The Fundamentals of C++

**Preface**

Welcome to yet another of my many computer engineering “documents” or whatever you’d like to refer to them as. This one will be on the fundamentals of C++. Please note that this will not really be an “introduction” to C++. This is not a language that I believe you should start with. In fact, I’d highly recommend that you learn *at least* one other computer programming before learning C++. Why, you ask? C++ builds on the principles of C, adding concepts of Object-Oriented Programming (OOP) to it. I have already written extensive notes for Java and C, which are two languages that will be very beneficial for better understanding C++. Now, if you absolutely only want to learn C++ and nothing else, I suggest you check out codecademy, and perhaps the C++ series of YouTube videos by “The Cherno” along with some other resources on the language. This document assumes that you’re at least familiar with C and will be split into 2 main sections: 1) The differences between C and C++, as well as additions to the language that C++ adds. 2) A more in-depth look at some of the functions, classes, and libraries available to you as a C++ programmer, as well as conventions for good C++ code. I really do enjoy C++ as a language, but perhaps its biggest flaw is that it is so extensible. By that, I mean there are so many ways to accomplish one task, that it is often difficult to make a confident decision as to how to go about solving a particular problem. If, at the very least, you understand the core of C, and how it behaves, you should have a much easier time understanding C++, although I would also hope that you understand OOP principles as well, as those will certainly come into play.

**Introduction**

C++ was created in 1979 by a gentleman named Bjarne Stroustrup. Stroustrup, born in 1950, worked for Bell Laboratories (not to be confused with Bell Canada, the internet/telecommunications company). Stroutrup wanted to modernize the very popular language, C, written by Dennis Richie and Ken Thompson. C++ was originally called “C with Classes” and sought to take the fundamental principles and syntax of C, and simply wrap an object oriented veil over it. C++ was not readily available for public use until about 1985. As of 2021, C is 49 years old, and while it is still taught in order to help better understand how computers interface with software, and how higher level languages are abstracted, it is not very practical for most modern applications. It has mostly been phased out with C++ being its modern replacement. C++ is known for being very lightweight and fast, as was C. 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.

**Basic Syntactical Differences**

I’ll just jump straight into this so that we don’t waste any time. Let’s begin with printing text to the console. In C, we generally used fprint, which would format our variables into a character array, and print it to stdout. You will notice that since C++ uses classes, we often will see namespaces used much more. The syntax for printing text in C++ is: std::cout << “text”; std is a namespace and the two colons :: are used to denote scope. std is C++’s standard library namespace, similar to C’s libc (glibc on GNU/Linux). cout and cin will need to be included by including <iostream> using, you guessed it, the #include directive. The function cout stands for character output stream. You’ll likely recognize the arrows << as being the redirection operator. Therefor, std::cout << “text”; can be read as ‘redirect the string “text” to the character output stream, defined in the std namespace‘. Without std, C++ would search what is called the global namespace, where cout is not defined, and a compile time error would be thrown. We might as well look at C++’s way of character input as well, while we’re discussing streams. std::cin >> variable; is how we take input from the user. stdin will await the enter key and will then redirect the character buffer to whichever variable we place afterwards. 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) Unlike C, std::cin will cast whatever is in the character buffer to the appropriate datatype (which is quite intuitive), meaning we don’t need to worry about the datatype of the variable. When it comes to std::cout, we can additionally 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 in my opinion. eg. std::cout << “Hello world!” << endl;

Another minor difference is that we now have Booleans as a primitive type. These are declared with the keyword bool eg. bool is\_true = false; which is a welcomed convenience over using macros to create booleans in C. Other than that, loops, functions, conditionals, and logical operators are all identical.

**Strings and Vectors**

In C, Strings didn’t really exist as a data type, we would simply have to create a pointer to some block of memory where we could store characters, similar to an array. Arrays were also viewed as contiguous blocks of memory that carried data of a particular type and could be indexed due to the array acting as a pointer to the first block of memory in the sequence. As you know, in an object-oriented language like Java, we have instance variables/wrapper variables/object variables/reference variables (all names for the same thing). This applies in C++ as well, as Strings and vectors are considered to be instance variables since they are instances of their classes. A vector in case you didn’t know is defined mathematically as a list, however, you can just think of it as an array list in the context of C++. Whilst arrays do exist, vectors are common substitutes for arrays. Unlike C, where we would have to #include <string.h> in order to use functions pertaining to strings, string is included in the std namespace of C++, and we can simply call functions pertaining to strings as we would for cout or cin. For example, I will define a string: std::string s = “This is a string”; and then I can perform actions on that string such as s.size(), s.clear(), s.replace, etc. Vectors are declared a bit differently. You may want to brush up on generic types from Java if you are not familiar with those, as vectors work in a similar fashion. A vector is declared as: std::vector<datatype T> vec(size); where datatype T is the datatype of the vector, and size is the size of the vector. Similar to most languages, if we know the data that we want to initialize the vector with, we can use curly braces and comma separated values eg. std::vector<float> prices = {9.8f, 3.4f, 6.2f, 0.98f}; Functions which apply to vectors are also stored in the std namespace, and so we can easily call those as well (in fact, string and vector share almost all of their functions). To add items to the beginning of a vector, we usually use the function push\_back() which will add an element to the end of a vector. To remove this element, we can use pop\_back(), although note that this function has no return value unlike most other programming languages.

**Classes and Objects**

This section of the document is where we are going to begin getting a bit more in-depth. The basic syntax differences should hopefully be fairly easy if you’re coming from C, but classes and objects will get more confusing, even if you’re familiar with a language like Java. I’d like to review how we use header files, source files, and object files, as well as the compilation process before we begin looking at classes and objects. In our header files in C (.h files) we would declare our function prototypes (as well as maybe structs, macros, or extern variables) that needed to be accessible by multiple source files. The source files (.c files) would #include our header files, in order to define the bodies of the functions we declared in our headers. Then, in one of our source files, we would declare main() and our program would begin execution there. Typically, we would call most of our functions from main(). Object files (.o files) were created during the compilation process, and we’re essentially the machine code/object code of our .c files. Finally, during the linking process, the linker would combine all of the object files, as well as all the necessary libraries, and spit out an executable which we could run. This may sound off topic from classes and objects, but I assure you that they are quite entangled. In C++, header files can either remain as .h files, or can use the .hpp extension (I prefer .h, but you could make a case that .hpp is the better choice since it might lead someone to believe that it is compatible with C, when it may not be). Source files, however, must adopt the .cpp extension. Generally, we declare our class outline and method declarations in the .h file, while the .cpp files essentially act the same as they did in C. A class can be created as follows:

#include <string>

class Animal {

//attribute/instance variables

bool fur;

double run\_speed;

std::string name;

public:

//method declarations

void set\_run\_speed(double new\_run\_speed);

double get\_run\_speed();

std::string get\_name();

};

This class might reside in a file called animal.h. Instance variables are defined at the top as they usually are, in order to be used as the objects attributes. public is the access modifier, and states that the following method declarations will be accessible from any file. We might use private for variables or methods which should only be accessible to the class itself, as well as subclasses/child classes. In animal.cpp, we would then define the actual code block/body for each of the declared functions in the animal.h’s Animal class.

#include “animal.h”

void Animal::set\_run\_speed(double new\_run\_speed) {

run\_speed = new\_run\_speed;

}

double Animal::get\_run\_speed() {

return run\_speed;

}

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

return name;

}

and then in Main.cpp, we might have main() defined, and call these functions. Note how each method definition in the animal.cpp file starts with the “Animal” namespace. This is because, even though we #include “animal.h”, we could have other header files included as well, and therefor we need to be specific about which method we are referring to, and which namespace it belongs to. This is required for method definitions otherwise we might accidentally override some other function when we didn’t mean to. One issue that we still have is that we have not created any instances of Animal ie. no Animal objects! In order to do this, we need to create a constructor. Here is the revision to Animal.h:

#include <string>

class Animal {

//attribute/instance variables

bool fur;

double run\_speed;

std::string name;

public:

//Object constructor

Animal(bool new\_fur, std::string new\_name);

//method declarations

void set\_run\_speed(double new\_run\_speed);

double get\_run\_speed();

std::string get\_name();

};

Here we add a constructor that will instantiate fur and name when an Animal object is created. This way we can initialize objects of type animal and set their attributes on creation. Note that if we don’t add our own constructor, there is a default constructor similar to Java (I compare everything to Java since I’m most familiar with it) meaning that we can still create Animal objects without adding a constructor, but we won’t be able to modify any of their attributes without using setter methods. Let’s see what we need to add to animal.cpp:

#include “animal.h”

Animal::Animal(bool new\_fur, std::string new\_name) {

fur = new\_fur;

name = new\_name;

}

void Animal::set\_run\_speed(double new\_run\_speed) {

run\_speed = new\_run\_speed;

}

double Animal::get\_run\_speed() {

return run\_speed;

}

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

return name;

}

Now that we have a constructor, let’s take a look inside Main.cpp where we will declare an Animal object, initialize it’s instance variables, set it’s speed, and print the speed:

#include <iostream>

#include “animal.h”

int main() {

Animal cat(true, “cat”);

cat.set\_run\_speed(23.4);

//prints “cats can run 23.4 km/hour”

std::cout << cat.get\_name() << “s can run ” << cat.get\_run\_speed() << “ km/hour” << endl;

return 0;

}

By default, instance variables are private (although it is never bad to be explicit). This means that if in main(), I had tried to access one of cat’s attributes without using a public getter method, I would have received a compile time error. We’ve discussed constructors, but let’s discuss something a bit unique to C++. Unlike Java, C++ has no garbage collection (after all, it is still a somewhat old language). And in C, we would need to use the .free() method to deallocate memory and allow it to be overwritten. Objects in C++ may need to be deallocated and this is where destructors come into play. A destructor is similar to a constructor in that you are able to run whatever code you’d like before the actual destruction of the object occurs. This is typically used for any extra cleanup that needs to be done. Here is the syntax for a destructor:

#include <string>

class Animal {

//attribute/instance variables

bool fur;

double run\_speed;

std::string name;

public:

//Object constructor

Animal(bool new\_fur, std::string new\_name);

//Object destructor

~Animal();

//method declarations

void set\_run\_speed(double new\_run\_speed);

double get\_run\_speed();

std::string get\_name();

};

That’s right, it’s essentially like the constructor but without the parameters and it uses a tilde to differentiate itself. You don’t need to put anything inside the definition of the destructor, but you **do** need to include it in your .cpp file even if the body remains empty. I don’t think it is necessary for me to show you that.

Hopefully, that gave you a very basic idea of how creating classes and objects works in C++. The Animal example was certainly not my finest, however, I aimed to get the core concepts in your head, and the rest will come with practice and viewing other examples as we continue throughout this paper. The basic idea is that .h/.hpp files generally house our classes (though not always; they can appear within our .cpp files as well), our functions/variables that we want to be accessible by multiple files should be under the public: scope identifier, and those functions require definitions in the respective .cpp file (with some exceptions that we are about to look into).

**Function Nuances in C++**

As you’ve noticed, the general syntax and principles of functions/methods in C++ have remained synonymous with C; however, there are many subtle nuances and tricks that are new inclusions in C++ that we ought to go over. The first thing I’d like to go over (since I just alluded to it) are **inline functions**. While these technically exist in C, they are sort of broken, and unused in that language. I said that the function declarations require a definition in the .cpp file (with some exceptions) and the exception is when we declare and define a function at the same time in the header file. Inline functions begin with the keyword “inline”, and we simply oust the method declaration, leaving only the definition. eg.

inline

void eat() {

std::cout << “Nom nom\n”;

}

This seems very practical and nice right? Well, it depends... Sometimes it can benefit execution speed, but sometimes it can hinder it as well. Generally, I just avoid it, but you will likely encounter it at some point. You could argue that it’s useful for short functions, but I’d argue back that lambda functions are even more practical, and we’ll be looking into those.

The next topic I’d like to cover is **default arguments**. I really like this about C++. If you have a method where you know that the argument will be the same most of the time when calling it, 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 default to the one we provided. Similar to var args (variable arguments), default arguments need to be placed last in the set of parameters.

A third addition to C++ from C is **function/method overloading**. If you already realized that C++ has function overloading when we overloaded the default constructor, I applaud you. I totally forgot that constructors were examples of overloading a function until now. But yes, it works the same as in other OOP languages. 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 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 method.

The final nuance of methods in C++ 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 essentially solve the same issue as method overloading but makes things even more streamlined. 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 return\_some\_value(T some\_value) {

return some\_value; //literally a useless function

}

Then the return type would be of type T which would be interpreted as whatever some\_value is. You can likely see why vector uses a template for most of its methods. Since a vector can be of any data type, its methods must be able to perform the same actions regardless of type.

**Pointers vs References**

In C++, 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 this syntax, C++ has an alternative solution called references. In essence, references accomplish the same goal as pointers allowing for changes in stack space to be permanent instead of being destroyed when the function’s return call occurs. There are differences between the two, however. The syntax is a bit less messy for references. We use the ampersand similar to how we would use the asterisk when declaring a pointer eg. datatype &ptr\_name; in order to differentiate the variable as a reference. After declaration, the ampersand always refers to a variable’s address. In order to set a reference to the address of another variable, we just use the equal’s sign. It is implicit that, because the variable is a reference type, denoted by the ampersand, that the assignment to a variable is, in fact, the assignment to the variable’s address. For example: int& ptr = num; Since ptr is a reference type, the equals sign implicitly suggests that ptr will contain the address of num (unlike C’s pointer types, where the assignment to the variables address was explicitly stated with the ampersand eg. int \*ptr = &num;) Another difference between pointers and references is that pointers would need to be dereferenced to alter the value at the pointers address. References act more like aliases; in that they can be thought of as an alias for another variable. So, in the example above, referring to ptr would be the same thing as referring to num. They are simply two names for the same thing. The difference between them is that a reference declared as a parameter for a function will store it’s data in heap space rather than stack space, and therefor can modify an argument’s value permanently. Another difference between pointers and references is that references cannot be of type void. That feature is exclusive to pointers. Take a look at this example code:

int main() {

vector<int> vect{ 10, 20, 30, 40 };

// We can modify elements if we use reference

for (int &x : vect)

x = x + 5;

// Printing elements

for (int x : vect)

cout << x << " ";

return 0;

}

Here we demonstrate how we can modify values in a vector using a reference. The for-each loop/enhanced for loop can be read as “For each element in vect”. &x implicitly sets itself to the address of the next element in vect upon each successive iteration. Since x is essentially an alias for the variable it’s set to, the line: x = x + 5 is equivalent to saying vect[i] = vect[i] + 5 (where i is the index pointer of the for-each loop).

Perhaps two of the final things about references that we should go over are the const keyword and the nullptr keyword. You are surely aware by now that const prevents a value from being altered throughout the runtime of the program. 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 method, 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 dont\_change\_me) {...} we could instead use a reference: func(int const &dont\_change\_me) and that would use the address of the argument rather than making dont\_change\_me it’s own local variable on the stack. The other keyword, nullptr is the C++ alternative to NULL when declaring pointers. If you recall, it is bad practice to declare a pointer and leave it unassigned in C. This is because the pointer must be assigned to some address no matter what, so if we forget that it is unassigned and begin using it, it will point to some junk data and likely end in a seg fault. Pointers in C++ should use nullptr instead of using NULL because of some datatype issues that might arise.

**Namespaces**

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 setting a default namespace similar to how we have default arguments for our functions. By using the “using” directive, we can specify a namespace for C++ to assume if no namespace is explicitly provided. Most commonly, after all of the other directives are defined, you will see the line “using namespace std;” This 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 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: namespace2::foo(); We can use the “using” directive multiple times in our code if we want to switch from std being the default to a different namespace being the default.

Going back to “using namespace std;” for a moment; this is quite a controversial line for many C++ programmers, and I think I’m inclined to agree that it should be discouraged. When we “using namespace std;” we ignore half the reason that namespaces exist in the first place. Why is this only controversial for std though, and not other namespaces. Well, if you state that you’re using a different namespace than std, you’re likely only using that namespace for a brief period of time. More importantly however, std contains many functions, all with relatively simple names. Often times, other code libraries will copy the names provided in std, but use a different namespace. If we are unclear in our code, things tend to get messy. You likely won’t notice this for personal small applications, but this stuff makes a huge difference for enterprise scale applications.

**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 can go your entire C++ career without ever creating a struct. However, it is up to you how or if 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 not complex and won’t require any functions to be called on it’s data, like a vertex for example. I’d use a class for more complex logic, if I want to be inheriting from other classes, or creating copy constructors, or calling 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**

In order to extend a child class to a parent class ie. inheritance, it’s as simple as using the following syntax: class DerivedClass : access-specifier BaseClass { ... }; where access specifier is public, protected, or private (although you can’t extend a private class so it’s really either public or protected).

**Setting Constructor Values**

If you’ve been paying attention closely, you may have noticed that in my example about struct vs. class, that I initialized the instance variables using the this-> keyword, similar to what you’d do in Java. We’ve also seen that you can simply name the instance variables one name, and then name the parameters another name, and then the compiler won’t get confused. Both of these ways are not ideal, however. Likely the most preferred method is an initializer list. The syntax for an initializer list is just a colon followed by each instance variable (comma separated) and input parameters next to each instance variable. For example, say I have a constructor:

int instance1;

double instance2;

float instance3;

public:

Initializer(int inst1);

Then in the definition, we will use an initializer list to fill out the values for each instance variable:

Initializer::Initializer(int inst1) : instance1(inst1), instance2(67.54), instance3(10.4f) { ... }

For values that are to be set at the time of object creation (instance1 in this example), the parameter for the constructor should still be given a different name to avoid the compiler accidentally getting confused, however, for instance variables that will have a default value, they can be set directly in the initializer list.

**static, const, auto**

Static, and auto each function slightly differently than they did in C, and I didn’t really explain const very well in the C document, so I feel that it’s important that you understand when to use them.

static: static takes on slightly different meanings depending on whether you’re using it in the context of a class or not. If a variable is declared static outside of a class, then this will affect the visibility of that variable to the linker during the linking phase. Essentially, static tells the linker to ignore other variables or functions with the same name in other translation units (.o files). It sort of makes the variables private in a sense. So, for example, I might have a variable that is declared as extern, that is to say that its definition is defined in another file eg. extern int findMyValueElsewhere; and its definition might be in a different .cpp file: int findMyValueElsewhere = 10; Printing the value of findMyValueElsewhere would yield 10 as expected, but if I were to make the definition of findMyValueElsewhere static, then it would not be visible across translation units and would give a linker error. If the extern definition was in the .h file included by the .cpp file then you would not get this error since static only secludes the variable from being seen by other translation units (.o files, like I said).

When we use static in a class, it functions as it would in Java. static means that the variable belongs to the class, and therefor there is only one instance of the variable that is shared by all instances of the class. If we change the static instance variable of one object, that will change it’s value for all objects of that class, hence why static variables should be accessed like this: ClassName::staticVar and not like this: instanceOfClassName.staticVar

const: const, as you probably know, prevents a variable from changing. 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 foolproof safe. const gets more interesting when used with pointers though. If they keyword is entered before the asterisk, this prevents the value at the address that the pointer’s pointing to from being changed. If const is entered after the asterisk in the pointer declaration, then the pointer’s value ie. the address it contains is prevented from being altered. Ex.

int a = 5;

const int \*ptr = &a

\*ptr = 4; //Throws a compile error

int a = 5, b = 9;

int\* const ptr = &a;

ptr = &b; //Throws a compile error

Hopefully, you are able to see the difference there. const can also be used with functions to prevent any values from changing within the function. This would look as follows:

bool MyClass::isTrue(bool test) const {

return (test)? true : false;

}

Note that using const in methods only works for class methods, not random methods that are declared outside of a class. Since we do not need to modify the contents of any of the variables within isTrue(), we make it const to add clarity to our code. If by chance we really needed to modify just 1 variable within a const method, and we’re sure that it won’t affect anything else, we can declare that variable with the keyword “mutable” eg. mutable int bypass; which will be allowed to change within const methods and not throw any errors.

auto: auto in C was simply the default keyword for local variables, but it functions much differently in C++. Instead, auto essentially allows us to create variables with interpreted data types. So for instance auto guessWhatIAm = 89.4f; and auto will assume that guessWhatIAm is a float since it was given a float literal. This really ought not be abused as it can definitely lead to spaghetti code and errors. It is most helpful for when you have very long declarations. Like for instance, if you had a typedef with some long name encapsulated in a long namespace, then instead of typing out: thisIsMyReallyLongNamespaceName::thisIsMyStupidTypedefName ohYeahAndTheVariableNameToo = 9;

We could shorten that by declaring it as auto ohYeahAndTheVariableNameToo = 9; typedefs actually a really bad example because auto would just assume the data type of our typedef but whatever, I think I got the point across, it’s best used for long data types like objects.