The C++ Programming Language

Preface:

This document covers the C++ programming language in as much detail as possible. Please note that C++ is an advanced programming language. In fact, I’d highly recommend that you learn two languages prior to learning C++ - preferably an OOP language like Java and a procedural language like C. If you are insistent on learning C++ as your first language, I suggest you check out Chernos playlist on YouTube. I do have a soft spot for C++ as a language, but perhaps its biggest flaw is that it has too many features to its detriment. There are so many ways to go about solving a problem in C++, that it tends to lead to analysis paralysis. It is also a very syntactically rich language and uses a lot of special characters for semantic purposes, which often makes it difficult to read. Having some prior experience with other languages will certainly come in handy when dealing with these difficulties.

Introduction:

C++ was created in 1979 by a gentleman named Bjarne Stroustrup. Stroustrup was born in 1950 and worked for Bell Laboratories. Denis Richie and Ken Thompson also worked for Bell labs and invented the C programming language. Stroustrup likely felt that C was lacking and sought to expand it. C++ was originally called “C with Classes”. Its original goal was to stay the same as C, but simply add OOP concepts like classes, methods, inheritance, and polymorphism. C++ was not readily available for public use until about 1985. In the present, most people have shifted over from C to C++. This is due to the fact that C++ can do everything C can do using essentially the exact same syntax, while also offering the flexibility to use more modern features of the language if needed. C++ is a very general purpose programming language that can create a wide range of applications. Examples might be game engines, operating systems, web backends, etc. With an idea of what C++ actually is, we can begin to look at some of the differences between it, and C.

The Myth of Backwards Compatibility:

It will sometimes be said that C++ is backwards compatible with C. This is simply untrue. C is also not a subset of C++ - they are two distinct languages that have deviated more and more over time. I say this because, even though you can write programs that would run in both a C compiler and a C++ compiler, this does not mean that all C programs will compile under a C++ compiler. There are things in the C language that are not present in C++, especially in newer versions of C.

Basic I/O:

Let’s begin with printing text to the console. Outputting text in C++ looks something like the following:

std::cout << “text”;

std is a C++ namespace and the two colons :: are used to denote scope. std is C++’s standard library (libstdc++) namespace, meaning that it is the namespace which encapsulates the definitions of entities/symbols defined in the standard library. cout and cin are symbols defined in iostream, which is a header file (C++ headers don’t need to have a .h extension). We can include this header file using, you guessed it, the #include directive e.g. #include <iostream>. cout stands for character output stream. Its type is ostream (output stream) which is a typedef for basic\_ostream, which is a C++ class. If you’re familiar with Linux, you will recognize this << as being the redirection operator. std::cout << “text”; can be read as ‘redirect the string “text” to the character output stream (stdout). Without the inclusion of the std namespace, C++ would search what is called the global namespace, which includes anything that is defined globally. The global namespace can also be referenced by prepending the scope operator without a name e.g. ::foo would look for foo in the global namespace. Though the global namespace is not explicitly defined, it is not an *unnamed namespace*.

When it comes to std::cout, we can use std::endl (end line) which will do two things: insert a newline feed, as well as flush the stream. It is acceptable to just manually add a newline character to the end of your string, however, endl is generally considered cleaner e.g. std::cout << “Hello world!” << std::endl;. Notice how we redirect multiple things to std::cout. This is akin to concatenating a string. For example, we can do something like:

std::cout << “Number is :” << num << std::endl;

This will output the text “Number is: “ followed by the value of num as a string.

In order to read from stdin, we can do something like: std::cin >> var;. stdin will await the enter key and will then redirect the contents of the stdin buffer to var in this case. Two things to note here: 1) Notice that the redirection operator points towards the variable, whereas, with std::cout, the redirection operator pointed towards the stream. This is pretty intuitive but easily forgotten. 2) std::cin does some special trickery to avoid runtime errors when accepting user input. Programmers need to be cautious of this, because std::cin might not return what the user actually entered. The way std::cin works when converting user input to an integer type is to extract all digits up until the first non-digit character. For example, if redirecting stdin to an int, and the user enters something like 12a34b56, std::cin will extract 12 and discard the rest. If the user enters no digits e.g. abcdef, std::cin returns 0. A similar concept applies to floats, with the difference being that std::cin will accept a period as being the decimal point for the float. C++ can recognize that certain types will always fail at compile time. For instance, attempting to redirect stdin to an object will fail at compile time (there are exceptions to this if we use operator overloading, which we will discuss later).

Primitives:

C++ shares all of the same primitive types as C except that booleans have been added into the language. These are declared with the bool keyword.

std::string, std::vector, and std::array:

Though C++ supports C-style strings and C-style arrays, it is considered more idiomatic to use std::string, std::array, and std::vector when defining strings, arrays, and dynamic arrays respectively. There are benefits to using these since C++ collections are polymorphic and share a lot of the same behaviours. Whichever you decide to use, just be consistent in your codebase, since swapping between C-style strings or arrays and C++ strings or arrays gets really messy really fast.

If you’re familiar with other OOP languages, the std::string, std::vector, and std::array classes should feel pretty familiar to you. std::string is actually a typedef for basic\_string<char>. Since basic\_string accepts a generic type (we’ll discuss this more later in the document), we can have other kinds of strings that behave exactly the same but with different character encoding widths that vary depending on platform. For example std::wstring is a typedef for basic\_string<wchar\_t> which is used for strings that need to support unicode. We also have u8string, u16string, and u32string. As is the case for most collections in C++ (including std::array and std::vector), there is support for iterators via the begin() and end() member functions. Likewise, rbegin() and rend() return a reverse iterator. We have modifier member functions such as clear(), insert(), push\_back(), pop\_back(), append(), replace(), copy(), resize(), etc. and also methods for searching substrings like find(), rfind(), find\_first\_of(), find\_last\_of(), etc. C++11 even added support for converting to numerics using the familiar sto\* functions e.g. stoi(), stof, stol. C++ strings can be indexed akin to C-style strings using the index operator to capture a character at the provided offset, or alternatively you can use the at() function which does the same thing. std::basic\_string has many, many constructors that I won’t cover here, but perhaps the most notable overload is the one that allows us to convert from a C-style string to a C++ string. Here’s an example:

const char \*cstr = “foo”;

auto cppstr = std::string(cstr);

We can also convert from a C++ string into a C-style string using the c\_str() member function. As you may be able to tell, C++ strings give us a lot of powerful operations and capabilities.

std::array is meant to be something of a replacement for C-style arrays. Personally, I don’t tend to lean towards using std::array, since I find C-style arrays to be a bit simpler, however, std::array is beneficial if you know that you’ll be repeating operations like reverse, swap, or copy a lot. std::array by itself actually does not provide that many member functions (only fill() which is equivallent to memset and swap() for swapping contents with another array). This is where the algorithm library comes in handy in extending the capabilities of std::array by adding things like sort(), max(), min(), and a dozen others. std::array is, once again, a templated class which accepts the type that will be used for the elements of the array, as well as the size of the array, since arrays must be statically sized at compile time in C++, same as in C. Creating an array in C++ might look something like:

auto arr = std::array<float, 3>{ 1.0f, 2.5f, 3.0f };

Note that in C++, objects which are able to be assigned initializer lists can instead use the above syntax where the initializer list directly proceeds the type. Here are other valid ways to initialize a std::array:

auto arr = std::array<float, 3>({ 1.0f, 2.5, 3.0f }); // Initializer list as an argument

std::array<float, 3> arr = { 1.0f, 2.5f, 3.0f }; // C-style assignment

std::array arr{ 1.0f, 2.5f, 3.0f }; // Generic type T is inferred

std::array<float, 3> arr{}; // Capacity is 3 but array is initially empty

std::vector is used for dynamically sized arrays, but is syntactically very similar to std::array. Unlike std::array, std::vector has a resize() function, as well as a couple more member functions like pop\_back(), push\_back(), erase(), insert(), etc.

C++ Header Files:

The standard C++ library has copies of all of the standard C library headers. An exhaustive list can be found here: <https://en.cppreference.com/w/cpp/header>. You’ll notice that some headers in C++ begin with the prefix ‘c’. This means that it is a compatibility header and is identical to its C counterpart. For example, #include <assert.h> is equivallent to #include <cassert> as they contain the same contents. I recommend always using the C++ compatibility counterpart over the libc header. Other headers such as stdio.h and threads.h are replaced with completely separate implementations, but that are usually functionally similar e.g. iostream is similar to stdio.h and thread is similar to pthread.h. Common stand-alone headers which don’t have C equivallents include fstream for file manipulation, iterator (for iterators), chrono for time (there is a compatibility header for time.h called ctime, but I recommend using chrono instead), algorithm for datastructures and sorting/comparisons, memory for smart pointers, etc.

As I mentioned earlier, C++ header files can optionally omit the .h extension. They also happen to be able to accept both the .h extension, as well as the .hpp extension. Source files do not abide by the same rules, as they must always have the .cpp extension.

Classes and Objects:

This section of the document is where we are going to begin getting a bit more in-depth on the OOP aspects of C++. First, let us look into how C++ uses header files and source files perhaps a bit differently than in C. Though not a strict requirement of the language, the most common practice is to define classes and their respective member function declarations/signatures within a header file, and then define these member functions within the source file. I personally try to avoid mixing header files intended for classes with other general purpose header files. This is mostly because I like to name my header files containing class declarations after the class, similar to Java. This is not, however, a requirement of the language in C++ (unlike Java). I will also give the source file with class definitions the same name. I find this just makes the codebase a little easier to read and maintain. You do have a lot of flexibility outside of this though. For instance, we don’t actually need header files at all. You could declare a class within a source file and also declare all of its member functions in-place, without separating them into declaration and definition. With that said, this is my document, so I’m going to teach things in the manner that I do them. Here is an example of a header file for an Animal class:

class Animal

{

private:

// Member variables

std::string m\_name;

bool m\_has\_fur;

double m\_avg\_dist\_per\_min;

public:

// Constructor(s)/destructor

Animal(std::string name, bool has\_fur, double avg\_dist\_per\_min);

~Animal();

// Member functions

std::string get\_name();

bool get\_has\_fur();

double get\_avg\_dist\_per\_min();

};

This class might reside in a file called animal.h (although it could just as easily reside in a source file as well). C++ allows us to define public and private segments within both classes and structs. The private specifier is assumed if no specifier is given (something we will discuss more shortly), however, I personally think it’s still good to explicitely declare your variables as being private because it takes no time and improves readability. We might use private for variables or methods which should only be accessible to the class itself, as well as subclasses/child classes, whereas public can be accessed from any file or class. In animal.cpp, we would then define the actual code block/body for each of the declared functions in animal.h.

#include “animal.h”

Animal::Animal(std::string name, bool has\_fur, double avg\_dist\_per\_min)

{

m\_name = name;

m\_has\_fur = has\_fur;

m\_avg\_dist\_per\_min = avg\_dist\_per\_min;

}

Animal::~Animal()

{}

std::string Animal::get\_name()

{

return m\_name;

}

void Animal::get\_has\_fur()

{

return m\_has\_fur;

}

double Animal::get\_avg\_dist\_per\_min()

{

return m\_avg\_dist\_per\_min;

}

Note how each method definition in the animal.cpp file starts with the Animal namespace even though we never explicitly declared an Animal namespace. This is because every class in C++ creates its own namespace to avoid namespace collisions for functions that share the same name across different classes. Even though we include the animal.h header file, the default behavior of the compiler when it comes across a function definition is to search for its definition in the global namespace. This is a similar issue to when we tried accessing cout and cin without the std namespace.

Let’s discuss instantiating the Animal class. There are technically 3 ways to instantiate an object in C++, though the third has a bit of a catch. Let’s look at an example of each:

#include <iostream>

#include “animal.h”

int main()

{

Animal cat(“cat”, true, 1.2); // Method 1

Animal dog = Animal(“dog”, true, 2.3); // Method 2

Animal bird = new Animal(“bird”, false, 6.8); // Method 3

delete bird;

}

The first method is something of a shorthand because it avoids the unnecessary repetition of method 2, where we first declare the type as Animal and also assign dog the instance returned by the Animal constructor. Method 1 does both the declaration and assignment in one step. Note though that Method 1 has a slight variation when using the default constructor. For example, Foo foo(); will fail at compile time because the compiler will think that this is a function declaration. For the default constructor in particular, the correct syntax is to just omit the brackets like so: Foo foo; In the case of cat, we are using a parameterized constructor, so the brackets are required. The first and second method share something in common which the third does not share, and that is that the first and second methods allocate the instance of Animal on the stack, whereas the new keyword used in method 3 creates a heap-allocated instance. See the next section for an explanation of the new and delete keywords.

The new and delete keywords:

Unlike most modern OOP languages, C++ does not have a garbage collector. In C, we used malloc() and free() to invoke the allocator and deallocator respectively. Instead of malloc() and free, C++ uses new and delete. The new keyword, when used for creating an instance of an object, will first allocate the appropriate amount of memory on the heap required for storing the object. Once the memory is allocated, the constructor is then invoked. Likewise the delete keyword will first invoke the object’s destructor, and then call the default deallocator to free the object. Although malloc() and free() are available in C++, it is generally advisable that you use new and delete for anything that needs to be heap-allocated. This is especially true for objects, since they may do important cleanup within the destructor.

Function Nuances in C++:

The next topic I’d like to cover is default arguments. I really like this feature in C++. Essentially if you have a function that has a particular parameter which will have the same value passed in most of the time, you can set that argument to have a default value so that if no argument is passed when calling the function, it presumes the default value. This is quite intuitive, we just set the parameter = to the value we want eg. void func\_foo(int param1, int param2 = 5) {...} Now if we call func\_foo() with only one argument eg. func\_foo(3), param2 will be 5, but if we call func\_foo() with both arguments, param2 will use the one we provided. Similar to var args (variable arguments), default arguments need to be placed last in the set of parameters. ie. you cannot have a default parameter followed by a non-default parameter.

A third addition which C did not have is function/method overloading. Obviously in C, we could sort of recreate the same principle by setting the parameters to void pointers and then casting the pointer to whichever type was necessary, as well as using var args, however, this is not actually the same as function overloading. If the overloaded functions signature differs by either having a different number of parameters or changing at least one of the datatypes of at least one parameter (can be achieved by rearranging the order of datatypes in the parameter list as well) then we can create a separate definition and overload that function.

Another feature that I’d like to mention is that we now have the ability to create function templates. If you recall back when we were discussing vectors, and how they utilize the same principle of generic types in Java in order to have dynamic typing, that is thanks to function templates. Function templates can be used on functions or classes, and allow the user the pass in a data type as a parameter which can then be used to dynamically change the data type of certain members, return types, or parameter types. This works in the same manner as inline methods. Instead of the keyword “inline”, we use “template <typename T>” followed by the function definition. Anytime ‘T’ is referenced, C++ interprets it according to it’s context similar to how dynamic/interpreted languages function. So, If I defined a function this way eg.

template <typename T>

T useless\_function(T some\_value) {

return some\_value;

}

Then the return type would be of type T, and some\_value would also be of type T. In order to call such a function, we would simply do “useless\_function<float>(7.0f);”. We can define as many types as we want by separating each with commas. For example “template <class T, class R, class Y>”. Also, just like functions, templates can have default parameters as well eg. template <class A=int, class B=std::string>. Templates are considered to be so powerful that C++ developers have created a concept called template meta-programming – essentially a way of creating a meta language within C++ using templates, which we will cover later on.

Pointers vs References:

Raw pointers have been carried over from C with the same syntax that you’re familiar with. This includes an asterisk after the datatype and setting the pointer to the address of another variable using the ampersand (&). While we certainly can still use raw pointers in C++, the idiomatic equivallent is to use references. In essence, references accomplish the same goal as pointers, however, there are differences between the two. For references, we use the ampersand, similar to how we would use the asterisk when declaring a pointer e.g. datatype &ref\_name; in order to differentiate the variable as a reference. Outside of variable declaration, the ampersand always refers to a variable’s address. In order to set a reference to the address of another variable, we no longer require the address operation as you would with raw pointers. Instead, we just assign the reference to the value we want it to point to, and the address of the rvalue is retrieved implicitly. For example: int &ptr = num; Since ptr is a reference type, the assignment operator implicitly assigns the address of num to ptr. Another difference between pointers and references is that pointers would need to be dereferenced to alter the value at the pointers address. Once again, dereferencing is done implicitly, so we can access member variables or functions with a reference variable the same way that you would access them normally. One distinct difference between pointers and references is that references cannot be of type void. This was an intentional decision made by Stroustroup to avoid dangerous type-punning operations. Take a look at this example code:

#include <iostream>

#include <vector>

void addFive(int &x)

{

x += 5;

}

int main()

{

std::vector<int> vec { 10, 20, 30, 40 };

for (int i = 0; i < vec.size(); ++i)

addFive(vec[i]);

for (std::vector<int>::iterator i = vec.begin(); i < vec.end(); ++i)

std::cout << "Element " << i - vec.begin() << ": " << \*i << std::endl;

}

Output: 15, 25, 35, 45

Aside from the iterator stuff at the end, which we’ll cover later, this code is fairly self-explanatory. We loop through the vector that we create on the stack and implicitly pass the address of each element in said vector to our addFive function, which accepts a reference to the element and increments it. This change persists precisely because we are modifying the actual underlying data that we pass to the function, as opposed to a local copy of it on the stack.

It is not uncommon within C++ programs to have to use a C library which uses raw pointers. Knowing how to convert between references and raw pointers is a skill which is acquired gradually. Below is some additional code examples to demonstrate the differences between references and raw pointers:

inline void takeRawPtr(int \*x) { \*x += 5; }

inline void takeRawDblPtr(int \*\*x) { \*\*x += 5; }

inline int \*retRawPtr(int \*x) { \*x += 5; return x; }

inline void takeRef(int &x) { x += 5; }

inline void takeRef2RawPtr(int \*&x) { x += 5; }

inline int &retRef(int &x) { x += 5; return x; }

int main()

{

int x = 10;

int \*px = &x;

int &rx = x;

takeRawPtr(px);

takeRawDblPtr(&px);

px = retRawPtr(px);

takeRef(\*px);

takeRef2RawPtr(px);

rx = retRef(\*px);

return EXIT\_SUCCESS;

}

It should be noted that references, unlike raw pointers, must be assigned when they are declared, and cannot be assigned as nullptr or NULL. Once again, this is a safety feature to avoid attempts at dereferencing null pointers, which is a common mistake made in C programs.

You are surely aware by now that const prevents a value from being altered during runtime. In order to save a bit of computational cost, if we happen to have a parameter that we do not want to be altered during the execution of a function, we can replace it with a constant reference and this will save the computer having to make a local copy of the parameter in stack space. For example, instead of having a function func(int const dontAlterMe) {...} we could instead use a reference: func(int const &dontAlterMe) and that would use the address of the argument rather than making dontAlterMe it’s own local variable on the stack.

Namespaces in Detail:

You may find as we write code in C++, that it becomes increasingly tiresome to explicitly state that our strings or vectors are from the standard C++ namespace. We can remedy this to some degree by telling C++ to search a particular namespace when it cannot find the definition of the function in the global namespace. By using the “using” directive, we can specify a namespace for C++ to search if namespace is explicitly provided. You will likely often see people add the line “using namespace std;” This tells C++ to search the standard namespace which means that we no longer have to type std::string, std::vector<datatype T>, std::cout, std::cin and so-forth. You may vaguely remember namespaces in C if you read through my notes, however, we didn’t discuss them much because aside from using the typedef keyword, they weren’t a large part of the syntax. As I briefly went over at some earlier point, namespaces essentially help us with scope. If we have a function foo() in library x and function foo() in library y, C++ needs to know which foo() we are referring to. We can also declare our own namespaces by wrapping our methods in a namespace wrapper. The general syntax is as follows:

namespace namespace\_name {

method() {

//code block

}

}

namespace is a keyword stating that we are declaring a new namespace, namespace\_name is the name that we want to call our namespace, and anything else within our namespace wrapper must now be prepended with namespace\_name eg. mynamespace::method(); This can get somewhat confusing since we can have nested namespaces, where we have namespaces within other namespaces by wrapping them within each other. Here’s an example:

namespace namespace1 {

foo() {

//code block

}

namespace namespace2 {

foo() {

//code block

}

}

}

Now, in order to call foo() from namespace1, we would call it as: namespace1::foo(); or if we wanted to call foo() from namespace2: namespace1::namespace2::foo(); We can use the “using” directive multiple times in our code to tell C++ to look in multiple namespaces, but this can become dangerous. For example, adding both “using namespace namespace1” and “using namespace namespace2” would cause ambiguity once again, which ironically, was supposed to be the goal of separating the 2 definitions of foo() into their own namespaces.

Because of the possibility of reintroducing ambiguity with the using directive, it is somewhat of a controversial topic within the C++ community. I think I’m inclined to agree that it should generally be discouraged in production code/professional code, however, if you know what you are doing then it may be acceptable in personal projects. Just note that when we “using namespace std;” we are sort of ignoring why namespaces exist in the first place so proceed with caution.

The Unexpected Difference Between Classes and Structs:

If you’ve seen any C++ code being written, or have done a bit yourself, you have probably questioned what the difference between a struct and a class is. There is 1 and only 1 difference between a struct and a class in C++, and that is that classes are private by default, and structs are public by default. That is legitimately the only difference, and if you’re coming from C, you may be wondering how that could be the case. After all, structs don’t have constructors right? Well, in C++, a struct can indeed have constructors. So for example, I might have a class that appears like the following:

class MyClass {

std::string color;

int len;

bool setValue;

public:

MyClass(std::string color, int len, bool setValue);

~MyClass();

};

Notice how by default, the instance variables are private, and therefor outside the scope of other classes, hence why we need to explicitly state that the constructor/destructor are public methods. But if we simply take the exact same code and replace ‘class’ with ‘struct’:

struct MyClass {

std::string color;

int len;

bool setValue;

MyClass(std::string color, int len, bool setValue);

~MyClass();

};

Now it is no longer necessary to make the methods public, although without a private modifier, the instance variables are also public now. I will include another code snippet down below to demonstrate an actual implementation of this struct in a program:

#include <iostream>

struct MyClass {

*std*::*string* color;

int len;

bool setValue;

MyClass(*std*::*string* color, int len, bool setValue);

~MyClass();

};

MyClass::MyClass(*std*::*string* color, int len, bool setValue)

{

this->color = color;

this->len = len;

this->setValue = setValue;

}

MyClass::~MyClass() {

}

int main(void) {

struct MyClass myclass1("red", 5, true);

*std*::*cout* << myclass1.color << *std*::*endl*;

*std*::*cin*;

}

Here you can see that I’m creating an instance of a struct, not a class, and because my constructor is public by default, a can print out the myclass1’s color value. If I were to change struct to class however, then I get a compile time error “cannot access private member declared in class ‘MyClass’”.

So that then begs the question... If struct and class are essentially identical, why do we even need both? The simple answer to this is to maintain backwards compatibility with C, similar to why C++ keeps pointers. You could go your entire C++ career without ever creating a struct. However, it is up to you if or how you implement structs in your code, and this might be an advantage to you. For instance, I tend to use structs when I’m creating a block of data that is related, but doesn’t necesarilly constitute as a real life object. I’d use a class for more complex logic, if I want to be inheriting from other classes, or creating copy constructors, or creating member functions, etc. You do not have to do it my way, but I figured it’d be useful to share how you *could* use structs if you so choose to.

Inheritance and Polymorphism:

One of the most confusing aspects about C++ is in how it deals with OOP. You see, C++ was really one of the first languages to use the OOP paradigm, and as such, many oddities resulted. Let’s discuss a few of these oddities. The first oddity, which was intentially designed by Stroustrup, is that there is no universal *root class*. In Java, and most other modern OOP languages, there is a root class (in the case of Java, this would be the Object class), from which all classes implicitly derive from. Because there is no root class in C++, there is a much heavier reliance on templates and multiple inheritance. It also leads to other strange behaviour, such as type casting errors when trying to cast one class to another. In Java, if you try casting Foo to Bar when Bar does not inherit Foo, an exception will be thrown. In C++, this behavior is undefined. Multiple inheritance is perhaps the greatest sin commited by C++. In Java, a class may only extend one base class, but in C++, a class may derive from multiple classes. This leads to all sorts of confusion and disarray. A classic absurdity that arises from multiple inheritance is the infamous diamond problem. This occurs when one class inherits from 2 base classes, which themselves share the same base class. This relationship creates the shape of a diamond if layed out in a heirarchical view. The issue is that child classes automatically inherit all properties of their parents. This includes both member variables and member functions. If class A exists at the top of the diamond, and has a public member function named foo(), then its children B and C necessarily contain foo() as well. You may study this example for reference:

#include <iostream>

using namespace std;

class A

{

public:

void foo()

{

cout << "foo" << endl;

}

};

class B : public A

{};

class C : public A

{};

class D : public B, public C

{};

int main()

{

D d;

d.foo(); // Compile-time error: Member ‘foo’ found in multiple base-class subobjects of type ‘A’

return 0;

}

This can be resolved by either casting d to be of type B or C: ((C)d).foo(), using the scope resolution operator; or by using something known as virtual inheritance, which we will discuss in a bit. In general, it is recommended that you avoid multiple-inheritance.

I don’t want to gloss over what I said earlier about child classes inheriting all member variables and member functions from their parents, as I believe that this is crucial to your understanding of how inheritance works in C++. Since a derived class always inherits *all* properties of its parent, it necessarily contains at the very least, those same properties, plus, optionally, any additional properties which it declares. Very important to note, however, is that just because it inherits all properties of its parent, this does not mean that it inherits all *access* to those properties. For instance, a child class may not access private members which belong to its parent. Additionally, a child class cannot initialize members of the base class using an initializer list in its constructor.

Virtual Functions / Virtual Inheritance:

I mentioned that we would discuss using virtual inheritance as a means of bypassing the diamond problem, so that’s what we will now discuss. Basically, you can think of virtual inheritance as being the same as the @Override annotation in Java. Virtual inheritance does have subtle differences though, but the end result is the same. In order to understand how virtual inheritance works, you must understand something known as the V-table (virtual table). When a member function is declared virtual, the class which declared the virtual function immediately receives a pointer to a dyamically allocated v-table. The v-table is just a table of function pointers, which contains a pointer to each implementation of the virtual function. For instance, if class Foo has a virtual member function named foo(), it will recieve a pointer to a v-table that contains all current implementations of foo(). If a class named Bar extends Foo, since child classes automatically inherit all properties of its parent, Bar too, will inherit the same v-table. Then if Bar overloads the virtual function foo(), the v-table will now contain two function pointers: one to the version of foo() belonging to class Foo, and the other version of foo(), belonging to class Bar. The v-table will continue to dynamically grow in size as more and more classes overload the function foo().

So, what are the rules for virtual functions in C++? Basically, the base class must place the virtual keyword in front of any member function which it wishes to allow its children to overload. The child classes do *not* necessarily need to put the virtual keyword in front of the function overload, however, I strongly implore you to consider adding it, so that readers of your code understand that it is being overloaded! But technically speaking, at the bare minimum, the base class is the only class which must place the virtual keyword in front of the function.

Pure Virtual Functions:

In C++, there is no such thing as the *implements* keyword. You may be asking yourself then, “how do I implement interfaces in C++”? Interfaces as we’re used to them do not exist in C++, and therefore, cannot be implemented technically speaking. We can, however, use a class filled with functions known as *pure virtual functions*. This essentially turns the class into an abstract class, meaning that it cannot be instantiated. Other classes may extend this “abstract” class (remember, abstract classes don’t really exist in C++, but for all intents and purposes, classes which have pure virtual functions are abstract) and must then create a local definition for the pure virtual function, similar to an interface. So how do we mark a function as being “pure” virtual rather than just virtual? Well, the syntax is a bit odd, but we essentially set the function equal to 0 like so:

class Shape

{

public:

// Pure virtual function

virtual void calculateArea() = 0;

};

Now anything that derives from shape must implement its own definition for calculateArea(), otherwise a compile-time error will be thrown. Note that pure virtual function inheritance is very similar to virtual inheritance, with the key difference being that the parent class gets to have its own definition for a member function with normal virtual inheritance (as do its children), but with pure virtual inheritance, only the child classes get to have their own definitions.

Virtual vs Override vs Final:

C++ has both a virtual keyword and an override identifier. This can be confusing because new C++ programmers aren’t sure when it is necessary to use virtual or when to use override. The answer is that it really comes down to preference. The virtual keyword must be present before a member function declaration in the base class for it (and by extension, further overrides to it in descendent classes) to be placed in a vtable. Children of the base class may optionally include the virtual keyword, although it is not a requirement for the compiler. Similarly, the override identifier is entirely optional. If placed after a function declaration, the override keyword ensures that the function’s signature matches that of the base class’. The override identifier may only be used if the virtual keyword is also explicitly used. We can use the final identifier rather than, or in tandem with the override identifier, to suggest that any classes which inherrit from the current class may not override the member function, effectively preventing further additions to the vtable. Here is an example:  
  
class A {

virtual void foo() {

...

}

};

class B : public A {

virtual void foo() override {

...

}

};

class C : public B {

virtual void foo() final {

...

}

}

class D : public C {

virtual void foo() { // Error: Declaration of ‘foo’ overrides a ‘final’ function

...

}

};

Member Variable Initialization Via Constructor:

There are 3 primary methods of initializing member variables via constructor in C++. The most basic method is to take in an argument for each member variable in the class that needs to get set when the constructor is invoked. The parameter’s name must differ from that of the member variable, but must also be of the same type. For example, assume our class Foo has an int, a float, and a std::string:

class Foo {

private:

int m\_integer;

float m\_decimal;

std::string m\_sentence;

public:

Foo(int integer, float decimal, std::string sentence);

}

Now we can define the constructor as such:

Foo::Foo(int integer, float decimal, std::string sentence)

{

m\_integer = integer;

m\_decimal = decimal;

m\_sentence = sentence;

}

Of course, there is another way of doing the same thing using the “this” keyword, which is a compiler intrinsic that expands to a pointer of the enclosing class. Rather than using different names for the member variables and the constructor arguments, we can instead access member variables with “this”, which removes ambiguity between the two variables. Pretend we have the same class Foo, but this time I’ve stripped away the m\_ prefix from the member variables. Here is the new constructor definition:

Foo::Foo(int integer, float decimal, std::string sentence)

{

this->integer = integer;

this->decimal = decimal;

this->sentence = sentence;

}

Between the two methods presented so far, which you use is entirely a matter of preference. There is, however, a third alternative for member initialization, which has a practical benefit over the former candidates. The third method to which I am referring are initializer lists. Initializer lists are unique due to the fact that they implicitly prevent double initialization. In our previous examples, the sentence member variable would be initialized once during the class declaration as an empty string, and then again during the assignment within the constructor. Assignment via initializer list takes precedence over any default initialization within the class declaration, which is a very miniscule optimization. On top of this though, initializer lists are, in my opinion, a bit more legible, since they separate assignment operations from any other logical operations which need to take place within the constructor. The initializer list goes in between the closing bracket of the constructor’s argument list and the opening curly brace of the constructor’s code block. Here’s what this looks like:

Foo::Foo(int integer, float decimal, std::string sentence) :

m\_integer(integer), m\_decimal(decimal), m\_sentence(sentence)

{

// Any additional logic can go here

}

Initializer lists are also the method used to chain constructors. In C++ chaining constructors is known as “constructor delegation”. This was introduced in C++11 so previous versions will not support constructor delegation. Take the following example code:

class DelegationExample {

private:

int number;

public:

DelegationExample();

DelegationExample(int n);

void setNumber(int n);

};

DelegationExample::DelegationExample() : DelegationExample(5) {}

DelegationExample::DelegationExample(int n) {

setNumber(n);

}

void DelegationExample::setNumber(int n) {

number = n;

}

int main() {

DelegationExample delInstance = new DelegationExample();

return 0;

}

In this example, when delInstance gets created, the default constructor is called. Because DelegationExample is in our initializer list, however, before the code within the default constructor is called (not that there is any in this example), the constructor with the int n parameter is called, which calls setNumber and sets number = 5. The call stack will then begin to return and the stack pointer will go back to run any code within the default constructor since it didn’t get a chance to once the initalizer list took over. It should be noted that initializer lists can also work for objects but there can be a lot of confusion about that. The following code is from stack overflow and demonstrates the different ways in which you could create an object with an initializer list and default constructor and the outcome of each method:

class NewFoo

{

int x;

int y;

};

// Version 1:

class Bar1

{

NewFoo f;

};

// Version 2:

class Bar2

{

NewFoo f;

public:

Bar2() {} // f not in list.

};

// Version 3:

class Bar3

{

NewFoo f;

public:

Bar3() : f() {}

};

int main()

{

Bar1 b1a; // x and y not initialized.

Bar1 b1b = Bar1(); // x and y zero initialized.

Bar2 b2a; // x and y not initialized.

Bar2 b2b = Bar2(); // x and y not initialized.

Bar3 b3a; // x and y zero initialized.

Bar3 b3b = Bar3(); // x and y zero initialized.

}

<https://stackoverflow.com/questions/13238234/initializer-list-for-objects-with-default-constructor>

Special Member Functions:

C++ generates “special member functions” for us when we create a new class. As we’ve seen, the default constructor is one such member function which is automatically generated for us if we do not create it explicitly. This makes sense, as without any constructor, we would not be able to instantiate the object (which we presumably want to do). C++ auto-generates other member functions, namely, a copy constructor, copy-assignment operator, move-assignment operator, destructor, and prospective destructor. Here is a summary for each:

**Default constructor:** A no-parameter constructor which allows an object to be instantiated.

**Copy constructor:** A constructor which takes as a parameter a reference to the outer class with the intention of instantiating a new instance of the class by copying the attributes of the instance passed into the constructor, effectively making a copy of said object.

**Copy-assignment operator:** Similar to the copy constructor, but overloads the assignment operator (=). The right operand, or rvalue, of the assignment is passed into the assignment operator function overload as a reference and the left operand, or lvalue, inherits all properties of the right operand.

**Move-assignment operator:** Works similar to the copy-assignment operator, but transfers ownership of the reference to the rvalue to the lvalue, thus invalidating the rvalue. The rvalue remains valid, but goes into an indeterminate state, unlike a language like Rust, where it is no longer viable for use.

**Destructor:** A function invoked when the delete keyword is explicitly used to cleanup the object.

**Prospective destructor:** A class may have multiple prospective destructors i.e. potential destructors, but only one actual destructor. This is useful in the case of, for example, templated classes, in which you may want to have the compiler infer one of multiple destructors depending upon a generic type or other conditional.

Deleting Special Member Functions:

As of C++ 11, we have the ability to delete member functions, including special functions such as the ones listed in the previous section. We can do this using the delete keyword. We can also specify which constructor the compiler ought to treat as the default constructor with the default keyword. Here is an example:

class Foo

{

Foo();

Foo(const Foo&) = delete;

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

};

In this example we delete the auto-generated copy constructor, as well as the auto-generated copy operator, making it so that we cannot make copies of any instances of Foo. Here’s an example:

int main(void)

{

Foo one = Foo(); // Okay

Foo two = Foo(); // Okay

Foo three(one); // Call to deleted constructor of ‘Foo’

two = one; // Overload resolution selected deleted operator ‘=’

}

The delete keyword can be used on non-auto-generated member functions as well. One reason you might use this is to avoid unwanted type promotion. For example, we can prevent type promotion from float to double from succeeding like so:

void callWithTrueDoubleOnly(float) = delete;

void callWithTrueDoubleOnly(double param) { return; }

In the example above, we prevent the call to callWithTrueDoubleOnly() from succeeding if the argument is a float (normally it would be promoted to a double to work with the function). Note, however, that an int will still work in this case. Rather than create another delete function for int, we can use templates to make this process cleaner:

template<typename T>

void callWithTrueDoubleOnly(T) = delete;

void callWithTrueDoubleOnly(double param) { return; }

Another reason for deleting member functions is to remove them in child classes that inherit from a base class.

Defaulting Special Member Functions:

The default keyword has a very niche use-case. When the C++ compiler generates one of the auto-generated member functions, it will also generate definitions for these member functions. The default keyword tells the compiler to use those definitions rather than expecting the user to implement one themselves.

Move Semantics:

C++ has a concept known as move semantics, which can (in my opinion) really confuse a lot of programmers who are new to the language. It will be helpful if you’re familiar with ownership and the borrow checker in Rust, though you should still be able to grasp the concept if not. The std::move() function allows us to transfer ownership of a value from one entity to another. Sometimes you’ll hear talk of move constructors or move assignment operators. These typically just use std::move() under the hood. In some cases, C++ uses move semantics implicitly, such as when returning objects from a function.

Entity Qualifiers:

static, const, and auto make a return in C++, however, with some slight modifications in comparison to C. As there are subtle differences with C++, it would be good for you to review them.

static: This keyword has not really changed since C. The static keyword still indicates to the compiler that the function or variable which proceeds will only be visible within the current translation unit. Of course, this means that we can still get away with things like defining the same function name twice, although this is still not recommended, as there are much better ways in C++ to do this (e.g., function overloading or namespaces). When a member function is declared static within a struct or class, it essentially makes the function no longer a member function of that class. In fact, a member function declared with the static keyword works exactly the same whether it is declared inside or outside of the class body. This is because all instances of a class have access to the same static member i.e., it is not reproduced for each instance of the class. The advantage to declaring a static member within a class is that we can access it using the class’ namespace e.g., Foo::staticMember; which can increase readability.

const: const, as you probably know, discourages modifications to a variable’s memory or value. There happen to be ways to get around this, however, using pointers. Because it is fairly easy to change a const value, it is typically considered more of a promise, rather than an absolutely sure-fire way of protecting the data from being modified. The const keyword takes on a different meaning when qualifying a member function. As you can see in the code snippet below, placing const after the function signature prevents modifiication of a class’ member variables. This only applies to member functions and not static or global functions, for obvious reasons.   
  
Class Foo {

private:

int i;

public:

void mutate\_i() const {

this->i = 4; /\* Cannot assign to non-static data member within const member function ‘mutate\_i’ \*/

}

};

int main() {

Foo f;

f.mutate\_i();

}

auto: In C, auto was the implicit default for all stack-allocated variables. There was essentially never any reason to actually use it. I suppose this is why C++ decided to repurpose it for something entirely different. In C++, auto now tells the compiler to infer the type of a variable. Note that this is still type-safe since the type is inferred from the rvalue of the expression. Once inferred, its type cannot be dynamically altered unlike an interpreted language such as JavaScript or Python. For example, we cannot do:

auto a;

a = 5;

Rather, we must do auto a = 5;. a’s type will be determined by the compiler by looking at the rvalue (in this case, 5), which it knows is an int, and therefore, will make a of type int.

constexpr:

The C++ keyword constexpr, pronounced “const expression” and introduced in C++ 11, refers to an expression which is computed at compile time and is immutable. The constexpr keyword can be applied to variables, functions, and classes. For example, an arithmetic operation such as float z = exp(5, 3); would normally be computed at runtime, however, placing constexpr before the statement would cause the result to be computed at compile time. Note that expressions marked with consexpr are only valid if all function calls and variables are themselves marked with contexpr (or const in the case of variables). For example, the following is an error:

int j = 0;

constexpr int k = j + 1; // Fails since j was not qualified with constexpr or const

Likewise, the following is also an error:

#include <cmath>

#include <complex>

constexpr std::complex<double> eulers\_ident = std::exp(std::complex<double>(0.0, M\_PI)) + 1.0;

The latter fails because std::exp() is not marked as constexpr.

C++ Casts vs C-Style Casts:

Casting in C++ is actually quite the conundrum, believe it or not. To maintain backwards compatibility with C, C++ implements C-style casts (using brackets), but the implementation of C-style casts function quite differently than they do in C (more on that later). The typical way of casting in C was simply to use the round brackets with the target datatype in between to indicate that a value should be cast to that type before variable assignment. This would look something like the following: int asciiToDec = (int)’a’; Here the character ‘a’ would be cast to int, and become 97 before being assigned to variable asciiToDec. However, I also mentioned that C++ was different in that casting a void pointer must be explicit. C++ has 4 standard types of cast that must be declared explicitly. Note that C++ has a really odd feature that C didn’t have whereby we can swap the bracket position for the variable and the type being cast to in a C-style cast. Here is an example:  
  
char a = ‘a’;

// Regular method:

// int b = (int)a;

// Alternative method:

int b = int(a);

I say this now so that hopefully the syntax for C++ casts makes more sense as we look at them. Without further ado, we’ll begin with the first type of C++ cast: const\_cast.

- **const\_cast <type> (expr):** A const\_cast will actually alter whether or not a variable is read-only. Ex:

int main() {

const int a = 9;

const\_case<int&>(a) = 4; // This is a valid reassignment!

}

- **static\_cast <type> (expr):** The static\_cast performs a non-polymorphic cast. For example, it can cast a base class pointer to a derived class pointer. Unlike dynamic cast, static cast occurs at compile time. In C++, we do not need to explicitly use static cast to downcast from a base pointer to another class type which inherits from the base class, however, it is advised due to it being idiomatic. By using static cast, we differentiate between const, dynamic, and reinterpret casts, unlike C-style casts, which have no visual method of differentiating between each type. Ex:

class A {};

class B : public A {};

int main() {

A a;

B \*b = static\_cast<B\*>(&a);

return 0;

}

- **reinterpret\_cast <type> (expr**): The reinterpret\_cast is essentially the same as static cast, but whereas static cast has some compile time checks to ensure that the conversions of types make intuitive sense (e.g. casting a double to an int fails), reinterpret cast has no cares about bit twidling to fit a data type that does not make sense for the actual underlying data. This is essentially equivallent to storing some data of one type in a void pointer in C, and then reinterpreting that data as another type which may or may not make sense. This practice is known as type punning.

Ex:

#include <iostream>

struct A {

int m = 5;

};

struct B {

int m = 10;

};

int main() {

A a;

B \*b = reinterpret\_cast<B\*>(&a);

std::cout << b->m << std::endl; // Prints 5

return 0;

}

- **dynamic\_cast <type> (expr):** The dynamic cast performs a polymorphic cast. They operate at runtime and will return nullptr if the cast cannot succeed. The primary use of this is to downcast a pointer or reference to a base class into another class type which inherits from the base class. This is only possible if the base class has at least one virtual function. Ex:

class A {

public:

int i;

A() : i(7) {}

virtual void foo() {}

};

class B : public A {

public:

int i;

B() : i(3) {}

virtual void foo() override {}

};

int main() {

B \*b = new B();

auto b2a = dynamic\_cast<A\*>(b);

if (b2a) {

cout << b2a->i << endl; // Prints 7

}

}

**How C-Style Casts Work in C++**

As mentioned earlier, C-style casts have a different implementation within C++, which is obfuscated from you, the programmer, and which makes it rather dangerous to use. When performing a C-style cast, it will attempt the following C++ casts under the hood, in order, until one succeeds:

1. const\_cast

2. static\_cast

3. static\_cast\*, then const\_cast

4. reinterpret\_cast

5. reinterpret\_cast, then const\_cast

This is pretty dangerous right of the bat, because lets say that you have a const variable, and then you perform a C-style cast to change its type, however you forget to add the const qualifier in front of the type you’re casting to. The compiler will attempt the first C++ cast, being const\_cast. Since we’re changing the type of the variable, the compiler will not accept this and continue to step 2. Static cast will also fail, due to the fact that we accidentally failed to add the const qualifer to the type being cast to. The compiler will then try 3, which will succeed, since it will change the variables type and remove the const qualifier silently, without warning us! This is even scarier if we perform a C-style cast from one type to another when those two types have no relationship with one another. A static\_cast will at least catch this and throw an error, but a C-style cast will just end up doing a reinterpret\_cast whether you like it or not. This is why it is so heavily discouraged to use C-style casts in C++, and why its best to stick to the primary C++ casts even though they can be mildly more annoying to type out.

decltype:

An easy way to return the type of a variable is through use of the decltype specifier. For example the assert statement in the example below will succeed:

int a;

assert((std::is\_same\_v<decltype(a), int>));

Operator Overloading:

Another feature that I am actually quite fond of in C++ (and I think many other people like as well), is operator overloading. This is pretty unique to C++ (though I’m sure other languages have this ability), a sort of C++ staple if you will. Have you ever wished to be able to check if 2 objects are equivellant using ==, or perhaps wanted to increment every variable in an array using ++? This is possible in C++ because we can overload the definitions of operators and create our own definitions. In C++, operators are treated as functions, where the statements to the left and right of the operator are parameters. For example, let’s say that we wanted to set an object of a parent class A, equal to an instance of the derived class, B (A a = B;). This is possible in C++ even without operator overloading, but my point is that the argument on the left of the = sign is treated as a parameter, as is B on the right. If = was the name of the function, this would sort of be like calling “=(a, B);” although this is not completely accurate. Anyhoo, parameters are implicitly passed for unary operators such as ++, --, &, etc. And are explicitely passed for binary operators such as <<, >>, ==, =, +, -, /, \*, %, etc. So how do we actually overload an operator. Well, let me preface by saying that operator overloading can only be performed on classes. Within a class, we can declare a function using the “operator” keyword followed by the operator we want to overload.

Let’s first say that we have a class called OpOverload which contains a member variable called “number”. We want to be able to use the pre and post increment operators on instances of this class. For example, we want to be able to do the following:

OpOverload op;

++op; // pre increment

op++; // post increment

where the pre and post increment operators will increment the “number” associated with the instance of the class. I will define the class, and the definitions for pre and post increment and explain afterwards:

#include <iostream>

class OpOverload {

private:

int number;

public:

OpOverload() {}

~OpOverload() {}

// Pre-increment

OpOverload& operator++() {

this->number++;

return \*this;

}

// Post-increment

OpOverload& operator++(int) {

OpOverload& temp = \*this;

temp.number = this->number;

this->number++;

return temp;

}

};

int main() {

OpOverload op;

++op;

op++;

}

Alright, I’m sure this is quite confusing, which is to be expected because the syntax isn’t exactly pretty. Let’s look at pre-increment first. Recall that when we say ++op; we are sort of calling the ++ function with op as a parameter. Because the pre-increment operator is unary, op is passed implicitely. Notice that we also return a reference of the class. This is because whichever object we are pre-incrementing will essentially get set equal to the return value of the operator function. In other words, our instance, op, will get set = to the reference which is returned from our operator overload. So, we call ++(); then, we acces the member variable “number” which belongs to “this”, which is the implicit argument that we passed, op. But hold on, why are we using post-increment and not pre-increment on number? Well in this case it doesn’t matter if we increment the number using pre or post increment because all that matters is what gets returned. Either way, number gets incremented before we return the reference to OpOverload, so no harm is done. As for the return value, we must dereference “this” before returning it since it is a pointer and we need to convert it to a reference. I hope you followed that, but I understand that it’s confusing.

As for the post increment, well this is perhaps even more confusing. With post increment, we are also calling ++(); but then that begs the question: what differentiates the pre and post increment overloads? How does C++ know which you are overloading? Well this is why we placed that ominous int parameter: in order for C++ to recognize it as a post increment operator. You couldv’e given this parameter a name like int i, however, since it is useless and never gets used except to check that this is a post increment overload, I did not include the variable name. Moving onto the code, I create a temporary reference called temp which gets set equal to the implit parameter, op (once again, dereference to convert pointer to reference). We set the member variable “number” of the temporary reference = to the current value of “number” in “this”. Then, we increment this->number. Notice though that we return temp, and not this. Recall that I said only what we return matters. Now op will be set to the reference which was returned, which was temp, so it’s member, “number” will remain the same, until we access number on the next line, where it will be incremented since we said this->number++; It’s all very confusing I know. This stuff simply comes with practice.

Now let’s say that we want to create a sort of “toString()” function like we have in Java, and overload the redirection operator << to print the member variable “number” of the object like so std::cout << objectInstance << std::endl; Here is the code for that:

#include <iostream>

class OpOverload {

private:

int number;

public:

OpOverload() {}

~OpOverload() {}

//Pre-increment

OpOverload& operator++() {

this->number++;

return \*this;

}

//Post-increment

OpOverload& operator++(int) {

OpOverload& temp = \*this;

temp.number = this->number;

this->number++;

return temp;

}

friend std::ostream& operator<<(std::ostream& os, const OpOverload& op) {

os << "Object’s number variable = " << op.number;

return os;

}

};

int main() {

OpOverload op;

++op;

op++;

std::cout << op << std::endl;

}

Output: Object’s number variable = 2

Notice the use of a new keyword for this one: friend. friend allows external functions or objects to access our classes private and public members. Since ostream must access op.number, we must declare this function as a friend function, essentially granting ostream permission to access our private/public members in OpOverload. ostream is simply short for output stream and must be returned when overloading the output redirection operator. The const keyword is used for our reference of OpOverload since it’s data is not modified in this function.

Here is another slightly more complex example of operator overloading where we overload the post increment operator to act upon each member of a collection simultaneously (in this case std::Vector):

#include <vector>

#include <iostream>

template <class T>

class Vec : public std::vector<T> {

public:

std::vector<T> m\_vec;

Vec(std::initializer\_list<T> l) : m\_vec(l) {}

~Vec() {}

void operator++(int) {

for (auto& i : this->m\_vec) {

i++;

}

}

};

int main() {

Vec<float> vec{1.0f, 1.0f, 1.0f};

vec++;

for (auto i : vec.m\_vec)

std::cout << i << " ";

std::cout << std::endl;

return 0;

}

So, let me begin to explain myself here. Since we cannot access the vector class directly, the way in which we overload it is to extend it. This is why I create a new class called Vec : public std::vector<T>. You’ll notice that the class has been templated, i.e. it accepts any data type from the user when we create an instance of it, and anywhere where T arises, we can replace it with the data type that was passed. Within the constructor, we take in one parameter, std::initializer\_list<T> l. Initializer list is an object which accepts a list of variables enclosed in {} (an initializer list). This way, when we create an instance of the class, we can pass in a list of elements that we want to add to the vector. The class contains a member variable, m\_vec, which is the vector that we are actually modifying. The constructor also initializes m\_vec with the initializer list that we pass in. Finally, we get to the overload of the post increment operator. This time we don’t need to return anything since the instance of the object is not being modified, only m\_vec is. An advanced for loop says that for every element in this->m\_vec (remember that this is the implicit lefthand parameter; in our case vec), we want to create a reference called i (a reference to the current next element in m\_vec). The auto keyword is common in advanced for loops, and in this case makes our life easier since the data type could vary. Alternatively, we could’ve said for(T& i : this->m\_vec). Anyways, we simply increment each element with i++;

And that’s pretty much it. The second for loop simply accesses m\_vec and prints every element to standard out. Notice that this even works for characters, so if we created the vector Vec<char> vec{‘a’, ‘b’, ‘c’};

vec++;

Output: b c d //Pretty neat!

Enums:

Enums are nearly identical to their C counterparts except for two primary benefits. C++ will treat enums as unsigned ints by default. One benefit is that we can specify which integer type we want them to abide by. We do this by adding a colon after the enum’s name followed by an integer type that we want its members to be. For example:

enum Color : uint32\_t

{

RED = 0xFF0000FF,

GREEN = 0x00FF00FF,

BLUE = 0x0000FFFF

};

The second benefit of enums are enum classes. An enum class is likely not what you’d think it would be. It simply indicates that the enum will no longer be treated as an integer type, meaning that comparisons between the enum members and any numeric type will fail. Only comparisons between two enums of the same type are valid. We are also now forced to prepend the enum’s members with its namespace.

enum class DrawCmd

{

REDRAW,

FILL,

CLEAR

};

...

DrawCmd cmd = DrawCmd::REDRAW; // *Must* use namespace for REDRAW

if (DrawCmd::REDRAW == 0) // Invalid operands to binary expression (DrawCmd and int)

{

...

}

Smart Pointers:

For the most part, the C++ community has strayed away from raw pointers in favor of something called smart pointers in C++. Introduced in C++, smart pointers act in similar fashion to a Garbage Collector. The difference between the two is that a GC is a separate program which monitors the heap an performs deallocation when it notices that the memory can no longer be accessed, whereas smart pointers are individual instances which act as wrappers for raw pointers and then perform deallocation on their owned resource once they go out of scope. This works because the smart pointer’s destructor contains the call to delete, which then invokes delete on its owned resource i.e., the raw pointer. Smart pointers are very useful, as they alleviate the issue of memory leaks, which occur when we forget to clean up a resource after it’s done being used.

There are 3 primary types of smart pointer in C++:

**- unique\_ptr:** Unique pointer is kind of like specifying the restrict keyword in C. It specifies that the underlying raw pointer may only have one owner, which is the unique pointer that encapsulates it. A runtime exception will be thrown if another pointer attempts access at the underlying raw pointer owned by a unique\_ptr. Because shared pointers typically ought to be avoided, unique\_ptr is a good default choice for POCOs.

**- shared\_ptr:** A shared pointer allows for multiple owners of a single resource. It maintains a reference count and does not delete the underlying raw pointer until all owners have gone out of scope or have given up ownership (i.e. ref count becomes 0). Shared pointers actually occupy double the amount of bytes as a normal pointer, since the shared pointer object must contain a reference to the resource being pointed to as well as a reference to the number of shared pointers that currently hold ownership over the same resource. To showcase this, we can run the following code:

#include <iostream>

#include <memory>

using namespace std;

class A {};

int main()

{

unique\_ptr<A> uniq(new A);

cout << sizeof(uniq) << endl;

shared\_ptr<A> shared(new A);

cout << sizeof(shared) << endl;

return 0;

}

Output:

8

16

**- weak\_ptr:** Finally, weak pointer is a special-case pointer which is used in tandem with shared\_ptr. Weak pointers provide access to an object which is owned by one or more shared\_ptr instances, but does not participate in incrementing the reference count. This is useful when you want to observe an object, but don’t care if it remains alive or not. The most practical application for weak pointers is to avoid a problem known as reference cycles. An issue arises when two or more shared pointers create a cyclic loop by pointing towards each other. For example, if we wanted to create a circular linked list where node A points to B, which points to C, which then points back to A. This will pose an issue with shared pointers specifically, because so long as the reference count is not 0, they will not be destroyed (even after main() exits)! This is a memory leak, and the way that it can be resolved is by making one of those shared pointers a weak pointer. Because weak pointers don’t maintain a reference count of their own, they will be destroyed when they go out of scope.

Here is a program that produces a reference cycle:

STL Containers:

Iterators:

Iterators are a behavioral design pattern, which are becoming more and more common in modern programming. Iterators do as they suggest – they iterate over a collection or container in a linear (O(n)) way. You may ask yourself, why do we use iterators if we can just use a for loop? Well the argument typically goes as follows: For loops assume that the container uses indices when it may, in fact, not use them. For instance, if we iterate over a linked list using a for loop and stopping when i == ll.size, this works fine, but the end user may be inclined to assume that the ith element of the linked list can be accessed using the index operator, as suggested by the variable i e.g. ll[i]. As we know, linked lists cannot be accessed in this manner. An iterator, on the other hand, is agnostic about which type of collection it is iterating over. That being said, although I’ve heard the argument that iterators should be used whenever possible for consistency, I do not personally believe that they ought to replace for loops entirely. If using an interpreted language such as JS, for loops can provide a big performance boost. For compiled languages, your milleage may vary.

In C++, iterators can be implemented by including the iterator header file. Each STL container has its own namespace, which we can use to access that type’s implementation of iterator e.g., *vector<int>::iterator* or *list<string>::iterator.* Notice that this gives us a consistent interface for declaring iterators, which abstracts away the implementation details of iterator for each type of container. Once we have an iterator, we set it equal to the element that we want to begin iterating at. Since this is usually the first element, we can use the begin() function which is implemented by each STL container to access its first element. We can then check if the iterator is smaller than container.end(), and if so increment the iterator. Here is some example code:

#include <iostream>

#include <iterator>

#include <vector>

using namespace std;

int main()

{

vector<int> ar = { 1, 2, 3, 4, 5 };

vector<int>::iterator ptr;

cout << “The vector elements are: “;

for (ptr = ar.begin(); ptr < ar.end(); ptr++)

cout << \*ptr << “ “;

return 0;

}

Note that the iterator must necessarily override the default pre and post increment/decrement operators. For containers which are allocated in a contiguous memory region, the iterator may simply take advantage of pointer arithmetic and jump forward by the size of it’s type and then check the address that the iterator is pointing to to see if it matches with the final element’s address. For containers which may allocate memory anywhere on the heap, such as a linked list, it would naturally need to override the increment/decrement operators so that they point the iterator towards the next or previous element in the list. If what I just said doesn’t make sense, that’s perfectly fine. Treating iterators as a black box until you gain a better grasp on pointers and data structures is not such a bad thing.

File I/O:

Arguably one of the more convoluted aspects of C++ is how it handles file I/O. If you’re primarily used to C, as I am, you know that there are really only two ways of creating a file, which is either to use the FILE struct and fopen to return a pointer to a file stream, or to use the open() system call to return a file descriptor. In C++, akin to Java, we have much more options when it comes to handling I/O. You are already familiar with the redirection operator (<< or >>), as it is used when we print statements to stdout using cout. Similarly, cin reads from stdin. As you are probably aware, these are both file streams, and their C++ implementation is contained within iostream. In order to handle file streams, we need to also include the fstream header file. This provides access to an object called ifstream. Well, technically speaking, ifstream is a typedef for basic\_ifstream<char>. If we provide a filepath as an argument to the ifstream’s constructor, the constructor will implicitly invoke basic\_ifstream’s implementation of open(). Alternatively, we can simply instantiate the ifstream and then call its open() member function later, when we are ready to actually open the file. The open() function signature is as follows: void open(const char \*filename, ios\_base::openmode mode = ios\_base::in);. Note the default argument “mode”, which is set to ios\_base::in if we do not set it manually. This parameter is actually an object of the bitmask member type openmode, which contains the constants in, out, binary, ate, app, and trunc. The flag bits can be OR’d together e.g., ios\_base::in | ios\_base::binary if we wanted to read the file as a binary file instead of text. Rather than checking if the call to open() returned nullptr or something like that (as we would do in C), we instead call the is\_open() member function, which will return a boolean indicating whether or not the file is currently open. At this point, there are multiple functions that we can invoke to read the contents of said file before eventually calling close() (note that the destructor for ifstream calls close() implicitly).

The easiest way to read the contents of a file as a string is to use the redirection operator on the ifstream object. For example:

#include <iostream>

#include <fstream>

int main()

{

std::string contents;

std::ifstream fs(“todo.txt”);

if (fs.is\_open())

{

while (fs.good())

{

fs >> contents;

}

}

fs.close();

std::cout << contents;

return 0;

}

Note the use of the good() member function. The redirection operator will read text until it encounters a white space character or newline character, meaning that if we have multiple lines we need to use a while loop in order to read each one. The good() member function essentially checks to see if we’ve reached the EOF, an error occurred, or if there was a R/W error. There is also the bad() member function, which only checks if a R/W error occurred. Alternatively, we could have just done while (fs), which is equivallent to while (fs.good())

Similar to ftell() in C, we can use the tellg() member function (stands for tell get) to return the current position of the file pointer as it moves through the stream. Once the entire file has been traversed, this function returns -1.

Another way we can read files is by using the getline() member function. This is pretty much just a wrapper around the redirection operator which returns the line that it read, not including the newline character.

I’ve also seen people use iterators to read an entire file as a string, which I have no opinion on in terms efficiency or readability or any of that. None-the-less, it’s still good to be able to know that there are other ways of doing the same operation. Here is an example of this:

std::string contents((std::istreambuf\_iterator<char>(ifs)),

(std::istreambuf\_iterator<char>()));

What’s happening here is likely not obvious to new C++ programmers. Essentially, std::string has a constructor which accepts two parameters, both of which are iterators. Here is the definition for said constructor: template <class InputIterator> string (InputIterator first, InputIterator last); This constructor actually just invokes the assign() member function which has the same method signature. What assign() does is replace the current contents of a string within the range specified by InputIterator first and InputIterator last. Recall that std::ifstream is just a typedef for basic\_ifstream<char>, and istreambuf\_iterator<char> is an input iterator that can read successive elements from a stream buffer. If you look up the constructor for istreambuf\_iterator on cppreference, you’ll see that the empty constructor constructs an end-of-stream iterator, and the constructor which accepts a std::basic\_istream initializes the iterator and stores the value of is.rdbuf(). I know that this is perhaps starting to become confusing, but is.rdbuf() is actually what tellg() returns (the file pointer’s position). So in other words, the empty iterator returns and end-of-stream iterator and the non-empty one returns the current position of the file pointer (which is pressumably 0 if we just opened the file). Then we construct a string from these two iterators using std::string’s assing() member function!

String Streams:

Continuing off of file streams, I wanted to also discuss string streams. String streams are kind of like StringBuilder in Java. They allow us to construct and parse strings. To use string streams, we include the <sstream> header. A Stringstream object can optionally accept an initial string as a constructor parameter. Since a Stringstream is a stream, similar to an ifstream, we can use the redirection operator to append strings or emit strings. Recall that the output redirection operator will emit up until the next whitespace character or newline is encountered.

Lambda Expressions:

Lambda expressions are very useful for things like event handlers or threads, where we essentially want to pass in an “inlined” function to the handler or thread dispatcher. As is common with lambdas/closures, we can either capture external variables from the lambda’s enclosing scope, or pass variables as arguments akin to a regular function call. We are able to specify whether we want to capture these variables by value or by reference. For example, assume we have a variable x, which we want to be able to modify in a lambda expression. This would like something like the following:

int x;

giveMeALambdaFunc([x]() { // Capture by value

// Do something with x

});

giveMeALambdaFunc([&x]() { // Capture by reference

// Do something with x

});

Allow me to mention that if we just want to capture all variables which are available within the enclosing scope, we can do [=] to capture all by value or [&] to capture all by reference. An empty capture clause indicates that no variables within the enclosing scope can be accessed. An alternative syntax for capturing x in the example above is shown below:

[&, x]{}; // Capture x by reference

[=, x]{}; // Error: = is the default

Lambdas can be invoked just like regular functions. Arguments explicitly passed in must correspond to the parameter list of the lambda.

Here are the actual order of specifiers that we can include in a lambda expression:

1. capture clause

2. parameter list (optional)

3. The ‘mutable’ specifier (optional)

4. An exception specification e.g., throw() or noexcept (optional)

5. Arrow operator followed by trailing return type (optional)

6. The lambda body

A lambda expression using all 6 of these specifiers would look something like the following:

[&](int a, int b) mutable noexcept -> int {

return (a \* b);

}

Templates:

You may or may not have used templates in other programming langauges before. In case you have never used templates before, I will summarize their utility. A template typically allows us to omit type information for a given variable or function parameter so that we can polymorphically reuse code by substituting the type that we want when the class is instantiated or function is invoked. Due to the fact that we can substitute any type that we like by default, we refer to the placeholder value as ‘generics’. In order to create a template in C++, we simply use the ‘template’ keyword, followed by opening and closing karats (<>), which will enclose one of two additional keywords (I’ll demonstrate what these are in a moment), as well as the name of the generic or generics. Assume that we have a function which adds two values called add(). Now assume that we want to be able to have three versions of the add() function: one for ints, one for floats, and one for doubles. Your instinct may jump to using function overloads. Overloads can become cumbersome, however, since we must type out the same function three times with the only difference being in the types that we assign to our parameters. This is solved with templates with the following code:

template <typename T>

T add(T a, T b)

{

return a + b;

}

When we want to invoke the add() function, we must now do so as add<T>(), where ‘T’ is the type that we want to pass in to our template as the generic. Here is the full code snippet:

#include <iostream>

using namespace std;

template <typename T>

T add(T a, T b)

{

return a + b;

}

int main()

{

double result = add<float>(2.5f, 0.9f);

cout << result << endl;

return 0;

}

If you’re wondering why I’m using double for the result, it’s simply because that is the widest type for floating point, so whether we use int, float, or double, result can store the return value. Anyhow, notice that when we specified float as the generic, the return type as well as both parameters (a and b) became floats. The beauty of this approach is that the compiler will generate a unique add() function at compile time for each type variant that it is invoked with. So for instance, if I were to invoke add() twice, once with int and once with float, it would generate two variants of the function. It is essentially doing function overloading for us behind the scenes. The downside to templates is that we can really pass in any type that we like as the generic and the compiler may or may not complain. For example, heres the same code, but now I’ve changed the function arguments and type for result:

#include <iostream>

using namespace std;

template <typename T>

static T add(T a, T b)

{

return a + b;

}

int main()

{

string result = add<string>("a", "b");

cout << result << endl;

return 0;

}

As you can guess, this is still valid code, but using the add() function to concatonate strings was likely not the programmer’s intention. In order to prevent certain types from being substituted as the generic, we use the static\_assert() function. As you may or may not know, static\_assert(), unlike assert(), performs a compile-time assert. Since generics are substituted at compile-time, static\_assert() can check for types that we want to blacklist or filter out. Something clever that most people do is to use the ‘using’ statement to create a type alias. This can be a bit confusing to follow, especially since I haven’t covered some of the functions being used here, but I’ll throw out the example anyways:

#include <iostream>

#include <type\_traits>

using namespace std;

template <typename T>

using \_T = typename enable\_if<

is\_same<int, T>::value ||

is\_same<float, T>::value ||

is\_same<double, T>::value,

T

>::type;

template <typename T>

\_T<T> add(\_T<T> a, \_T<T> b) {

return a + b;

}

int main() {

float result = add<float>(1.0f, 1.0f);

cout << result << endl;

return 0;

}

Here, \_T is an alias that only exists if T is either an int, float, or double, otherwise the type is undefined and the compilation fails. This works due to enable\_if<T, U>::type, which I’ll let you read up on. The type that we substitute the generic type T for when calling add<T>() will be wrapped by the enable\_if<T, U>::type check, aliased as \_T.

I mentioned two keywords that we could use in the template declaration. The first, we’ve already seen, which is typename. The second is the class keyword. When simply defining templates, these two keywords may be used interchangeably. There are cases, however

Exceptions:

std::optional:

Similar to modern languages e.g. TypeScript or Rust, C++ attempts to mitigate the use of nullptr by providing us with the std::optional<T> template. This template, introduced in C++17, allows us to wrap some data in the std::optional type. An std::optional can either contain no data, or some data. In order to perform a check, we can use the std::optional’s member function has\_value(), which returns true if the std::optional does contain some data, or false otherwise. The value() member function is essentially a getter for the value if the std::optional does contain something. Rather than tediously checking has\_value() followed by value() each time we want to access an std::optional, we can use the value\_or() member function, which either accesses the underlying data if it exists, or returns the value that we provide as an argument if empty. Here is a code snippet example:

#include <iostream>

#include <optional>

using namespace std;

int main() {

optional<int> opt;

cout << "opt: " << opt.value\_or(5) << endl; // Prints 5

opt = 10; // This works because std::optional overloads the assignment operator

cout << "opt: " << opt.value\_or(5) << endl; // Prints 10

return 0;

}

Custom Allocators:

Async Using Promises and Futures:

In C++11 futures and promises were introduced for asynchronous programming. If you’ve not read my POSIX programming notes, that’s okay, I will quickly summarize the difference between asynchronous operations and parallel operations. Parallel operations can only take place if the system supports multiple threads. A task can be subdivided into smaller units of work which can execute concurrently (at the same time) on separate threads i.e., in parallel. Asynchronous programming is built upon coroutines. Coroutines are a fairly simple concept with a fancy name. In essence, each subdivision of the task is assigned to a coroutine (a function). Whenever the main thread encounters a statement of code that may block execution for an extended period of time (these regions are explicitly marked by the programmer), then it can execute one of the pending coroutines as a callback function while it waits for the original statement to finish. In this architecture, the thread is never blocking, but rather continuously completing tasks. The completion of tasks may occur out of order, hence why this is called asynchronous programming. At the end of the day, the thread may still have to block. For example, if it is waiting for the result of some lengthy math operation so that it can print it to stdout, and there are no pending coroutines, then the thread may remain idle. To get a bit more technical, multithreading involves the system scheduler, wherby timeslices are the determinate factor for when context switching between threads occurs. With coroutines/async, the thread performs context switching when the programmer indicates that it should, thus providing less overhead, and more control. This has the added benefit of avoiding dead-locking on shared resources.

With that explanation out of the way, what are promises and futures? A promise is used by the producer/writer of the asynchronous operation. It is the equivallent of using await in node.js. It essentially says “I promise to return the result of this operation to you at a later point in time. Until then, you are free to perform context switching and handle another coroutine”. The future is used by consumers/readers, and its equivallent to the await keyword in node.js. It essentially says “The following statement can only be completed after receiving the promise that the producer/writer promised to give me”. There are three methods of retrieving a std::future:

1. Through a std::promise

2. A call to std::async

3. Through a std::packaged\_task

The std::packaged\_task and std::promise objects contain a member function called get\_future() for this purpose, whereas a function marked with the std::async keyword implicitly returns a std::future. To illustrate the differences between these methods, I’ve taken the snippets from this stack overflow post: <https://stackoverflow.com/questions/12620186/futures-vs-promises>.

/\* Method 1: Using std::future \*/

auto promise = std::promise<std::string>();

auto producer = std::thread([&]

{

promise.set\_value("Hello World");

});

auto future = promise.get\_future();

auto consumer = std::thread([&]

{

std::cout << future.get();

});

producer.join();

consumer.join();

In method 1, we create a std::promise which will contain data of type std::string. We create two threads using lambda notation – one as the producer, and the other as the consumer. The producer sets the value of the std::promise to be “Hello World”. The line where we get the future using promise.get\_future() will essentially add an await call to the queue, which will get fulfilled when the producer thread sets the value of the promise. The consumer waits for, but does not block on the call to get() (although in this case there are no other asyncronous tasks to be completed so it does actually end up blocking in this example).

/\* Method 2: Using std::async \*/

template<typename F>

auto async(F&& func) -> std::future<decltype(func())>

{

typedef decltype(func()) result\_type;

auto promise = std::promise<result\_type>();

auto future = promise.get\_future();

std::thread(std::bind([=](std::promise<result\_type>& promise)

{

try

{

promise.set\_value(func());

}

catch(...)

{

promise.set\_exception(std::current\_exception());

}

}, std::move(promise))).detach();

return std::move(future);

}

The example given by the guy who answered the question is a bit overengineered, so we’ll have to work our way through it. The idea here is that we pass in a function

Multi-threading:

If you have experience with multithreading in C, threading in C++ should feel pretty familiar to you. Of course, given that C++ is an OOP language, the implementation for threads is a bit different. C++ does improve upon POSIX threads by not forcing the user to follow a specific function pointer template for the thread’s routine. Recall in C that threads accepted a function pointer which had to be of type void \*(\*routine)(void \*args). Looking at the man page for std::thread will reveal that C++’s implementation is a bit more convoluted:

template<typename \_Callable , typename... \_Args, typename = \_Require<\_\_not\_same<\_Callable>>> thread(\_Callable &&\_\_f, \_Args &&... \_\_args)

thread (const thread &)=delete

thread (thread &&\_\_t) noexcept

Before I break this down, I just want to preface and say that threads are an object in C++, so std::thread is both the type and constructor. Note that there are three constructor implementations, one of which is deleted. The latter two are both copy constructors, meaning that they just run the same task as the thread that gets passed in as a parameter. Of course, since the second is deleted, this inhibits us from passing in a const reference to a thread. The reason being that the thread must be mutable. Now looking at that scary template... It’s actually not so bad; we simply take in two generics: \_Callable and \_Args. You can probably deduce that these represent the return type and variable argument list type of the function/lambda/functor respectively. The \_Require type is not actually something we pass in, but rather a constraint specifying that the return type of the callable cannot be the same as the type used for the variable argument list. Why this constraint exists, I’m not entirely sure. Let’s now look at a typical implementation of multithreading in C++ that calculates the sum of the square root of consecutive powers of i (of course taking the square root of a power is quite redundant, but it adds some computational time into the mix). Here is the code:

#include <iostream>

#include <vector>

#include <thread>

#include <mutex>

#include <cmath>

#include <iterator>

#define TPOOL\_SZ 10

static float sos; // Sum of square roots

static void calc\_sqrt(float n)

{

auto m = new std::mutex();

m->lock();

sos += sqrtf(n);

m->unlock();

}

int main()

{

auto threads = std::vector<std::thread>(TPOOL\_SZ);

std::vector<std::thread>::iterator i = threads.begin();

for (; i != threads.end(); ++i)

{

float idx = i - threads.begin();

\*i = std::thread(calc\_sqrt, std::pow(idx, idx));

}

for (i = threads.begin(); i != threads.end(); ++i)

{

i->join();

}

std::cout << "Sum of Square Roots: " << sos << std::endl;

return EXIT\_SUCCESS;

}

This is pretty standard stuff as far as multithreading goes. Create a thread pool, use a synchronization primitive like a mutex to lock on critical write segments and then block until each thread is finished using join() before printing the result. Unfortunately, we are doing a bit of a nono with the mutex. Reading the man page for std::mutex tells us that we should not call lock() and unlock() directly, but instead use a scoped lock such as std::unique\_lock, std::lock\_guard, or std::scoped\_lock. The reason that C++ advises us to use lock() and unlock() directly is mostly due to the fact that this ignores Resource Acquisition Is Initialization (RAII). This is a popular idiom within C++ that proposes initialization/acquisition should occur within the constructor of the object, and that deallocation should occur at the end of the object’s lifetime i.e. in the destructor. This is basically the entire idea behind smart pointers.

**The Standard Library:**

If you have experience in C, but not a lot of experience with C++, as was my case, you may get confused by the standard library. As you hopefully know, the POSIX specification defines the implemenation details for the C standard library, and then your compiler is typically what actually fufills those implementation details and determines what happens in the case of behavior that is undefined in the POSIX spec and so forth. My point being, on Linux, we use the GNU version of libc (glibc) with compilers such as gcc, or we can use another implementation such as llvm-libc if using Clang, which is part of the LLVM project. This is not so different for C++. If using gcc/g++ as your compiler, you ought to use GNU’s implementation of the standard C++ library, called libstdc++. If using LLVM/Clang, you ought to use libc++. Even when compiling with Clang, libstdc++ is often still used on Linux. One of the main reasons that libc++ exists if because libstdc++ is under a GPL3 license, meaning that Apple cannot ship it, whereas libc++ is under a custom BSD-based license.

**The Standard Template Library (STL):**

Hopefully I’ve cleared up some of the confusion pertaining to the standard library, however, more confusion arises when discussing the Standard Template Library (STL). Many people assume that the standard C++ library and the STL library can be used interchangeably, or that the standard C++ library includes the STL library, however, neither of these are accurate statements. Let’s first discuss what the STL even is. The STL is a library which implements basic data structures and algorithms, such as vectors, maps, iterators, as well as algorithms for sorting, searching, and data manipulation. The STL was written sometime in the 80s, before the C++ language was even standardized. When C++ was eventually standardized by the ISO/IEC, the language committee decided to model parts of the C++ standard library after the STL. In other words, the standard C++ library implements its own version of the STL, which is similar, and may even look identical in some cases to the end user, though the implementation details may differ internally. Because they are so similar, some people (even those who understand the differences between the standard C++ library and the STL) will claim that the standard C++ library *is* the STL, when in fact, it is not.

**GCC vs G++:**

You may be confused as to when we use gcc and when we use g++. It’s simple really – a quick read of the man page for g++ will reveal that it is near identical to gcc, except built specifically with C++ in mind. It states that you ought to use g++ instead of gcc when compiling C++ projects, primarily due to potential conflicts with flags and so forth.