The C++ Programming Language

Neil Kingdom

October 10, 2024

Contents

1 Abstract

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 Cherno's 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 trying to overcome these barriers.

2 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.

3 The Myth of Backwards Compatability

You will hear it 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 once you begin to use more modern versions of C such as C11 and onwards.

4 Basic I/O

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

```cpp
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).

5 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 constructor overloads 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.

6 C++ Header Files

The standard C++ library has copies of all of the standard C library headers. An exhaustive list can be found here. 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.

7 C++ Function Nuances

C++ has a couple new features when it comes to functions which C did not have, so I'd like to highlight some of them in the following subsections.

7.1 Default Arguments

The first addition to C++ functions that I'll 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 e.g.

```
void foo(int param1, int param2 = 5)
{
    // Do something
}
```

Now if we call foo() with only one argument e.g. foo(3), param2 will be 5, but if we call 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. i.e. you cannot have a default parameter followed by a non-default parameter.

7.2 Function Overloading

The other addition to C++ which C did not have is function 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.

```
void foo();          // No parameter version
void foo(int a);     // Parameterized version using int
void foo(float a);   // Parameterized version using float
```

8 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
{
    public:
    // Constructor/destructor
    Animal(std::string name, bool has_fur, double speed);
    ~Animal();

    // Member functions
    std::string get_name();
    bool get_has_fur();
    double get_speed();

private:
    // Member variables
    std::string m_name;
    bool m_has_fur;
    double m_speed;
};
```

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.

```cpp
#include "animal.h"

Animal::Animal(std::string name, bool has_fur, double speed)
{
    m_name = name;
    m_has_fur = has_fur;
    m_speed = speed;
}

Animal::~Animal()
{}

std::string Animal::get_name()
{
    return m_name;
}

void Animal::get_has_fur()
{
    return m_has_fur;
}

double Animal::get_speed)
{
    return m_speed;
}
```

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:

```cpp
#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.

9 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.

10 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.

10.1 static

The static keyword has not really changed since C. It 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.

10.2 const

As you probably know, const 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();
}
```

10.3 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.

11 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:

```cpp
#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.

12 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 e.g. 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 missing the point as to why namespaces exist in the first place, so proceed with caution.

13 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();
};
```

Note that the instance variables are private by default, and are therefore not visible to anything external of the class itself, 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:

```cpp
#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 need to group related data which doesn't comport itself well as an "object" per-se. 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.

14 Object Oriented Programming (OOP)

At this point in the document, we're going to take a bit of a detour into what I would argue sets C++ apart the most from a language like C, which is its object-oriented capabilities. C++ has some pretty interesting ways of handling OOP, to say the least.

14.1 Inheritance and Polymorphism

One of the most difficult aspects to grasp in C++ is how it deals with inheritance. You see, C++ was really one of the first languages to popularize the OOP paradigm, and as a result, a few questionable design choices were made. Let's discuss a few of those design choices. The first design choice, which was intentially chosen 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 are implicitly derived. 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 a diamond shaped dependency graph when visualized hierarchically. The issue here 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 consider analysing this example for reference:

```cpp
#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 concienciously design your code in a manner which avoids 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.

14.2 Virtual Functions and 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 including 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 explicitly declare the function as virtual.

14.3 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:

```cpp
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.

14.4 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:

```cpp
class A {
    virtual void foo() {
        // Some code
    }
};

class B : public A {
    virtual void foo() override {
        // Some code
    }
};

class C : public B {
    virtual void foo() final {
        // Some code
    }
}

class D : public C {
    virtual void foo() { // Error: Declaration of 'foo' overrides a 'final'
     ↪  function
        // Some code
    }
};
```

15 Accessing Private Members Using the friend Keyword

In my opinion, perhaps one of the most contentious features of C++ is its ability to allow other functions or classes to bypass access restrictions using the friend keyword. A class may opt to make its members accessible to other specified classes or functions without making them public to all external entities. In other words, we can make other global functions or classes a trusted 'friend' of the class, stating that they have exclusive permissions to bypass any access modifiers or implicit access restrictions. Let's first look at how we can make another class a friend of our class. This will allow the other class to access all members of the current class at any moment:

```
class A; // Forward class declaration

class B
{
public:
    B() = default;
    ~B() = default;

    void foo(A &a);
};

class A
{
public:
    A() = default;
    ~A() = default;

private:
    std::string b_can_access;
    friend class B;
};
```

Here, we've marked B as a friend of A. This allows B to access all members belonging to A that are private. In the example, the only private member of A happens to be b_can_access

16 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 {
public:
    Foo(int integer, float decimal, std::string sentence);

private:
    int m_integer;
    float m_decimal;
    std::string m_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:

```cpp
class DelegationExample
{
public:
    DelegationExample();
    DelegationExample(int n);
    void setNumber(int n);

private:
    int number;
};

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
{
private:
    NewFoo f;
};

// Version 2:
class Bar2
{
 public:
     Bar2() {} // f not in list

private:
     NewFoo f;
};

// Version 3:
class Bar3
{
public:
    Bar3() : f() {}

private:
    NewFoo 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
}
```

This example was borrowed from this stack overflow post: Initializer list for objects with default constructor

17 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.
- **Move constructor**: Works similar to the copy constructor, but transfers ownership of the reference to the rvalue into the lvalue, thus invalidating the rvalue.
- **Copy-assignment operator**: Similar to the copy constructor, but overloads the assignment operator (=). The right operand (rvalue) of the assignment expression is passed into the assignment operator function overload as a reference and the left operand (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.
- **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.

17.1 Deleting 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:

```
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.

17.2 Creating Default Definitions of Special Member Functions

We are able to spare some typing by utilizing the ’default’ keyword in C++. Similar to delete, default was also introduced in C++11. The default keyword is primarily useful for the aforementioned special member functions which are implicitly defined for us. We can make these implicit declaratios more explicit by utilizing the the default keyword. Up until now, you’ll have seen me make a lot of empty destructor definitions, which is not necessary. Here’s an easier way of doing the same thing:

```
class Foo
{
public:
    // Explicitly tell the compiler to generate a default contructor/destructor for
    ↪  Foo
    Foo() = default;
    ~Foo() = default;
};
```

18 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. Move constructors allow us to transfer ownership of a value from one entity to another. We invoke the move constructor via std::move(). In some cases, C++ uses move semantics implicitly, such as when returning objects from a function. Move semantics can be used in regular functions as well.

The difference between moving and copying is that copying allocates memory for the copy and then copies the contents of the initial value over to the newly allocated space, whereas moving an object effectively creates a pointer to the original object but invalidates the original owner so that we don’t have two pointers accessing the same memory. Unlike a language like Rust, C++ does not enforce this invalidation, meaning that we can still access the contents of the original owner, though they will be in an unknown state and their contents may get overwritten. Move semantics utilize a double ampersand to indicate that were moving, not just borrowing a reference to the original object. Here’s an example which illustrates a valid usage of the special move constructor member function:

```
#include <iostream>
#include <utility>

class Foo
{
public:
    Foo() = default;
    Foo(Foo &foo) = delete;
    Foo(Foo &&foo) = default;
    ~Foo() = default;
};

int main(void) {
    Foo a = Foo();
    Foo b = std::move(a);
    // a is invalidated. Don't use it unless you're a sadist
}
```

Note in this example that if the move constructor were deleted, then the call to std::move() would fail. std::move() belongs to the utility header, so that ought to be included. Note that explicitly declaring the move constructor using the default keyword will actually implicitly delete the copy constructor because it assumes if you declare one explicitly then you would have meant to declare the other explicitly. In other words, if you intend to use both the copy and move constructors, either declare both explicitly with the default keyword, or declare neither and have the compiler generate them implicitly for you. The same logic applies to the copy-assignment and move-assignment constructors.

19 constexpr

The C++ keyword constexpr, pronounced ”const expression” was introduced in C++ 11. It informs the compiler that a variable or function can be computed once at compile time, and then substituted everywhere it’s referenced. The constexpr keyword shares the same property as the const keyword, meaning that variables or functions declared with constexpr will be immutable. Here’s an example of how to use constexpr:

```
constexpr float circumference = 2.0f * M_PI;
```

The variable circumference would normally be computed at runtime each time it’s referenced, however, placing constexpr before the statement will cause the result to be computed at compile time and then reused, similar to how macros work. Note that expressions marked with constexpr are only valid if all entities in the statement are also marked with constexpr (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:

```cpp
#include <cmath>
#include <complex>

// Fails since std::exp() is not a constexpr
constexpr std::complex<double> eulers_ident = std::exp(std::complex<double>(0.0,
↪   M_PI)) + 1.0;
```

20 C++ Style Casts vs C-Style Casts

21 Operator Overloading

22 Enums

Enums in C++ come in two forms. We have C-style enums, and then we have class-based enums, which are a new C++ addition. Let's start with C-style enums. These are effectively identical to how you would declare an enum in C, but with one minor difference, which is that we can optionally specify an integer type as the underlying type used by the enum. Here's an example:

```cpp
enum Color : uint32_t // Optionally specifying uint32_t as the underlying type
{
    RED     = 0xFF0000FF,
    GREEN   = 0x00FF00FF,
    BLUE    = 0x0000FFFF
};
```

Class-based enums are pretty much the same as C-style enums, but with some slight differences. Class-based enums treat their members as if they were static, meaning that in order to use an enum, we have to access it through the scope of the outer class e.g. ENUM::TYPE. The other distinguishing feature is that we can't perform implicit conversions from a numeric type to a class-based enum, whereas we can do implicit conversions between integer types and C-style enums. This is definitely a good reason to prefer class-based enums in your code. Here's an example:

```cpp
enum class DrawCmd
{
    REDRAW,
    FILL,
    CLEAR
};

// ...

DrawCmd cmd = DrawCmd::REDRAW;
if (cmd == 0) // Invalid operands to binary expression ('DrawCmd' and 'int')
{
    // Do something
}
```

Notice how the if statement throws an error because even though the value of cmd does happen to be 0, the compiler treats DrawCmd::REDRAW as a distinct type apart from int. This code would work fine for C-style enums, however, since it can implicitly cast C-style enums to integers.

23 Smart Pointers

A lot of C++ developers will claim that using the new and delete keywords is evil, because raw pointers are evil, and are to be shunned immediately. I personally think this is a very dramatic take, but I will admit that smart pointers are a step in the right direction, and this idiom is being adopted in other languages like Rust. What makes smart pointers appealing is that they are automatically deallocated when they go out of scope. This means that we don't have to manually manage freeing/deleting allocations, which is nice because sometimes ownership can get confusing. Smart pointers will still invoke new and delete under the hood, so your constructors and destructors will work in the same way they did when manually invoking new and delete. Let's take a look at all three smart pointer types and where they are useful.

- **unique_ptr:** Unique pointers are the most common type of smart pointer in C++. A unique pointer is named such because it has sole ownership over the object it encapsulates. This means the only way to transfer the value held by the unique pointer would be via move semantics. Usually, it's good practice to only have one owner over any given object at a time anyways, so unique_ptr should be the default go-to. There are two ways we can create a unique_ptr. The first method is to pass a new instance into the unique_ptr's constructor and the second method is to use the std::make_unique() function. Here are examples of both:

  ```
  #include <memory>

  // Using new instance passed to constructor
  std::unique_ptr<Foo> a(new Foo());

  // Using std::make_unique()
  std::unique_ptr<Bar> b = std::make_unique<Bar>();
  ```

  The advantage of std::make_unique() is primarily that it performs error handling and will throw an exception if something goes awry.

- **shared_ptr:** The second type of smart pointer in C++ is known as a shared_ptr. Shared pointers, as the name would suggest, allow you to have multiple owners pointing to the same data. Shared pointers are also sometimes called reference count pointers. The reason they're sometimes called reference count pointers is because they have an internal member variable which increments each time a new reference is lent and which is decremented each time a reference is dropped. The internal count will start at something like 0. If the count is then decremented to -1 (indicating that the final reference was dropped) then the underlying memory is deallocated. The code for creating shared pointers is effectively the same as for creating unique pointers, except we use shared_ptr as the type rather than unique_ptr and std::make_shared() instead of std::make_unique();.

  It should be noted that unique pointers and shared pointers fall under the same category of being *strong* references. This simply means that, unlike the next type of smart pointer we'll look at, the underlying memory is deallocated when we either go out of scope (in the case of unique pointers) or when the reference count reaches whatever number is indicative of there being no remaining owners (in the case of shared pointers). The use_count() member function of shared pointers will print the strong count thereby indicating how many active references we have to the shared object.

```
#include <memory>

std::shared_ptr<Foo> a = std::make_shared<Foo>();
std::cout << "Strong count: " << a.use_count() << std::endl; // 1
{
    std::shared_ptr<Foo> b = a; // Shares the memory address held by a with b
    std::cout << "Strong count: " << a.use_count() << std::endl; // 2
}
std::cout << "Strong count: " << a.use_count() << std::endl; // 1 (since b was
↪   dropped)
```

- **weak_ptr:** Our final type of smart pointer is known as the weak pointer. Weak pointers, unlike unique and shared pointers are considered to be a weak reference (as opposed to a strong reference). Weak pointers only really make sense in the context of shared pointers. A weak pointer can borrow a reference to the memory held by a shared pointer, but unlike creating a new shared pointer, the weak pointer will not decrement the reference count of the original owner when it goes out of scope. A weak pointer doesn't really care if the originating shared pointer has been destroyed or not. Of course, we need to perform some safety checks before we use a weak pointer in case the original owner *has* been destroyed. Because of multithreading, we also need to create a lock over the weak pointer while we use it, so that another thread doesn't accidentally drop the last strong reference to the memory, thereby making further operations with the weak pointer invalid. We can do this using the lock() member function of std::weak_ptr. Here's a sample:

```
#include <memory>

std::shared_ptr<Foo> a = std::make_shared<Foo>();
{
    std::weak_ptr<Foo> b = a;
    if (auto p = b.lock())
    {
        // Do something with b
    }
    else
    {
        std::cerr << "Could not lock weak pointer b" << std::endl;
    }
}
```

When dealing with smart pointers of any kind, if you need to convert them into raw pointers, you can use the get() member function. This will effectively return the underlying memory held by the smart pointer.

24 STL Containers

25 Iterators

26 File I/O

26.1 String Streams

27 Templates

28 Newer C++ Features

28.1 Type Checking Using 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>));
```

The std::is_base_type<Base, Derived>() function will check if the function passed in for Derived is derived from the class passed in for Base. Under the hood this function uses a combination of the decltype keyword and static casts (which we'll look at a bit later on).

28.2 std::initializer_list

28.3 std::optional

28.4 Lambda Expressions

29 Exceptions

30 Custom Allocators/Deallocators

31 Synchronization

31.1 Promises and Futures

31.2 Multithreading

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 by reminding you that threads objects in C++, so std::thread is both the type and constructor. As you can see, we have a templated thread class which contains three overrides of the constructor. 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 both the function/lambda/functor used by the thread and the type of the variable argument list, respectively. The _Require type is not actually something we pass in, but rather a constraint specifying a certain type that _Callable *cannot* be. I am not actually positive as to what this does, but I think it prevents passing in another thread as the callable type. 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 <array>
#include <thread>
#include <mutex>
#include <cmath>
#include <iterator>

#define TPOOL_SZ 10

static float sos; // Sum of square roots

static inline void calc_sqrt(const float n)
{
    auto m = new std::mutex();
    m->lock();
    sos += sqrtf(n);
    m->unlock();
}

int main()
{
    int count = 0;
    auto threads = std::array<std::thread, TPOOL_SZ>();
    std::array<std::thread, TPOOL_SZ>::iterator i;

    for (i = threads.begin(); i != threads.end(); ++i, ++count)
    {
        threads.at(count) = std::thread(calc_sqrt, std::pow(count, count));
    }

    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. In this scenario, we could also use an atomic type instead of the mutex like so:

```cpp
#include <iostream>
#include <array>
#include <thread>
#include <atomic>
#include <cmath>
#include <iterator>

#define TPOOL_SZ 10

static std::atomic<float> sos; // Sum of square roots

static inline void calc_sqrt(const float n)
{
    sos = sos + sqrtf(n);
}

// ...
```