementAtPos(int x, int y). With std::optional you cannot do that and would have to return a pointer. The reason why it’s unavailable in std is a bigger topic itself and you can read more online.
* value\_or(default\_value) is also designed with backwards compatibility in mind. If new code parts want to use std::optional<T>, but need pass T to an old method (where a magic value represents “not set”) you can pass value\_or(magic\_value) to it to still be compatible with this old code part.
* boost::optional brings this concept even further and provides a constructor boost::optional<T>(bool isSet, T value) that can be used to filter out magic values on construction. So if an old method returns T(with either a value or a magic value “not set”) you can initiate your variable like boon b.st::optional<T>(result != magic\_value, result). This constructor is sadly not available in std::optional though.

### Video #77 - Multiple Types of Data in a Single Variable in C++

* New to C++17 is std::variant. Similar std::optional in the sense that is allows us to not worry so much about the underlying data type, and be more concerned with if that data is actually available or not.
* Allows us to create a variable that can be one of multiple types, e.g. we can declare a value that will either be a string or an int: ``` #include
* std::variant<std::string, int> data; ```
* We can use std::variant::index to determine which type the data *actually* is. In the example above data.index()=0 means that data is a std::string and data.index()=1 meands that data is an int.
* Alternatively, we can use std::variant::get\_if to return a pointer to our data that will be NULL if the data is not the type we requested. auto value = std::get\_if<std::string>(&data);
* Are variants just “type-safe unions”? Short answer: not exactly. The sizeof() a union will be equal to the size of its largest type, whereas the size of a variant will be the combined size of all of its types, e.g. sizeof(std::variant<std::string, int>) == sizeof(std::string) + sizeof(int).
* **Best Practice:** Prefer variants to unions because they are *type safe*.
* Could use std::variant as an alternative to std::optional when we want to be more specific about what may have gone wrong when evaluating a function. See app/optional.cpp for an example of using std::variant to possibly return an error code enum type.

### Video #78 - How to Store ANY Data in C++

* New to C++17 is std::any. We can use it store *any* type of data in a single variable (technically possible with a void\*, but this is a C++17-safe way of doing it).
* Remember, std::variant is effectively a type-safe std::union, but they differ in size. However, std::any behaves differently for “small” and “large” types. For small types, std::any stores its data as if it were a union, but for large types (< 32bytes on MSVC), std::any will perform a dynamic memory allocation to store the larger data type (unecesary heap alloccations are something we want to avoid).
* **Best Practice:** Probably don’t ue std::any. “If you need to store multiple data types in a single variable, use std::variant because it’s type-safe and it **won’t** perform dynamic memory allocation. If you actually *need* a variable that can store *any* type of data, probably rethink you’re program design.”
* **Best Practice:** “Use std::any where in the past you would have used void\* or shared\_ptr<void> (which solves the problem of lifetime management that void\* has). Which is to say, ideally, almost nowhere.” - [StackOverflow](https://stackoverflow.com/questions/52715219/when-should-i-use-stdany)
* [Further discussion](https://devblogs.microsoft.com/cppblog/stdany-how-when-and-why/)

## Move Semantics

### Video #85 - l-values and r-values in C++

* (and also l-value and r-values references).
* Often (but not always) an l-value is on the left of the = sign and an r-value is on the right of the = sign. For example, in int i = 10, i is an l-value and 10 is an r-value.
* An l-value has a location in memory, and an r-value is simply a *temporary value* that has no memory allocated to it. An r-value can be a literal, like 10 or it can be the return value of a function. In all cases, we cannot assign another r-value to an r-value.
* You cannot create an l-value reference, e.g. int& from an r-value. You can only create an l-value reference from an existing l-value.
* A special rule (related to const): You can create a const l-value reference from an r-value, e.g. const int& val = 10;. This allows us to pass either an l-value or an r-value to a function like: ``` void set\_value(const int& value) { // do something with value }
* int main() { // Create an l-value (i) int i = 10;
* // Call set\_value with an l-value  
  set\_value(i);  
    
  // Call set\_value with an r-value (only possible because set\_value takes a  
  // const l-value reference)  
  set\_value(10);
* } ```
* Let’s look at one more example where first and last are l-values, but first + last is an r-value because it is a temporary object that gets created and then assigned to the l-value full. ``` void print\_name(const std::string& name) { std::cout << name << std::endl; }
* std::string first = “Yan”; std::string last = “Chernikov”;
* // Assign the r-value “first + last” to the l-value “full”. std::string full = first + last; print\_name(full);
* // Only works if print\_name accepts a const l-value reference. print\_name(first + last); ```
  + Note that a function that accepts both l-values and r-values MUST be written to accept a const l-value reference. This is why you’ll see a lot of const references being used in C++.
  + Do we have a way to write a function that only accepts temporary objects (r-values)? Yes! We need to use something called an *r-value reference*. We can modify the code above such that print\_name accepts an r-value reference.
* void print\_name(std::string&& name) {  
   std::cout << name << std::endl;  
  }  
    
  // The line below will throw the error "An rvalue reference cannot be bound to an lvalue."  
  print\_name(full);
* Being able to distinguish an r-value from an l-value is important in the context of *move semantics* and optimization. If we know that we are dealing with a temporary object (an r-value reference), then we don’t have to worry about things like making sure we keep it alive, etc.

### Video #89 - Move Semantics in C++

* C++11 introduced *r-value references* which are necessary for implementing *move semantics*.
* Consider the case where we need to create an object and then pass it to some function that will take ownership of that object. Prior to C++11, this would require us to create a “throw away” object in the current stack frame and then copy that object to the fuction that’s receiving it. This creates unnecessary copying and modern C++ should allow us to avoid unnecessary copies.
* So how do we *move* an object rather than copying it?
  + We must implement a *move constructor* for the class that we wish to support moving.
  + We need to use std::move to invoke that class’s move constructor.
  + See app/move\_semantics.cpp and the String class in types.h for an implementation of a \_\_move constroctor\_ \_and use of std::move.

### Video #90 - std::move and the Move Assigment Operator in C++

* The **move constructor**, Type(Type&& other) is invoked when constructing a new object and passing it as an r-value reference.
* The **move assignment operator** is invoked when we want to *move* an existing object (an x-value near the end of its lifetime?) into another existing object.
  + **If we define a move constructor for our class, we *should* also define the move assignment operator**. This is referred to as the **Rule of Fifths**. More on this later.
* std::move is **really just a cast** and is used in place of (Type&&)source to cast an l-value to an r-value (actually an x-value) and deduces the moved-from type at compile-time rather than requiring the user to cast the l-value to an r-value manually, i.e. (Type&&).
* std::move and const:
  + A const object, by definition, **cannot be modified**. This includes transferring its resources to another object, as that would essentially be modifying it.
  + Attempting to std::move a const object will result in a **copy operation** *unless* the move contructor for that type is explicitly deleted, then it will result in a compilation error.
* const std::string str = "Hello";   
  std::string other = std::move(str); // This is a copy, not a move!

## Workflow & Debugging

Add a section that groups together videos about workflow and debugging.

### Video #49 - Using Libraries in C++

* The ethos: If you download my repo from github, that repo should contain everything you need for it to compile and run.
* This video: Learning to link against binaries.

### Video #50 - Using Dynamic Libraries in C++

* Static linking happens at compile time.

### Video #51 - Making and Working with Libraries in C++

* TODO:

### Video #70 - Conditional and Action Breakpoints in C++

* TODO:

### Video #72 - Precompiled Headers in C++

* TODO:

### Video #86 - Continuous Integration in C++

* TODO:

### Video #87 - Static Analysis in C++

* How do we write *better* code, i.e. code that produces fewer bugs.
* How do we use a *static analyzer* to improve our code?
* TODO: Finish this video.

## Writing Our Own Data Structures

### Video #91 - ARRAY - Making Data Structures in C++

* Finally time to take what we’ve learned and write out own *data structures*, e.g. arrays, lists, sets, maps, trees, etc. The STL implements most of these data structures for us, but we can learn a lot by trying to implement our own (and maybe even make them fast/ more efficient than the STL data structures).
* In this video, we’ll be implementing our own version of std::array - a fixed-size, stack-allocated array data structure in C++.
* Recall some differences between std::array and std::vector:
  + std::vector always allocates on the heap and can be dynamically resized, whereas a fixed-size std::array allocates its memory on the stack.
  + **Best Practice:** If you don’t need heap allocations, don’t use them because they’ll just slow things down.
  + std::vector can be dynamically allocated, i.e. its size does not have to be determined at compile time, but a std::array must define its size at compile-time: ``` // This is just fine size\_t size = get\_size(); int\* heap\_arr = new int[size];
  + // This throws an error: “Expression must have a constant value.” int array[size];
  + // However, we can use the size of an array to set another array’s size std::array arr1<int, 10>; std::array arr1<float, arr1.size()>; ```
* See include/array.h and app/custom\_array.cpp for implementation and use of our custon Array class.

### Video #92 - VECTOR/DYNAMIC ARRAY - Making Data Structures in C++

* STL’s std::vector has the following important characteristics:
  + std::vector is a re-sizeable array.
  + std::vector is heap-allocated (as opposed to std::array which is stack-allocated and fixed in size at compile-time).
* What do we need to implement our own vector (dynamic array) class?
  + A pointer to the beginning of a block of heap-allocated memory.
  + The ability to resize our vector when we run out of room to *push back* a new element. This requires allocating a new block of memory on the heap, copying over the contents of the current vector, and then freeing the memory that was copied from.
* Resizing strategies:
  + Maybe revisit [Video #47 - Optimizing the Usage of std::vector](#X5a97ba28eeb57400d84e30a106d667d2bdcd2bd).
  + Instead of copying, we can *move* the contents of the old vector into the newly resized vector.
  + (13 -10) Note that in more *sophisticated* dynamic array (vector) implementations, you have the option of specifying a *custom allocator* that may not necesarily hit the heap each time it needs to resize. This is beyond the scope of this video, but it can be really useful if you’re writing a custom piece of software.
* Two options for adding elements to our Vector:
  + push\_back: Used to add an element to the Vector container by either copying or moving. Moving should be preferred.
  + emplace\_back: Rather that constructing an instance of the element in the stack frame of the calling function and then *moving* it into our Vector data storage, instead we construct the new element *in place*, i.e. in the memory that we’ve already allocated for it in Vector::data\_ by simply taking the arguments that we’ll pass to the constructor the element type.
    - **Best Practice:** Prefer *placement* new for *truly* constructing objects *in place*.
* **Operator new and Operator delete:** (::operator new and ::operator delete)
  + *“A lot of care needs to be taken when you manually call the destructor of the objects in your container”* - Cherno.
  + We do this for both Vector::pop\_back and Vector::clear. For a type that doesn’t perform any heap allocation, like the Vec3 class, we won’t really run into any issues calling ~Vec3 manually, but as soon as type that your container supports does do some sort of heap allocation, you can very quickly run into issues.
  + “When you write p = new T[N], the compiler generates code that calls operator new[] to allocate enough memory for N objects of type T plus whatever book-keeping information it needs. When you subsequently call delete[] p, the compiler calls the destructor for each of the N elements in the array that p points to, and then calls operator delete[] to release the memory that it got from operator new[].” - [SO](https://stackoverflow.com/questions/50069257/why-does-operator-new-allocate-memory-for-the-size-of-the-array)
* Choosing to include the Copy Constructor and CopyAssignment operator for the Vec3 class (when Cherno decided to delete them) forced me to revisit some topics related to [move semantics](https://stackoverflow.com/questions/3106110/what-is-move-semantics).
  + [**The Rule of Three**](https://en.wikipedia.org/wiki/Rule_of_three_%28C++_programming%29): If a class defines any of the following then it should probably explicitly define all three.
    - Desstructor
    - Copy Constructor
    - Copy Assignment Operator
  + [**The Copy-And-Swap Idiom**](https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom)**:** Make a copy (typically of a heap-allocated member variable), swap the contents with the copy, and then get rid of the copy by leaving the scope.
  + **RAII:** There is tie-in to the concept of RAII (Resource Acquisition is Initialization) here in the sense that in the Vec3 class, we have chosen to manually manage our own memory via new[] and delete[].

### Video #94 - [Writing an Iterator in C++](https://www.youtube.com/watch?v=F9eDv-YIOQ0&list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb&index=94)

* We’ll be adding an iterator to our custom vector class (from [video #92](#Xd52cd5712317f120adb0445afeb64f74d0b1a7c)).
* An aside: How do we actually get better as a c++ developer?
  + More focus/emphasis on reading and writing real-world code rather than just focusing on textbooks and tutorials.
  + Eventually you’ll get to the point where you’ll only continue to learn and get better by looking at and working with an existing codebase.
  + For example, the std::vector template class implements an iterator. So if we want to implement an iterator for our own custom class, we *should* be looking at the STL for guidance and using it as an example.
* In this video, we’ve written an interator for our custom Vector class, but this isn’t particulary difficult because it really just boils down to incrementing a pointer. However, the concept of an iterator applies to *any* data structure, e.g. a graph, tree, map, etc.
  + For example, with a graph data structure, we may want to update the ++ operator to visit a child node and move down (or up) the data structure in some hierarchical way.
  + **TODO:** Implement an iterator for our custom Array class, include/array.h.
  + **TODO:** Take a look at the iterator for std::unordered\_map.

### Video #95 - How to Really Learn C++

* What should I do next in my C++ learning journey? A simple answer: *open source projects*.