ProgrammingAbstractions-Lecture27

Instructor (Keith Schwarz):I'm guessing by the fact that I can see myself that we're on. Welcome back to CS106L. My name's Keith. I'm a section leader here. I teach CS106L, which is the standard C++ program and laboratory. I recognize a few people here, which is great.

Julie wanted me to come in today and talk to you about the C++ programing language, what's it look like outside of CS106B? What sorts of things do you need to be aware of to just kind of give you a general picture of what this language looks like?

Before I get into the actual C++ stuff, I wanted to start off by congratulating all of you for making it through this class. That's not a small accomplishment. If you think about when you started, day one, probably looking at this screen right here, probably thinking, what's that [inaudible], what's Genlib, what's C [inaudible], and why's it really, really less than something? Look where you are now.

You now know how to be a client of the vector, the stack, the map, the queue and the set. You know that recursion, you know pointers, you know Linklis and binary [inaudible] and graphs. You know algorithmic analysis and graph algorithms and searching and sorting. You know how the lexicon works. You know how to implement all of these classes. You know data abstraction. That's not a small accomplishment. That's really something to be proud of.

What I wanted to say is that these are real, practical tools that will follow you, no matter where you take them. Some of you might go on in CS. You might think, wow, I really like this class. I want to see where it goes. That's great. No matter where you take it, if you go and take algorithms, you take compliers or data bases or operating systems, the skills you've learned here is really going to be the foundation of all of your programing. You're always going to be using maps. You're always going to be using vectors. You're going to need recursion, no matter where you apply it. It's really wonderful that you have these skills.

Of course, some of you aren't going to go on in computer science. That's totally fine. You took this class and said, it's not for me or, you know, maybe not. Well, that's fine too because whether you go into sociology or philosophy or psychology or math or physics or chemistry, there are problems to be solved, and there are problems you can solve with a computer.

What you've learned in this class is the most valuable skill of them all, which is how to take a problem, model it and solve it. That really is something.

The skills you've learned here transcend any specific programing language. You can do everything you've done in here, in Java, in C++, in PHP, in Python. Everything that you think of, you will be using these same tools.

That said, we've taught you everything in this class using the C++ programing language. We use C++ because it's very good for expressing the concepts that we were going over. It has great support for recursion, has exposed pointers, so you can do things like Linklis and binary trees quite nicely.

It has objects so you can do data abstraction. It's got templates. It's a very good mix of different programing strategies and different programing styles that means we can teach you things without having to bog you down in language syntax. That said, this really hasn't been much of a class in C++. Think of it was we all put you inside the C++ bus and drove you up to the top of the mountain of CS106B knowledge. You know that the bus got you up there, but you don't really know what's going on under the hood.

That's okay because what you learn in this class is more important than that. All the programing language and knowledge in the universe is not going to help you solve a problem if you don't know how to use a vector, you don't know how to make a map, and you can't make a recursive function solve things.

What I want to do today is talk about what the C++ language is. What's it look like the real world? What kind of stuff do you need to know? More importantly, how do you take these skills that you've learned, which are very generic, and go and apply them in this language.

I love C++. I've been using it for years. I think it's a beautiful language. It's got support for so many different programing paradigms. It's easy to use once you get a handle on it. It lets you solve problems in so many different ways. Really, I just want to show you what it looks like.

Think of this as an appetizer. I'm just going to give you the – this is the C++ appetizer plate. I'll show you a little bit of this, a little bit of that, give you enough working knowledge that you know where to go to get help. You can see this is what you need to learn, get an overview of what we've been doing, what we've provided you, what we haven't provided you, so that by the time you're done, you can think, maybe I want to go on in this language.

I'm not going to force you to learn C++. I can't do that. I'm not going to follow you around, did you learn C++ yet? Did you learn C++ yet? No, I'm not going to do that. You guys are all competent programmers right now. You know how to solve these problems. You know how to take on these challenges. So I just want to show you the language so you can decide whether or not you want to pursue it.

Maybe you will. That'd be great. If not, that's okay, too, because you know how to tackle problems, and all you need now is a language to do it in.

I want to show you basically four aspects of C++. First some philosophy and some history because every language has a personality. Every language wants to treat you as a programmer a different way, give you some different options. So I just want to show you what to expect when you're working with C++. As a language, it has a lot of criticisms. How many of you have heard bad things about C++ in any way, shape or form?

Wow, every, single person in this room, basically. So there's a lot of criticisms leveled at this language, and what I want to do is show you what the mindset is so you can take a look at these criticisms and evaluate them. In all seriousness, most of the criticisms leveled at this language are criticisms with things a language doesn't try to do.

At the same time, if you get this philosophy and you get this history, you'll understand where this language came from, what it's designed to do and where it's going. The second thing I want to talk about is the actual mechanics. What does the programing language look like? So I'll talk about the libraries, and I'll talk about the core language features.

What happens when you take away things like [inaudible] and Synfi? What are you going to have to do now that you don't have those? In terms of it's actually pretty straightforward and you already know most of it.

Then some language features that we didn't cover, things that you would expect to see in a professional setting that we didn't go over in this class because it's needlessly complicated and doesn't have much to do with this general data structure and algorithm challenges we've been giving you.

Finally, just some references. Where do you go? You want to learn C++? Great. We have a list of wonderful books and resources you can go and reference, and it can get you up to speed very quickly. So it's kind of a game plan. See what's out there, have some fun with it and learn where to go. Sound good?

Okay. So I want to start off with a quick history of C++. So C++ did not come into being one day when a whole bunch of programmers came into a room, made some demonic incantations over a bubbling cauldron and walked out. It didn't happen. It was not invented in one day. People did not sit down and say, our language is going to look like this. It has a history. It has evolution, and you can see different traces of we tried doing this. It didn't work. Let's go add this feature, etc.

It started off with this guy. So I will attempt to pronounce his name. It's Bjarne Stroustrup. I've been told that the way to pronounce this is to say Stroustrup with a Norwegian accent while cramming a potato down your throat. I'll just get it close.

Anyway, he's a Danish computer scientist, and he was getting his PhD in computer science from Cambridge, and his specialty was in distributive systems and operating systems. So his goal was, let's take some problems, spread it out over several computers, have them communicate via network protocol and get some results.

He chose to write this program in a language called Simula. I confess, I don't know much about this language. I haven't heard anyone using it, but back then, it was the his object-oriented language, and he said it helped him think about the problem. It had classes, so you could build a computer object, a protocol object, a network object, and they'd send messages to each other the same way that a physical computer, a physical network and a physical protocol would communicate with each other.

He got it working pretty fast and ran it, and much to his surprise, he found out that it was so abysmally slow that he could get no practical results out of it. The question is what went wrong? It wasn't his program. It was the Simula language implementation. He actually ran a profiler on his code to figure out why it was going slowly.

80 percent of the time his program was running, it was going automatic garbage collection, even though he was doing [inaudible] memory management. So think about this. You spend several weeks writing a program. You run it, and for every line of code you're writing, four lines of meaningless code that someone else wrote are also executing. That's really slow.

So he had to change approaches. He went and rewrote this entire program in a different language called BCPL, which is related to C and the B programing language that came before C. Of course, it's very low-level, it's very fast, but it didn't have these nice high-level features in it. He got it working, but it took a lot of time.

When it was done, he decided, I'm never going to tackle a problem like this again until I have a proper tool for the job. He ended up at AT&T Bell labs, which is the same place that the C programing language came out of. In fact, he knew Kernighan and Ritchie, who invented that, and came up with this language called C with classes.

It was a precursor to C++, and people loved it. It combined the best features of Simula, which is this object-oriented design, with the best features of C, which is the runtime efficiency. It was something that was fast, flexible and helped you think about problems. What he would do is work on this implementation. He had people actually using the C++, and when they needed new features, they said, Bjarne, it doesn't work. He'd go fix it.

We had this cycle of real programmers solving real problems with this developer saying, okay, let's go fix it. After a while, you ended up with C++. I think this is the 25th anniversary of C++, so how exciting.

I want to talk about the philosophy a little bit. What is really driving the development of this language? What does it expect out of you? So I have some quotes from this book called The Design and Evolution of C++. It's a good book. You should read it.

The first one, C++'s evolution should be driven by real problems, and you don't want to get down on a quest for perfection. Basically, this language is designed to solve real problems that real programmers like you are going to run into in not necessarily the best way. It tries to do a good job. It's not intended to be perfect. There are some sloppy edges, but it works.

Because it is driven by real problems and real design issues, you can solve real problems with it. If you think that's a good thing in a language, I personally do. I think it's good to have a language that can solve problems, then C++ is a good choice.

This is probably the one that you've seen the most. Don't try to force people. C++ gives you so many choices that your question should not be, how do I do this in the language, but more, which of these hundreds of options is the best choice for me? It really trusts you as a language. It will give you an incredible amount of flexibility, even to the point where it will let you make the wrong decision.

I've always thought C++ is a language that will let you ride a bicycle without a helmet because that one time that you need to go under a bridge that's exactly a quarter inch over your head, you wont have your helmet knock you off your bike. You do have to worry about a lot of risks, but in the end, you'll have more flexibility than you'll find in most other languages you see.

Finally, I think the most important one is this. C++ makes programing more enjoyable for serious programmers. The two things here are serious programmers and more enjoyable. This is a professional language. It's used in industry everywhere. It's very fast. It's very efficient. It has a bit of a learning curve. It is kind of tricky to get into the language, but once you've got it, the second part, the more enjoyable, is so true.

This language is fun to write things in. Once you've got it down, you will actually step back from your program and say, wow, I just wrote something that took every, single word out of this set that I had [inaudible] contains a specific letter in a single line of code. Or I lettered all the contents of this file into my set in one line of code. Or I just wrote a template that is a multi-dimensional ray of any dimension I want.

Those are not trivial accomplishments, and you can do it with this language. It's fun. It's a great language, and if you think that this sort of philosophy is what you want, where it will trust you, it will really let you make the decisions rather than forcing you into any one paradigm, learn C++. I think you'll love it.

So philosophy. There's lots of it, but I'm not here to teach a philosophy class. I'm here to teach you about C++, so let's get down to some of the details. What does this language look like?

The first thing I want to do is talk about what Genlib.H has. Since day one, you probably have seen things like this. How to include Genlib. Very first line, right here. What is in this Genlib thing? Has anybody actually looked at Genlib before? Actually, pulled up the Genlib file and looked inside? No one? Okay.

If you look inside Genlib, it looks mostly like this. There are basically three important lines you need to know. Time includes string. So we talked to you about the string classes that was a built-in type, like Ent or double. It's a class. It's like any other object, like a vector or a map. You do need to pound include it. We just took care of it for you because since you use it everywhere and it's easy to forget and you can get some pretty nasty compiler errors if you don't, we thought, we'll be nice. We'll do that for you.

The second thing is this error function. You've seen error. Genlib just gives you a prototype for it. The end. But this last one right here using name space STD. Using the standard name space. What on earth is this line? It looks almost like an English sentence. Compiler, make my code work. What's going on here?

Well, I'll show you. Suppose I have this little program back here. Can anybody see this? So I want to pound include [inaudible] and pound include string. If you notice here, I don't have Genlib, and I'm going to make a string and print it out. So the question is, when I say string, my string, what am I referring to?

Well, when this happens, the compiler says, well, go look for something called string. When we pound included string, it put it inside this big bucket of something called name space standard. The idea is that the real name of string is standard string. The real name of C out is standard C out. The real name of endal is standard endal.

It's just so that if you make your own versions of those things, it doesn't have the same name as the standard. It won't conflict, and your program will still compile.

But the problem is that if you try to run this, what will happen is that the compiler says string, looks up here and hits this wall. It says, oops, that's inside the standard name space. I can't go in there, and you'll get this really nasty compiler error, something like, string's not defined. C out's not defined. Endal's not defined.

Now, most of the time when you're programing, you want things like C out and string to actually exist. So rather than saying you have to call it standard string, standard C out, standard endal, you can introduce this little phrase right here, using name space standard.

What it says is take this wall and get rid of it. It's like, Mr. Compiler, tear down this wall, and the result is that this barrier gets a little bit transparent. It's all still included inside of the standard name space, but now it's accessible. So when you go, string goes, oh, yeah, that thing, and actually compiles.

Now most of today, I'm not going to be hitting you with C++ and saying, you must know this. You must know this. There's just not enough time to go over everything, but this little line might be something worth committing to memory because it's the most noticeable change you'll see when you stop using Genlib. If you don't put that line in your code, you'll get lots of compiler errors, and it will just scare you.

So if you put this line in, I'd say 90 percent of your compiler problems will go away. That's that line. That's Genlib, and if you want to take a look, if you right this code, you basically replace all of Genlib. The first header file you'll have to include is this one, CSTDLIB, the C standard library. I'm thinking that when the invented the C programing language, it cost them several million dollars per consonant or vowel they put in their header files. So they put it as small as possible but still readable.

The idea is that you have these two lines. Pound include string using the standard name space. Then right here in this error function, you just print everything to C error, which is like C out. It's just designed for error handling. You call exit minus one, which says, quit this program. Report some error to the operating system and we're done.

This is it. This is all that is contained in Genlib, and the only two important lines are these two right here. Including the string class and using the standard name space. If you do that, and you get [inaudible] Genlib, you're going to be in perfectly good shape.

Now, Genlib is probably one of the easier ones, but what about some of these other headers you've seen? I could go into some detail about all of these, but I don't have time. I really wish I could show you how these work, but I'm going to give you a quick overview today to show you what you'll need to know if you want to get off these libraries.

The first one is [inaudible]. This is the one that gives you string to integer, integer to string, string to reel, convert to uppercase, convert to lowercase, etc. Behind the scenes, most of this stuff is backed by this class called a string stream. You've seen [inaudible] streams, you've seen – they're all streams. You can do less than, you can do greater than.

The point of the string stream, it does the same thing, except that it writes to a string buffer. So you can take some integer, dump it into a string, that's your integer to string function right there. There's a couple other things you can do with it. We use it for doing conversion between strings in other data types like integers. So you might want to look into that if you want to play around with it.

As for convert to upper or lower case, there are these functions, two upper and two lower, that take individual characters and convert them to upper or lower case. Just fore loop over your string. Call that function every time. Presto. Your string is in uppercase. That was pretty straightforward.

If you're interested, in CS106L, we actually wrote some of these functions. If you want to have a working implementation of [inaudible], if you go to the CS106L web site, under the code section, there is a working implementation there. So if you want to keep using these things, just go download that file, take a look at it, play around with it, and it should work. Student:

So if you wanted to set up a coding environment on some Linux machine that doesn't have any of the standard PC or Mac startup files, we can just use that?

Instructor (Keith Schwarz):Yeah, it uses only standard C++, so it will compile everywhere. It works pretty well. I think we only defined a few of them, but you can see how they work, and you can define it for – you can do string to integer, string to double or string to reel pretty easily just be changing a couple types around.

Okay. This is probably the big one you're going to miss. Sympio.H. How many of you have ever played around with raw C++ input functions before? They're kind of nasty. I've always thought this thing called C in, which is the counterpart of C out that does input, is kind of like a rose. It's very sweet. It's very delicate, but the second you break it, it stabs you with a thorn, and it hurts a lot.

Really, you have to think of this as probably one of the most powerful one of the hardest-to-use input libraries out of any programing language you'll see. We gave you this Sympio library just to take care of all these things for you so you don't have to worry about it.

Everybody remember get line? You can use it to get a line out of a file? You can use get line on this C in stream to do input reading. Real line as text uses string stream to convert it to an integer, plus or minus a couple extra things. That's get integer. Again, if you go to the CS106L web site, there is a working implementation of this. If you do want to consider doing C++ beyond this, it actually is a pretty good set of input functions. You'd be surprised how many people don't actually know how to write these things.

Go fire it up. Have some fun, see what they do and that way, you continue using the coding conventions that you've seen but by only using standard C++.

What about random? The random functions in C++ are pretty simple. There's two functions, rand and Crand. Random and C, the randomizer. Again, consonants are expensive, vowels are expensive. Rand gives you random integers in the range zero to rand max. So if you want to get a random double, you want to get a random integer, a random Goulian, just take the number in that range, scale it down to zero one, scale it back up, translate it, etc.

It turns out that this is probably the easiest one to rewrite from scratch. You just have to be a little bit careful of how you do our bounce checking, but it's not so bad. The header file for that is C standard library, but you can probably build this one up from scratch pretty simply.

The one that you actually are going to be missing is this one. Graphics.H and extended graphics. Unfortunately, C++ does not have a standard graphics library. It just doesn't exist. There's a couple reasons. One, it's very hard to standardize graphics. You have to make sure that it's something that works on every, single platform, which is very hard to do. Also, if you're writing C++ code from a microprocessor where you don't have graphics, it would be difficult to support the library.

So there's no standard C++ equivalent, but I bet you this program that I'm using right now is written in C++. It's got some kind of graphics. Most of this is from third-party libraries. I do some Windows programing, so I use – for the Win32 API, which does some graphics stuff. There's a bunch of cross-platform ones like open GL. You can do X Window system. I think Mac's is called Carbon. I might be wrong about that. [Crosstalk]

Instructor (Keith Schwarz):Yeah, there's a lot, and you won't have trouble finding it. You just won't have a standard library that does it for you. So shop around with this. The ones we give you are pretty nice. I might tell you how they work except that I don't know because it's really, really hard. So take a look around, and you'll find some ones that are quite nice.

Everything I've shown you up to this point are simple, like how to read input, how to convert to a string, things that, while they're nice and important, are not going to make or break your program. The thing that is going to make a difference are these ADTs. The vector, the set, the map, the stack, the queue. Those are important.

If you don't have a map, the number of things you can do in a program drops exponentially. It's really impressive how much you need these data structures.

The good news is that C++ not only has a very good support for these data structures, it has probably the best library in any standard programing language that does these things for you. It's called the Standard Template Library, which is the STL.

For every, single container class you have seen in this class, all the ADTs, you will find them in the STL. The names might be a little bit different, the syntax is a little bit weirder, but conceptually, it's the same thing. A vector is a vector, no matter what language you write it in.

The STL on top of that gives you these things called algorithms. I'll talk about this in a little bit. Basically, it's the best thing since sliced bread. I'm not kidding. It's really wonderful and amazing, and I'll talk about that in a little bit.

The STL is one of the major features of the C++ standard library. It's brilliantly designed. It's not necessarily the most intuitive thing, but once you get a handle of it, it's very powerful.

The way you can visualize what this looks like is as this somewhat pyramid structure. At the very bottom, you've got containers. Your data structures, this is your vector, you map, your set. You have iterators on top of that. You've all seen iterators in this class. You've seen map and set iterators.

The STL is basically all about iterators. Every, single container class has iterators. You can iterate over a vector or map or set. Those worked to build these things called algorithms where they're functions that operate on ranges of data.

It's really impressive what you can do with these things. I'll show you a little bit of that later. These container classes, the good news is that the names are pretty similar.

Basically, take the 106 version, make it lowercase, you've got your STL. So big vector turns into little vector. Big map turns into little map, etc.

One exception is grid. There is no STL grid class. It's very easy to make one. All you have to do is take a vector and do some math to wrap a two-dimensional container into a one-dimensional one. In fact, that's how our grid works, so you'll have to build that one. Otherwise, everything's taken care for you, and that's nice.

Again, they have the same fundamental abstractions. You know how to work with a stack, so you know how to use the STL stack with a little extra syntax. The big thing I will say up front, the emphasis in the STL is on speed. In fact, they are designed to be really, really fast. That means their notion of safety is, good one.

So what this means is you'll have to do your own bounce checking. When you're iterating, you'll have to make sure you don't walk off the edge of your container and stuff like that. One of the major reasons we didn't go over the STL in this class is that, which is if you're trying to figure out how to work with a vector, it's really bad if you also have to worry about doing your own bounce checking.

If you're working with a map and you have to use the STL map, you probably wouldn't ever want to use another map for the rest of your life. It's pretty complicated. The point is this is what you'll see in the professional environment because it's a very good library, and it's very fast.

I want to give you a quick example of what this might look like. This is a program I've written using the CS106 vector. I want to go change it to use the STL to show you it's the same thing with a little bit different syntax.

The first thing you have to do is say, the Specter.H we gave you, so you have to change that to the standard vector, which is vector in brackets. Number of characters we have changed so far. Three characters.

Next thing, the big vector becomes little vector. Four characters. The only big difference is this add function. Anyone who isn't in my class want to guess what you would call the function to append something to the end of a vector? Anyone want to take a guess?

Student:[Inaudible].

Instructor (Keith Schwarz):Append? That would be nice. It's actually called push back. There's good reason for this. It means all these different containers have similar function names. Push back, append, potato, potato. Okay. That's probably about ten or eleven character changes.

Then to loop over the thing, it's exactly what you've seen before. There is a size function. There are brackets. You can go and iterate. You can go and access things. The result, as you can see, is not that different. There are a couple subtleties that I didn't go into here about how the STL behaves different from the vector you've seen, but overall, it is the same container.

The other one I want to call your attention to is the STL map. So if you work with the STL, you need to know two containers, and the rest follow from there. If you know vector and you know map, you've got the whole thing down.

The STL map is like our map except that instead of going string to some value, it's any key type you want to any value type you want. So you can map strings to ints, you can map ints to strings, you can map stack of queue of int to vector of double. I don't know why you would, but you have the ability to do so.

The problem is that the interface on map was probably designed by someone who didn't like people. It's really hard to use. Anyone who has used it will tell you that. Once you get the hang of it, though, it's pretty intuitive. The result is you have a very fast map container that is going to be the backbone of any of the programs that you write. It's quite a useful class.

Again, I don't expect you to memorize every, single bit of syntax here, everything I'm saying. Just keep it in the back of the mind if this is what you want to look into. The map you do need to know. Go take a look at that one if you want to pursue C++.

Now iterators. So the iterators you've seen in this class are very nice. They're well-behaved. Has next, next. Has next, next. I'm done. [Inaudible] iterators are designed to look like pointers. You might wonder why anyone would voluntarily make something look like a pointer. There's a very good reason.

This means they're about as smart as a pointer, which means they know nothing. So if you want to iterate over a range, you have to have two different iterators. You have to say start here, stop there, keep walking forward until you get there.

Admittedly, it's more work, but it's very useful because it means that if you want to iterate over a 20-element slice out of a 400 million element container, you can do that. Try doing that with our libraries. You have to go iterate all the way up the start, then keep iterating more.

The other thing you need to know about the iterators is that they're read/write. So you can actually crawl over a container and update the values with an iterator, something that you cannot do with our libraries.

It's a bit more work. The syntax basically looks like pointer reading and writing, but the result is that they're much more powerful, and they're a lot more flexible. Here's a quick example. I just wrote some code here that uses a vector iterator. The thing to take a look at is this right here.

So basically, there's a couple parts to point out. I want to show you this so that if you see this later on, you'll get it. Can everybody see this? The first thing is this begin function. We call it iterator, they call it begin. It just says, give me a start iterator. Since we need two iterators to define a range, we keep going until we hit this iterator called N, which means stop.

To move the iterator forward, has next and next would be too nice, so it's ++, and then to read from an iterator or write to it, you just de-reference it. So you do star iterator, as if it's a pointer, which in many cases, it actually is. That's the gist of what this looks like. There's a few more nuances and subtleties, but if you keep in mind that if you're looking at STL stuff, just treat this like pointers, it will make a lot of sense.

The last thing I want to talk about are algorithms. These things are amazing. The idea is that everything's got iterators, so if you write functions that work on iterators, they will work on any container type. So if you want a binary search or a vector, you call binary search. It's a prewritten function. You want to sort something? Great. Go call sort.

You guys have seen permutations before, right? You have to go do recursion pulsing in the front, permute the rest, stick it back on, that sort of stuff? STL, you can do it in a while loop. You can while you next permutation something, come up with every, single permutation, and you don't have to write a single recursive function ever.

These sorts of things, I think there's 75 different algorithms, anything from permuting to searching to sorting, to transforming to rearranging to sorting to splitting. You name it, there's probably an algorithm that does it. If you play around with the STL, you get to see all this stuff. It's a lot of fun.

I've just been showing you some of the libraries. That's basically all the library changes you need to know. Basically, everything you want to do is still there. The syntax is a little different. Some of the libraries you'll have to write yourself, but it's all there. It's not like you have to relearn everything from scratch.

But the libraries are only as good as the language is, and there are a couple of language features of C++ that we didn't really talk about in this class. I want to go over some of them because you will see them a lot in professional code. After all, you cannot use a library any more than you understand how the language works. So getting a rough understanding of what this will do will make your life easier in the long run.

The first thing I want to talk about is const. You've seen const before in the context of const and my context equals five. Just make a global constant. Saying that const makes global constants is like saying a computer is something that adds numbers. Yes, but they can also run Windows, for example.

Const shows up just about everywhere. It's the little keyword that could. You'd be amazed just how useful this little keyword can be. The biggest thing that it does is helps protect you against your own mistakes. If you're writing a program and you know something shouldn't change, tag it const because that way, later down the lie, if you do accidentally change it, it says, that's a mistake. Compiler error. You screwed up.

It protects you. That should've been a double equal, not a single equal, those sorts of things.

You've all probably seen that the C++ compiler doesn't treat anything as sacred. Like, oh, you don't want to return something? Or something like, oh, wow, you did single equal instead of double equal. You're the boss. The second you try to break const, it pulls out its big ruler of judgment and whacks you on the wrist with it, like shame on your for breaking this constants.

It is the compiler trying to help you out. Please know about this keyword. I'll give you an example. See this function prototype? Void do something takes a vector by reference. So I've passed my vector into this function, and the question is, what's it going to hold when it comes back? Well, it could be the same. It could be completely empty. There could be 137 copies of the number 137. You don't know. In this class, we've shown you, pass by reference can be either for efficiency reasons because it's not fast enough, or it can be because you want to actually change the thing.

It's difficult to see what this function does because you have no idea which of those two it is. The idea, though, and this is something that you will see in professional code everywhere, is doing something like this. If I say, void do something const vector and my vector, that says that this vector can't be modified within the function. It's like handing this thing off saying, I'm giving you this value for efficiency reasons. Don’t try to modify it. In fact, you can't modify it.

It makes your code self-documenting. Someone looking at your functions can go, that's const. I can't possibly change anything there, and they'll be totally fine handing off the string containing their graduate thesis to it, for example. If you had your graduate thesis stored as a string, which I don't know why you would ever do this, but if you see a function take the string and parameter, you'd probably be a little bit worried. Like, is it going to replace my graduate thesis with, hi, I don't have a graduate thesis or what? The const is useful.

The question is, if it's such a useful keyword, why didn't we show you this? The biggest reason is you can't just use const every now and then. You can't say, I'm going to mark this parameter const. I'm gong to say that one's const right there. It doesn't work that way. Let's say I do mark something const. That object has to know how to behave when it can't modify itself. So you have to make the entire class written const correct. Then if it has any data members that are classes, that class has to be const correct.

You get this exploding effect where you stick a singe const in, and suddenly every, single piece of your code is marked const. That's a good thing because it makes your code self-documenting, but the point is it's an unnecessary language complication. When you're trying to write the PQ class, you should be worried, how do I build a chunk list, not oh, is this function const?

So in the professional world, you will see it. Here we thought, it makes things too complicated. We'll give it a pass for right now.

Another big one. Object copying an assignment. You've seen disallow copy. It says, don't copy this object. It's not standard C++, but I've been told that Google actually has a macro that does something just like this, so you're programing the same way Google people do.

The difference is that if you're trying to write a PQ and it can't copy itself, it's a little bit frustrating. Think about it. You make some PQ. You can make a vector copy. You can make a map copy. If you can't make a PQ copy, you'll be wondering, why not?

So it turns out C++ lets you redefine the way copying works. So these two functions here called copy constructors and assignment operators. While it's very useful to know how to do this, there's a major reason we didn't tell you about it in this class. It's because it's really hard. I think I spent, in 106L, an entire week going over this. We don't have a week to tell you how to do this.

Here's a list of some of the things you have to keep in mind when writing these functions. [Inaudible] to clean up your own memory, not memory you don't own. To clean up the memory, to copy an object, to copy the object correctly, to handle these [inaudible], to follow the same rules that C++ syntax dictates. There's an awful lot of stuff going on there.

To expect you to get all this right when you're worrying about your PQ is just too much. It really is. Imagine it's the day before it's due. You've got the PQ working beautifully, and it doesn't copy right. What does that tell you? You've written your copy function wrong, but you totally understand and made your point, which is how to build a priority queue. This is a chunk list. This is a heap. This is an unsorted vector. That's what's actually important, not building stuff like this.

When you're in the professional world, you do need to do some extra work to make your objects copy correctly. So you should look into these functions a little bit. Again, to stress, the stuff you've learned in here, the actual here's how you build these data structures, is more important than these global syntactic nuances that you've got to be aware of.

One more thing. This is pretty cool. I have this code snippet right here. I make a string, and I say, my string is equal to this is, and then I tack onto it, a string. I'm going to iterate over this entire string, and every step, I'm going to print out the current character. So this is just print out the string one letter at a time.

I’m going to highlight a few things. Why is it legal to [inaudible] equals a string with something else? Why string but not priority queue? Why are we allow to do this? It's a string. It's an object, and we just code plus equals on it. Why does it let you do less than, less than? What does that mean? Why do streams let you do that but nothing else? Why can I read a string with brackets even though it's really an object?

At a high level, we all know what it means to add something to a string. It means take something and stick it on. You know what this less than, less than. It means put into a stream. [Inaudible] C++, I want to define these operators for my classes. It's a technique known as operator overloading.

Basically, all you do is write functions that are called operator and whatever the name of your operator is. So operator equals, operator brackets, operator less than, less than. Operator plus, plus. It's just a syntax convenience. You can write code that looks more intuitive that does something complex behind the scenes.

For example, if I write my vector bracket I, it's not like, oh, it's a bracket. It's much faster. It really means, call the function called operator brackets. Again, it's a convenience and nothing else. If you treat it as anything more than a convenience, you can kind of hurt yourself with it.

You can overload basically every operator in C++. You can overload things like bit wise [inaudible] with assignment. You can overload the exore operator. You can overload parenthesis, and, comma, all these operators that most people never use.

The ones that you see most frequently, though, are these ones. Overloading operator less than, less than for stream insertion. You can actually make it so that you can make a vector class than can print itself out to the screen. It's kind of useful.

The assignment operator, operator equals, the less than operator for comparing things and the parentheses operator. You can actually overload parentheses. It's pretty cool stuff.

It's actually for something called functors, which is really awesome. If you do play around with C++, this is definitely something to keep in mind because it will make your life a lot more enjoyable.

I think more than any other language features, C++, except possibly multiple inherence, this is the most widely-criticized feature of the language. The reason is that you can make things that make no sense legal. Like, defining the modular operator for two PQ. My PQ, my other PQ.

Anybody think of a sensible interpretation of what it means to mod one PQ by another? If you do, please send me an email or shout it out right now. I can't think of one. But you can make it legal. I don't know why you would, but you can. The point of this is, if you think about it, call it the philosophy. Let the programmer make the choice, even if it lets them choose wrong.

You can do this. You probably shouldn't do it, but by giving you the opportunity to do so, it means that when you really do need to write a class that has a module that's defined, you can do it. It's flexibility. It means that you have to make sure that what you're doing makes sense, but it's flexibility.

You've all just finished Pathfinder, right? Did anybody templatize the [inaudible] PQ, by any chance? Anyone? You guys are smart. Here's the thing. Let's say I wrote something that says, A equals B. What does this mean? It says assign B A.

What if I write this in a template function? I have a template of some unknown type. I'll just say A equals B. What's it mean? Well, if it's an int, it means take the int and copy it. If it's a floating point number, it says take the floating point number and copy it. If it's a string, it says make a string deep copy.

The point is that every, single time you use this same syntax, A equals B, but the result is different, and it's always the correct result. This is what you can get with operator [inaudible], the ability to take code and make it look right for every, single class so that in templates, you can just call the function, and you're guaranteed it will work.

That's a quick tour of some of the language features. I know this might seem like an awful lot. I've been going very quickly through it, but want to conclude by saying so what? I just harangued you with a lot of language features, a lot of syntax, a lot of libraries. What's the point?

The point to take out of this is that C++ is big, but it's also a very powerful language, and it's a very expressive language. I think it's a very beautiful language. The features that you've learned in this class will go anywhere you want them to. If you want to take your coding skills and apply them to C++, you can, and you have this flexibility. You really have this gift.

What you've learned in here is something most people never learn. It's how to make a computer solve a problem for you. That's a skill that's going to follow you for the rest of your life, no matter where you take it.

If you're interested in C++, I have a huge number of references here, if you want to go see them. The point is that if you want to solve a problem with a computer, you need three things. First, programing skills. You all have that. You need a good idea. I don't think I have any good ideas for programing. If I did, I'd probably do them myself, but if you have a good idea, and you want to make $1 billion with it, the last thing you need is a programing language.

Hopefully this quick tour of C++ has showed you. This is what this language is. It's a tool for solving real problems that trusts you and that means that overall, you'll have fun doing it. So if you want to go and run with this, if you want to say, I'm a good programmer. I can do this and learn C++. I think you will love it. I think you will enjoy it.

I think it will be some of the most fun you will have sitting in front of a computer. So have fun with this stuff. That's the whole point. If you're in this class, I hope you enjoy it. I hope you said, yeah, I can make a computer solve something. I can do graph algorithms. I can do graph [inaudible]. I can make data structures. I can make a computer program that plays Boggle better than I ever could or anyone I know ever can. Have fun. That's the whole point of this.

If you want to do it in C++, come talk to me. I will give you some references. Enjoy. Good luck on the final, and I will probably see you around on Friday for the final. Enjoy.

[End of Audio]

Duration: 42 minutes