ProgrammingAbstractions-Lecture01

Instructor (Julie Zelenski): Hi. Welcome to CS106B, programming [inaudible]. The website was probably the most important thing to take away from here, right, is where can you find information about the class? We're going to talk today and give some overview and stuff like that, but this is kind of the home base for all the material. If you managed to get the handouts on the way in, you're golden. Otherwise, you can grab them from the website. There's a lot of background information, staff information, office hours, all sorts of stuff gathered there. It's kind of one stop shopping for figuring things out about our course.

Let me tell you what I'm going to do today. The first day, I know a lot of you are shopping and trying to figure out what's the right fit for you, so hopefully today, I'm going to try to give you some information that will help you make a good decision. I want to tell you a little bit of what our course is about, tell you about the administration and logistics – most of that stuff is pretty ordinary, and you can read about it. Of course, I have to do a little bit of marketing. I get to give you my unbiased opinions of why this is the absolute best class you could possibly take.

I get paid per student, you know. That's not true. Maybe we'll even have time to check out a little bit of the C++ language before we're done. Let me tell you about CS106. CS106 is the introductory programming sequence here at Stanford. It's our version of CS1, where you start at the beginning when you are interested in learning more about programming. We have a two-quarter sequence, A and B, that kind of follow together. They're not particularly tightly coupled, which is to say if you took A and you took a break, you could come back to B.

We also have an alternate form of B, the CS106X, which is kind of an honors version of that second course. So after you've taken that first course and you're feeling really jazzed up, there is an alternative more intense way to get through the second course that's offered. It's offered this quarter, in fact, so if you're thinking about that, that is an option for you as well as sticking with us.

What do we do in 106A? The official title is programming methodology. It is starting at the very beginning and assuming you have no background in programming. It's teaching you how it works, what the languages look like, what the syntax is like, what things you need to know about how to solve problems using a computer. It covers a lot of the fundamentals about logic control and in general, I think the big issues of 106A have less to do with any particular syntax or feature that you learn about a language but about how do you solve problems on a computer?

Somebody gives you a specification of you need to write a program that does X, and you have to figure out how to make that happen, how to break it down, how to step through it, how to develop it, how to test it, how to iterate on it, how to make changes in it later, how to debug it when it's not working well, and those things, I think, transcend any particular language. W e happen to use the Java programming language in our 106A course because

it's a great tool for introductory programming, but I don't think of that as being really critical.

In fact, if your introductory course was in some other fairly modern, high level language – let's say you learned in Python, C, Scheme, or something other than Java, you're probably still fine, 'cause what we're really counting on in B is that you know how to program and think like a computer scientist. Not a lot of the details of the syntax are going to be important to us. B picks up from there, saying okay, you've got some fundamentals. Let's start really learning some of the techniques that extend the range of problems you can solve.

We look at recursion, which is one of the fundamental problem solving techniques that involves using something akin to mathematical induction to solve problems in terms of themselves. Looking at a lot of algorithms for sorting and searching and hashing and doing things efficiently, knowing how to compare and contrast alternatives in algorithms, having some formalisms by which to discuss those things, and learn some of the classics that are out there for solving these problems.

The dynamic data structure, which involves use of pointers to construct things like lists, trees and heaps, graphs that model certain structures that are very effective in solving certain kinds of problems – we'll work through those. A large part of our time is spent on this concept of data extraction, which is why extraction shows up as the main word in our title. As we start to solve more and more complex problems, the code itself can kind of become overwhelming if we don't have some technique for managing that complexity.

One of the big themes for 106B is how we can use this idea of extraction, building something and dealing with the low level details but then closing up the box and treating it as kind of a fixed entity and building on top of that and then closing another layer around that as a way of working on something, finishing it and moving on to some larger piece without having those details cloud our way. It's a very powerful technique for solving larger problems.

In that context, we'll be looking at some of the classic data structures, like stacks, cubes, lists, maps and sets as part of the domain for that. We do happen to use the C++ programming language, but this is not a C++ course, so to be clear about what you're getting versus what you wanted, we use Java here. We use C++ here. We happen to think they're good reasons to actually expose you to both languages. In particular, C++ is an enormous language. It has a lot of language features as well as a very large standard library, and our goal is not at all to turn you into this industrial strength knows every detail about quirks and ins and outs of C++.

There is another class, 193D, that does attempt to do that. In fact, if that's what you're looking for, I suggest you take a look at that. What we're here about is learning advanced programming techniques – taking those foundations and building on them to be able to solve more interesting problems. We happen to use C++. You will learn some C++, but I almost consider it a side effect of what we're doing.

Just a little note on placement – if you're kind of in between and not really sure, these are the very rough guidelines, but they give you some idea of which groups gravitate where. If you are new to programming or you're not confident about your background – maybe it was a long time ago. Maybe it was self-taught. Maybe it was in a course that you felt was not as good as it could have been or you didn't do as well in it, 106A is a great place to start. It actually is by all accounts an extremely popular course at Stanford and services a wide group of people with a little bit of background or no background.

If you do have something like a solid first course experience – you did well in 106A or took a similar course or perhaps even self-taught your way through a lot of those materials and you feel ready to move on, 106B. An AP course in high school – the A curriculum is a pretty good match for 106A here, so you're in a great place.

If you have this and you've got a little bit more going for you or you're super enthused and you have a lot of extra time this quarter and want to sit in the company of only the uber geeks, you can check out 106X, which covers the same kind of topical ground but at a different level intensity. It amps up a little bit of the assignments, covers some of the material that we won't get a chance to cover, and just pushes the envelope a little bit there.

If you have experience comparable to the first two courses – you've done all the things that we're talking about here in B and you feel comfortable with it, it might be that the right place for you is 107, which is the third course in our sequence. That's somewhat rare, so if you're thinking about that, I encourage you to talk to me a little bit to make sure that you won't be missing out on something important in doing so, but certainly there are students who have – for example, the APCSAD curriculum is pretty comparable to this course here, and so depending on how high quality the course you had was, it might very well be that 107 is right.

In some situations where that course was a little bit lacking, there may be some ways that we can help reinforce the things you've learned and build a stronger foundation to move forward from rather than jumping ahead. Any questions about placement?

Let's talk philosophy. I think there's a statement about what we officially are, but I also think that 106 has a long tradition at Stanford that comes back from student motivation, which is interesting. I was here as an undergrad in the 80s when the 106s were just getting off the ground, and at the time, Stanford only had a graduate computer science department, and the belief in the ancient period for computer science was you should get a math degree, and only then would you be mature enough to learn about computers.

There was a groundswell of Stanford students who said we want access to programming. We want it. Part of the 106 was a really careful thought about what the 106 would be a Stanford and what we want them to be in a philosophical sense. One is that we welcome students of all majors and backgrounds. We don't have a version of 106 that's for the majors or potential majors and a version that's for the non-majors and a version that's for

terminals. We really think that we can bring you all together and design a course that addresses this wide disparate group but still serves it well.

I'm going to turn you into a CS major. That's my plan. At Stanford right now, not having to make that choice about a major until junior year is a gift to allow you to explore and to feel unencumbered by having made some decision when you applied, and I think it's important to respect that gift that Stanford gave you by trying to make sure our courses don't funnel you one way or the other before you figure it out. You are all welcome here. We try to make it accessible to everyone. We have certain plans that help to make that work.

We do try to provide a solid, practical foundation in programming that given our placement at Stanford in the middle of the Silicon Valley, there's kind of a strong influence for us to try to produce students who from the get go are learning things that are actually quite useful outside of the classroom rather than teach you a very academic and mathematical language like Scheme that is very rarely used outside of the classroom. We're trying to teach you on the tools, languages and techniques that are actually in active practice.

We are using Java and C++, two of the most prevalent languages out in the industry, and we do a lot of learn by doing. We assign challenging, full-fledged programs that you work on and you build, and so it's not designed to be academic exercises. You really are building skills that have applicability here and outside of the class. We have a big emphasis on truth and beauty. This is one area in which some of the substitute courses that we have seen students come in with have a little bit more trouble with is tackling this part of it, which is that there are a lot of ways you can get a program to work. Many of them are not pretty.

You can just type and type and type and eventually, you can get your way to something that works. That, in the end, might produce a program that from external appearances works. It plays hangman or whatever was the desired goal, but that internally is a mess. It's not well structured. It's not well [inaudible]. It's not easy to understand. It wouldn't be easy to modify. It makes a lot of decision that are really sub optimal, and we're really interested in producing engineers that have a good sense of design and really appreciate what is involved in writing good, well designed software, not just working software.

We will be giving you feedback on both the correctness and functionality of your code but as important if not more so, also on how well we think you did at designing and implementing and writing code that is of a high quality. We make a big deal out of that. That is something that is not always shared by other classes, and so in particular, someone who is self taught or in a class that didn't emphasize this might feel that there's a little bit of a gap there where we need to make that up with you, and we can work with you.

This is a very individual thing, because there's not one good example of the perfect style in the way that a lot of different people express themselves in written communication

very well but differently. The same thing is true about programs. You will have your own unique style, and we'll work with you to coach you on getting your style to come through and be beautiful and elegant.

This kind of comes back to point one as well is that we make heavy use of undergraduate section leaders as mentors in this program. We have a staff of 50 or so undergraduates who work with the 106A, B and X courses as a team. They have specific responsibilities with their section, so mentoring and grading and meeting with sections to give individualized feedback on their programs as well as answering questions, solving problems, being in the lair 30 plus hours.

They're most weekday nights about six hours and often well past the midnight when it's supposed to cut off solving people's problems, helping when you get stuck, answering your questions and making sure you're all making forward progress. That comes back to point one. In some universities where they're choosing to use their intro course as a weeder, it's like let's separate the weak from the [inaudible] early and let's make it really hard and not provide too much support, and that way we'll make sure we get the people we want.

We have a different idea. Programming is hard, especially when you're learning. There's a lot of complexity to master, and there are a lot of details that can interfere with moving forward, and we don't want you to get stuck on something that we can very easily resolve for you. So make sure you have accessible staff members in person, in email and regularly in session to help get through the roadblocks and keep you moving forward.

What do you need to know to be sure you're going to do well? There are people who think to be a good computer programmer, you need to be good at math, logic, and drinking Jolt. I think it comes down to more personality traits than any particular technique or skill. You don't need to be good at math. How much calculus and trig shows up in this? Not very much. A little bit of logic – that helps. I think it comes down to traits like curiosity and determination and hard work. Starting early, asking questions when you don't understand something, trying to solve the problem by logic – why does this case work and why does that case not work?

I think these are the skills that serve you the best in this class and probably every other thing you'd want to tackle. There is a lot of time that will be spent here to master this. I can talk about programming all I want, and you can go oh, yeah, that makes a lot of sense. But when you go to write it yourself, it's a very different experience. That's where being focused and staying on task and getting help when you get stuck can help you move through that.

My unbiased opinion about why 106B is one of the best courses at Stanford – it's going to be totally obvious when I say these things, and you guys are going to have to go along with it. I'm actually a big time geek, and I happen to love programming. That's why I'm a perfect fit for teaching this course. I have taught 106 B or X more than I've taught any other class that I've taught at Stanford in my time here, and that's because each quarter

when we're setting the schedule, I say give me B. Give me X. There's no better course to teach.

Programming is just awesome. If you love programming, I think there's almost nothing better to do in the world. You have this task. You're trying to get there. You're coding. You're making stuff happen. You're testing, iterating and running. You see stuff. You build things. When you're done, you know it works. When a program works, you know it.

It does what it's supposed to do. It gets the right answer. It plays the game. It solves the problem. Finding and fixing that last bug – although debugging is one of the last aspects of programming a lot of people bemoan about, I happen to think that if you are driven by debugging, that is one of the most awesome detective stories ever.

Trying to figure out why it happened when you did this, why it went this way when you moved that and what this effect caused and understanding once you make the fix how it fixes it – staying up late. I have stayed up late more nights debugging than anything else in my life, and I'm not sad about that at all. That means I'm in the right place. Hopefully, some of that resonates with you. If that sounds really awful to you, hopefully we can change your mind a bit. That is, in some ways, part of what drives computer science is this wanting to build things.

We are engineers. We have this computer science name. Just remember – any subject that's name is something science is not a science. We're trying to puff ourselves up a bit. We do a lot of really great work, and there are a lot of neat scientific principles and theories that underpin what we do, but in the end, I think what drives a lot of us is just the engineering – building stuff that is really neat.

It kind of reminds me that my husband's a mechanical engineer, and so I used to be envious, because he would always build things. He's always toting foam core around the campus on their paper bicycles and stuff, and it's like they build all these things. Now, having watched all the things he builds, it's like [inaudible] really hard. You need all these materials. You need all these tools. In computer science, you don't need anything. You need a compiler and you need a computer. You need your thought. It's like an abstraction we built out of our brains.

There's this relatively small set of things that you need to master and then you can combine them in these very sophisticated and interesting ways to solve all sorts of problems. There's a very low overhead, and the range of things you can attack with the same set of skills is huge. Every domain out there can benefit from somebody applying computers in a useful way without fail. There's all sorts of problems where technology is part of the answer – not the only answer, but certainly something that you can take whatever interest you have and combine it with computer science and construct something cool.

I happen to think that what happens in the second course is amazing. The first course, you've kind of got to get up to speed, and there's a lot of basic material that needs to get covered, and it does set you on the right path, but in this course, we really get to blossom beyond the basic things.

There are a bunch of really neat and very accessible techniques that a second quarter student can understand and master and do really cool things with. You can learn how to do something like create a database that has a million entries and then ask for somebody by name and be able to instantaneously be able to find the name. Change the size of it. Make it ten million, a billion, and still be able to provide that kind of instantaneous access. You're going to learn how to do that.

The technique is not some superhuman thing. It's something very clever, admittedly, but it's very accessible. Taking that same million thing and learning how to sort it efficiently. What if you happen to know things about how it's almost sorted but just a little but out of sorts? Are there ways you can actually make it even faster to put it in sorted order? There are techniques, for example, like recursion that take on problems that you might not have any idea when you first look at the problem how to solve, but once you've got your head around recursion, you can look at that and say I can write a five line function that will solve that problem.

This is the kind of power we're going to give you with our quarter together. There's a bunch of really amazing theoretical and algorithmic stuff to explore that really increases the kind of things you can do with programming. I'm particularly fond of it. As always, I think the section leading program that we have created and built at Stanford is a huge part of what makes our 106 courses so successful, and so having somebody who's individually working with you, meeting with you weekly and giving you that feedback that's individualized and personalized for you and helping you get through the rough spots is a lot of what helps to make the experience very fun and very personal, too.

What section leaders do I have here? Not a one. Where are those section leaders? We haven't yet identified who's going to what class, so maybe they're all thinking they're going to go somewhere else, but they're wrong. We'll get some. 106B – great. You agree? Have I convinced you? Is anybody still hesitant? Is it a lot of work? Oh, no.

Let me tell you a little bit about logistics. There are some random things you may want to know about how the class works. We're going to meet here Monday, Wednesday, Friday 2:50 to 3:05. It looks like we almost exactly fit in our classroom, which means we're all going to be very friendly and cozy. The lectures are being taped and are available online, and so that has the neat side effect that you can watch them and review them later. You could watch them lots of times.

It also means that if it were pouring rain and you were sitting at home in your bunny slippers, you could just say hey, I'm not going outside and you could turn on your computer and watch. I'm a big fan of having you come in person. That's because I don't want to lecture to an empty room. I also think there's an interaction there that I'm fond

of, and so I hope that you will make every effort to attend in person as much as you can. It is nice to know that if you do miss a lecture or you get caught up with something, you'll have a chance to review it later online.

We will have sections that will meet once a week, just like 106A. The section leader who meets with you is the one who will be grading your programs and sitting with you and doing the conferences. What you need to do to get yourself into a section – there are several section times listed. Ignore them.

What you do is you go to the online – if you go to our class webpage, there's a link that tells you how to sign up for a section. The section times aren't up right now, but they will go up tomorrow, and they'll be up through the weekend. You go in and put in your preferences about what times fit your schedule or not, and there's this big computer program that gets everybody into a section that fits into their schedule.

The signups will be up from Thursday at 5:00 until Sunday at 5:00. If by Sunday at 5:00 you've got your schedule fixed, then you're fine. If you happen to change after Sunday at 5:00 and after the assignments have been made, at that point, it's a little bit harder. We can make adjustments, but it's on a very case-by-case basis. The best thing you can do is by Sunday at 5:00 have a pretty firm idea about what you can and can't do. There's a list of preferences, so maybe what you can do is pick things that you know will work no matter what happens.

The workload – everybody wants to know how much work. It's a five-unit class, and it's a five unit engineering class. You get your five units worth, I would say, so don't worry about that. I won't shortchange you. We have programming assignments not quite weekly. I think there are seven of them across ten weeks, so you can calculate it out. It's about a week and a third for any particular one. The students report that it's about a 15-20 hour project, each of them. Some people get them done in less than that. Some people take a little bit more. I would say that's kind of the mode range for what people are reporting.

I do think that of the people who report less, some of those are people who are naturally gifted, but a lot of it has to do with your habits about how you work and how you make progress, so if you are one of the people who feels you might be more likely to be on the other end, you can come and talk to me and I can give you some suggestions.

Choosing to work in the lair where the helpers are on duty has really positive effects in that when you get stuck, you have easy access to somebody helping you work through it rather than spending an hour or two fighting something that turns out to be simple but required knowing something that you didn't yet know.

I'm a big fan of learning things yourself. There's also a time when a well-placed bit of advice from somebody can save you a lot of time. There will be a midterm and a final exam. They'll both be in class, open book, open notes. The midterm is actually technically out of class. We're going to have it at night because we need more than a 50-

minute period to get any coverage of that. Our final exam is scheduled in our university-scheduled slot. Sadly, that is at the very end of the exam week, but that's when the registrar put us, and that's when we need to go. You may want to take a look at that before you head off for spring break.

Unfortunately, our publisher will not allow me to distribute the course reader electronically. They're not exactly very in the modern age on this. Yeah, I'm working on it is the truth. By the time the world sees this, hopefully, we will have some other strategy. We currently are in negotiations. The nice thing about the course reader is that we have not changed it in the last year. If you know somebody who has it from last fall or last spring, it has some minor edits and typos that were fixed, so if you can get a hold of an old one, it's good.

We are getting no royalties on it. We're publishing it at production cost, so where it would have been a $100.00 textbook had it been bound and snappy, you're just getting what it cost to photocopy and bind the thing, and Eric and I are eating ramen. It's hopefully cheap enough that you can find a way to get to one or get an old one without it being too much of an obstacle.

People in general find the course reader to be an asset. We do say it's required reading. It does have a lot of material that's very useful in understanding the course. There are other people who don't make as big a use of it, but there are some sections that are really very valuable and other ones that may be more or less depending on your learning style. There's also a lot of good sample problems and review questions in it that help to test your understanding.

You know, it's tricky, because the university in general discourages you from giving alternate exams because of – you can imagine the issues of having an exam that has been seen by some number of students before some other number of students take it. Even though we're all bound by the honor code, it does create a situation where there is some temptation. It's possible it could be a little bit early without a lot of gap, but I don't think early enough to make a lot of help is the truth. We can talk about it. The current plan is not, I would say.

A little bit about compilers. We use C++ and we also use some custom libraries, which limits us to distributing on a certain number of platforms we've had a good chance to test on and work with. The compilers that we have support for is X Code on the Macintosh. Anybody who has Mac OS10 can freely download that and install it.

We're using Microsoft's Visual Studio Version 2005 on Windows, and we have an arrangement with Microsoft where they have distributed the software free of charge to students, so if you would like to install that on your own Windows computer, we'll give you some handouts on Friday that tell you what to do to get the compiler and get it installed on your computer.

Our cluster computers in the dorms and the libraries and the lair have both the compilers and libraries installed, so if you work in a cluster, you don't have to do anything special. You just walk up and it's got the stuff ready to go. I'm a Mac person. I've been a Mac person forever. I can't get over the fact that you go to the start menu to shut the machine down. That makes Windows impossible for me to use. I would say campus wide, there are more Windows machines than Macs on campus, so if you want to take a popular vote, you could do that. If you want to be on the side of the Mac bigot, you can come and be with me in X Code.

I've got ten minutes to tell you a little bit about C++. That is the next journey that we're going to go on together. The first question is why are we doing this to you? I just got comfortable with Java and now you're telling me to throw away my Java and start over. Let's generate a little bit of love for C++. The advantages of early multilingualism – I have two small children at home that are two and four, and I read a lot about bilingualism. It's very clear that for natural languages, when you're acquiring a language at those young ages that that is the best time to introduce a second or third language.

It's been looked at in terms of programming languages as well, that when you are learning a programming language, there are certain kind of ruts your mind gets into about the way a language is that is based on your early experiences. If you spend a very long time working only in one language, those ruts get deeper, and you have a certain way of thinking. You're a little bit stuck in that paradigm and its approach. What's easy to do in that language, what's hard to do in that language tends to make a stronger impression on you in a way that makes it harder as you grow and explore the languages to kind of get out of those ruts and shake yourself out of it.

There's been some pretty good evidence that somewhere between one and two is a good time to think about branching out and starting to think about different ways of doing stuff and seeing some different syntax and some different ideas to help build in the flexibility from an early age in your career to buy you some strength later. That's part of what we're doing.

Another part of it is actually that a lot of our upper division courses rely on a knowledge of C and C++, that family of languages, and that the longer we postpone it, the more painful it becomes. In the later courses where you're learning about compilers or graphics or networking, they don't have the time in those classes to stop and teach you C or C++. They need you to know C++ to get the work done. Moving the foundation into a programming class seems to make the most sense in the context of our curriculum.

We do switch you over here. The good news is that it's not as big a change as it might sound at first glance. Java is actually highly derivative of C++ for a start. They're kind of cousins in the scheme of language design. They have a lot of syntax. How much C++ do you need to know to start? The answer is zero. You don't even need to know what the word means. In fact, you actually probably know a surprising amount about C++ just by virtue of what you already know just translated a little bit.

Things like the four loop of Java or the way you declare variables or the way parameters are passed into a function is exhibited in Java in very much the same way it is in C++. There are a bunch of things you already know and you don't even realize. It is not assumed that you know C or C++. If you happen to already know those things, you're ahead of the game. If you have not, then not to worry.

How much C++ are you going to learn? We will spend the first three or four lectures just talking about how things get expressed differently in C++ and mostly talking about the differences in the libraries. The syntax itself is quite similar. Some of the more extensive changes have to do with how the C++ string is operated on versus the Java string.

How you do file input and output reading in C++ is a little bit different than the way it's done in Java. We'll spend some time saying here are some things you know how to express in one language. We're going to teach you how to express them in another language. It's just mapping from your previous vocabulary onto a new one.

Along the way, we will actually introduce some of the C++ features that we need to support our pedagogical goals. We'll be talking about how classes get designed in C++. We'll see how to use classes. We'll see how to define those classes, and we'll look at things like templates, which is the C++ construct for doing generics that you seen in Java – how you can build containers that are type unspecific and things like that.

We will learn a little bit of some of the fancier features of C++ like the pass by reference parameter, but there's actually a very large amount of C++ that's just off the table for us. We will not make extensive use of the standard template library or the fancier features of static and [inaudible] and a bunch of key words that mean nothing to you and should mean nothing to you. You will learn enough to have reading familiarity with C++ and to be able to express yourself quite well in the subset we're using, but it is a subset of C++ that you're being exposed to.

If you find yourself really wanting to master C++, we are offering CS106L. CS106L is a lab companion course that is open to students enrolled in 106B or 106X. It meets twice a week. It's actually on Monday and Wednesday late afternoon. It's 4:15 in Hewlett 103.

It's being taught by a veteran section leader who is very well versed in C++ and who of his own volition volunteered and created this course because he himself was a little frustrated as a prior 106 student in wanting to get at some of those C++ things that weren't fitting with our goals. It is a place where you can get more exposure to standard C++, do some exercises that help you to test out those things and see how those things are expressed and get another unit. It's a pass/fail lab course.

You can also just attend or grab the materials if you just want to look at them. It's a great way to broaden that knowledge of C++ beyond what's useful for us in terms of our goals. I'm going to ask you some questions, because I don't get to do all the talking. How much C++ do you need to know? Some of you may know nothing about it, and that is perfectly fine. Some of you probably know something about it or at least have heard something

about it. I'm going to have you guys tell me what it is that people tell you about C++ that makes you either excited to learn it or frightened to learn it or interested in how it works.

Student: It's what Java is based off of.

Instructor (Julie Zelenski): It is what Java is based off of. That should be a little reassuring that there is a syntax there that got adopted with some minor changes. It should feel more familiar than different when you look at it. It was very strongly influenced by a generation of programmers who C++ was their native language who designed the Java language. That's a good thing to know. What else do you know about C++?

Student: It's extended C.

Instructor (Julie Zelenski): Yes. It's extended C. Here's how it fits in the spectrum. C is kind of a 1970s creation. C++ is a decade later. C is the language which it is based on. It is an extended C. It's called a superset. Everything that compiles and works in C still exists in C++, but then they added a bunch of features. Not only did they add a bunch of features but they tried to fix some of the things about C++ by replacing existing things.

For example, there's some string handling in C that's kind of very primitive. They added a string object with much cleaner handling and safer semantics into the C++, but they kind of left the old one around. Some parts of C++ feel a little strange because of this history to it – the legacy of incorporating everything C was plus the stuff means at times there's a little bit of weirdness there. It also means that the language, as a result, is very large. C's safety and runtime features were extended by what got added in C++.

Student: A friend of mine told me that [inaudible].

Instructor (Julie Zelenski): That's a good thing to know. C++ might be a little bit more dangerous than Java. That is true. Java is very concerned with safety, in particular since Java was designed for web delivery of content. It was very important that the program have very constrained features on what it can and can't do, and so as a result, Java tends to be very parental. When you forget to initialize a variable or forget to return from a function, Java's very aggressive about saying hey, you've got to fix this.

C++ is a little less parental. Here's the overly protective mom, helicopter mom, the ones who stand. That's Java. Java's making sure – oh, are you okay, sweetie? Let me stand here in case you fall. C++ is crack mom. She's like yeah, I'm over here with my friends. Don't play with the kitchen knives. It's a professional's tool, and professionals don't want to feel encumbered. There are certain things they want to do that require some of this low-level access, and safety usually comes at a cost.

Any sort of feature where the language is double-checking for you is taking time and efficiency. There's a cost associated with that. Every time you want to get something out of an array, it's checked to make sure that that number was not off either end. Every array

access costs you a little bit more. C++ says I'm not going to charge everybody that penalty. If you actually have the bad sense to write a program that does those things, you deserve to be punished. As a result, you will at some point in this quarter get to experience some of that firsthand.

Some of it is a growth experience. Some of it can be frustrating. It is part of what professional tools often look like. They are making these tradeoffs of efficiency over safety that put more of the work back on you as the programmer to be a little more attentive on those things. You can write programs that crash in very spectacular ways much more interesting and varied and dangerous than the kind of things you can do in Java. Good to know.

Instructor (Julie Zelenski): You want to use pointers. You're going to get to use pointers. Pointers are really neat, and they're also very challenging. Pointers are these ways of building these very flexible and amazing data structures – the kind of things that we're going to try to build. At some point, they are going to be the only way to achieve those things well, so building these things called trees, graphs and lifts rely on understanding a mastery of the pointer type.

The pointer type is complicated, and it's part of that danger thing, which is having access to rearranging memory by virtue of addresses opens up a lot of opportunity for there to be mistakes. Mistakes can be made in the passive voice that have consequences. You'll get to experience firsthand what that's like. There's joy in it, because getting it right is awesome, and there are things that you can achieve that are really extraordinary with pointers, but when it's not working, it can be frustrating. You're getting a little bit of both.

It turns out C++ does not have a graphics system built into it. Java is actually distinguished from previous languages. Java tries to solve all problems. Traditionally, a programming language tends to have a set of libraries that have facilities for data management, reading and writing files and sometimes some networking, but they don't tend to actually solve application layer problems. That tended to be a different piece of technology. The Mac OS might offer a graphics library that was written in C++, but C++ the language didn't have a windowing system or graphics system.

C++ does not have those features itself, so any C++ compiler you get comes with these basic things about handling files and managing these types of data structures, but it does not come a priori with a bunch of graphics routines. That said, Windows and Mac and Linux and all these things have graphics routines that are written in C++, but they're all different. To say what are they like relative to Java? They're all different relative to Java. It's not standardized.

There's a wide variety of them out there, and they tend to have a lot of very impressive and different solved problems, because C++ has a longer history than Java that a lot of problems have been solved in C++ that are available to you, too. There's a lot of existing

code other than what you might think is standard. I will see you on Friday, and we'll be seeing some C++. If you have questions about your situation, come and talk to me now.

[End of Audio]

Duration: 43 minutes

ProgrammingAbstractions-Lecture02

Instructor (Julie Zelenski):Welcome. A couple things that I wanted to go through administratively to get us all up to speed – if you did miss Wednesday's class, the handouts are available on the web and the online video also, so you can take a look at what we did. We just did a course overview and a little marketing and just gave you some ideas of what we're planning to do together this quarter to make sure you're in the right place. That might be a good thing to review if you did happen to miss is.

The section times are now available on the web. They went up last night. They'll be up for the next couple of days. You go in and list the preferences of times that fit into your schedule, and then we do this big matching to get everybody into a section that works. There's no reason to do it early versus later. We do all the matching once we have everybody's preferences in. You do want to do your preferences at some point before it closes so we can make sure we have your information in the mix.

If you forget to do this, you'll be at the mercy of where we can fit you, and that might be much less convenient for you or, in fact, even impossible. Be sure to get your preferences in for us to do the match with. One thing I should have spent a little bit of time on on Wednesday – how the honor code applies in computer science and how to make sure that all of us are doing our part to uphold the honorable community that we have pledged to be a part of.

It's a little bit unfortunate and a little bit of an embarrassment to computer scientists that we actually account for a disproportionately large number of cases that are brought before the judicial board at Stanford. My personal thought about that is not that we somehow attract the dishonest student, but I think there are certain things that combine to make some of the pressures of CS a little bit outstanding relative to some of the other things. If you're up working late and you're trying to write a paper and it's not really coming together, at some point, you can say it's good enough and you can send it off, even if it is not your best work.

It's a lot harder to do that when you have a program that doesn't do what it's supposed to do. It may crash right away or give the blue screen of death, and that feeling of failure or lack of accomplishment is one that's hard to stomach as the talented people you are. Sometimes, that leads you into temptation about how you might be able to get the program working through means that aren't really your own work.

Let me be clear about what our rules are. I put out a big handout last time. There's also a big follow-up thing on the web that I think is worth reading, because I think everybody needs to know where the boundaries are. In general, you need to be doing your own work. Your own work means you're sitting down and coding and thinking and designing and debugging independently. That said, we have a lot of staff hours that can help you when you get into trouble, and asking questions about any general features of the language or the library or how this works or strategies is totally fine.

You should definitely rely on each other to get through those rough spots. When it gets to the point where you're looking at somebody else's code or looking at code together or looking at someone's code from a passporter or giving your code to someone else, you're crossing that boundary we don't want you to cross. When that happens, it often leads to the code coming out much more similar than you might imagine. In fact, we use automatic tools to help us identify situations where that might come up. We do move forward with bringing those cases to judicial.

It's personally a very sad thing for me. I want to get this out now. The point is we're here to help you, and we want you to succeed. If you're getting stuck and you're having trouble, the right thing to do is to ask us for help. We can often get you past that sticking point and get you onto doing the right work for yourself, but using someone else's code – people do make big efforts to try to conceal what they've done, and it doesn't work is the truth. We do catch it. The result is not a pretty outcome for anyone.

That said, you do not need to worry about us mistakenly identifying your code as looking like someone else's because we're asking you to solve the same problems. It turns out that code is as individual as a fingerprint. The way two people are writing an essay about the treatment of class in Dickens' "A Christmas Carol" don't look the same. It doesn't matter that we ask you to do the same thing. They really do look different. You have nothing to worry about as long as you are doing your own work.

Please do read those things, the handout and the web stuff just to go through all the minutia of how to know when you're getting into those gray areas, and certainly ask if you have any questions. About the alternate final exam – I don't want to get into the slippery slope of having much alternate exams. That actually creates issues with security and fairness that I'm not prepared to figure out how to navigate, but I did get a certain number of requests, and I've decided to be a little bit more lenient than I usually am.

I'm going to offer exactly one alternate final time, which is 24 hours earlier than our regular exam. We'll have more information about that as we get close to that, but for people who have an exact conflict or need to leave a day early, that will accommodate you. I will make no other accommodations. If this means you cannot take this class, then don't take this class. I am not capable of managing an excessive number of people who all want to take it at a time that works best for them. The university's scheduled slot is Friday, 12:15 to 3:15. Everyone really should have that time available, but if that for some reason is completely unavoidable for you, then Thursday, 12:15 is it.

One thing I'm going to try to do on Fridays is take a little time to sit and talk about stuff and having something less formal than office hours. It's not really about having CS questions answered. It's just getting to know each other and having a chance to talk. I think most students really wish they had spent more time with their professors, and I think I'm a fascinating and interesting person. That's something that comes up again and again is the feeling that, especially in a large class like this, the feeling of being anonymous and not being able to make that connection. It's something that I'd like to try to fix, and I want you to try to fix it with me.

Fridays, there's a school of engineering undergraduate council meeting that will meet every third Friday, but on the other two, my plan is to walk out of here and go over to that café that's down in the basement of Terman and get a nice cup of coffee and a cookie and sit down and talk. You're welcome to come and talk about anything and everything that's on your mind and just get to know me. If not this week, then maybe some other Friday will work for you. I will be happy to make your acquaintance.

What I'm going to talk about today – I'm going to do some C++. I'm going to talk about how C++ relates to Java, show you some sample programs, do a little coding with you, see if you can kind of get the feel for how things are going to work, try to talk about some of the specific mechanisms that are different in C++ that we need to get familiar with. The reading that goes along with this is the handout four, which tries to go through the high-level overview of the Java/C++ differences as well as the reader chapters, which fill out more of the details about the language.

What we're moving onto next week is the C++ library, so we're talking about string and stream and the CS106 libraries that go along with it that form the foundation of things we have to build on. Any questions administratively? Anyone read the handout two and had some questions about policies, procedures, or stuff I talked about Wednesday?

I also brought something. A former student of mine brought me her homemade caramels, and if they sit in my office, I'm going to eat them all, and I don't want to do that, even though they are extremely delicious, and I thought I would bring them as a bribery offering where it's hard to talk in a class that's this big, and you feel like everybody's staring at you, but I figure I'm going to offer up sugar as reward for courage, and so I'll have this little bag of stuff here, and if you have a question, just figure not only will you get your question answered magnificently, there will be a bonus butter/sugar bolus that goes with it.

This is Java. It's like a death match. What's the same? The first thing I want to talk about is that there are a lot of things that are the same, and so you are building on a background that is going to serve you well without having to make a lot of adjustments. They're almost so similar that at times, it can be a little bit unhelpful in the sense that it's easy to get tripped up because they almost but not quite are exactly the same. There was one unfortunate quarter where I was teaching two classes – one that was being taught in C++ and one that was in Java, and I was a mess the whole quarter.

Luckily, I'm not doing that now, so hopefully, I'll have my C++ hat on straight the whole quarter. The general syntax – I'm going to mention these things, and then I'm going to show you a sample program. Things like the comment sequence, the slash star or the slash slash – the use of almost all the punctuation characters is exactly the same. The statement terminator that the semicolon is used for, the commas to separate arguments going into a function call, the way you make variable and parameter declarations – exactly the same.

All of the primitive variable types – char, int, double, plus the kind of more esoteric log and float and short and things like that that are present in Java are present in C++ and have basically the same semantics. There is one minor adjustment there, which is Java's Boolean type, the true/false variable type is actually called Bool in C++, so you have to type a few less characters on that end.

All of the operators that manipulate the basic types, the assignment operator [inaudible] the equal sign, the equals equals to compare, not equals to see if they're not equal, the arithmetic plus, the plus equal, multiply equal shorthands, the plus plus that comes in both pre and post increment, the relational less than, less than or equal to, the logical operators and/or have the same behaviors. They have the same short circuiting, the same precedence table, so all of that is without change.

The control structure – so for redirecting control flow and making choices about what code to execute when, the floor and [inaudible] mechanisms, the if with the optional else, the switch statement, which you may or may not have seen, but you can definitely look up in the text if you haven't – that kind of has a variant of the if else, so it lets you decide based on the value of something which case to move into, the return that's used for the function – also things like the break and continue, a little less known keywords there or the do while – they work the same way in Java and C++.

Let's look at a complete C++ program. This one is printed in handout four, if you want to follow along there and make notes on it as we talk you through. What I'm going to do here is I'm actually planning on just walking through this line by line to kind of point out some of the things that are different. Then I'm going to move to the compiler, and we'll do a little bit of experimentation with developing and writing code and manipulating something that looks like this to get some of our bearings about what things are going on.

First off, just some general structure. A C++ program is traditionally written in a text file with a suffix some name.CPP. There's actually some other names that get used like .CC or .C that are also accepted by a lot of compilers. Some text extension that identifies that this file has C++ code – we'll use CPP. The program is traditionally in a simple form just one file with all of the code that's necessary to be in there.

Let's take a look at it. This one starts with a comment – always a good idea. This is average.CPP, and it adds scores and prints their average. It's just like all the good style habits that you have in Java. The next three lines are #include statements. #include is analogous to the Java import. The C++ compiler is pretty rigid about wanting to see things in the proper order. Before you start using some facility, it wants to know about it. That's part of the safety of making sure you're using it correctly.

If you plan on drawing something in the screen and doing some graphics, it wants to be sure you're making calls to the graphics function correctly with the right arguments, the right names, the right arrangements, and so in order to know that, it has to ahead of time have seen that bit of structure. The way this is done in C++ is through these interface or

include files. The #include mechanism is fairly primitive. It basically says look for a file with the name genlib.h and take its contents and dump them right here.

The contents of genlib.h and simpio.h are actually to give you a little insight. We're going to see them more later. It lists the operations and facilities that are available in this particular header that we're trying to include. In this case, the genlib is a 106 specific header that all our programs will include. It gets us set up with some basic foundations. The simpio is our simplified IO header that adds some features for interacting with the user at the console.

The next one down there is include . is a C++ standard header, and that helps to point out the difference between why in one place I was using double quotes to name the header file I'm looking for and in one place I'm using angle brackets. This is a way that's distinguished for what header files are, what are called system standard, and it looks for them in a particular place where the standard header files are. Those are always enclosed in the angle brackets. In double quotes are local headers that are specific to this project or this environment that are not C++ standard.

Whenever you see these double quotes, you'll think it's my own local headers or headers from the CS106 library. Anything in angle braces is a system standard header you'd find everywhere. It matches what you do in import. You say I'm going to be using these features in this class. Let me import its features into here so that the compiler knows about them.

The next line I've got here is a declaration of a constant. This is the const whose name is NumScores, and so it looks a little bit like a variable declaration. From this point over, you see I have int NumScores = 4;. It looks a lot like a variable operation. The const in the front of it is our way of telling the complier that this particular value once assigned will not change. It is equivalent to kind of the use of final in the Java language in many ways.

This is being declared outside of context. One point to make here is that C++ drives on more than one way of expressing code and what we call its paradigm. Java is the completely object oriented language. Every line of code you wrote went inside a class. It's all about classes. You wrote this class that manipulated the thermometer. You wrote this class that drew with the turtle. There was no code that ever existed outside of a class. You started off with public class something and you started writing methods that operated on that class. Maybe they were static or maybe not. The way you started code was always to say start from this class' main to get work done.

C++ is a hybrid language. It has object oriented features. It has the ability to find and use classes, and we'll see that quite extensively, but it also inherits the legacy of C, which is pre this whole object oriented development where it operates more procedurally, they call it, where there are functions that exist at the global level that don't have a context of a class that they operate in. There's not some [inaudible] that's being received. They're just functions that exist outside. It's a little set of things to do that don't live inside a class.

For much of the term, we're going to be doing stuff in a very procedural way, at least from the code we write, and we will use a lot of objects. We'll actually be mixing and matching a little bit of those paradigms together, so using the procedural paradigm to write a lot of code that happens to manipulate objects using an object oriented set of features. In this case, the program here – this is actually up at the top level, and it's saying this is a global constant that's available anywhere in the file. After this declaration, I can use the NumScores, which will refer back to this without any further adornment.

The next thing underneath that is a prototype. This is actually a little bit of a novelty that in Java you are allowed to declare if you have three methods, A, B and C, and maybe A calls B and B calls C, you could declare A, B and C in any order that's convenient for you. You want to put them alphabetically? You want to put them in the order they're called? You want to put them in the order top to bottom down or bottom up? It doesn't matter. C++ tends to want to look at a file once top to bottom and not have any reason to go back and revisit things.

It happens that in the main function, which is the one where code starts from, I want to make a call to this GetScoresAndAverage function. If I had left this prototype off and I had this code here, when it gets to this line where I'm making that call, the compiler hasn't yet seen anything about GetScoresAndAverage. It's on the next page. It will complain. It will say I don't know what the function is. You haven't told me about it. It came out of nowhere.

This prototype is our way of informing the compiler about something that's coming up later, and so the prototype tells about the return type, the name of the function, the type and names of the arguments and the order of them and then ends with a semicolon. It matches exactly the function header that we'll see on the next page, and it just tells the compiler in advance here's something that's coming up later in the file. It's a function with this name and this signature. It's something you didn't have to do in Java but is a part of the C++ world.

There's actually an alternate way I could have done this, which is when I show you the code for GetScoresAndAverage, if I actually defined them in the other order, I could avoid that separate prototype. It's a little bit of a matter of personal taste which way you feel comfortable doing it.

Let's look at this guy. Main is special. Lowercase m-a-i-n has no arguments and returns an integer type. This is the function in the C++ program where execution starts. There can be exactly one main function with this name and this signature, and when the program gets up and running, the first thing it does is start executing the lines of code that come in this function. No objects are created or any kind of fancy outer structure the way it might be in Java. It immediately goes here and starts doing what you told it to do. Main has a special role to play in any program.

What this program does is first prints something. This is the C++ syntax for printing things out. Even if you haven't ever seen it, you probably can get a little bit of an idea of what it's trying to do if you just step back from it and squint a little. It looks a little like system.out.print lin. This is what's called the consult out, or the standard out that's here on the left-hand side. The use of the double less than here is what's called the stream insertion operator. The standard output operation is saying insert into that stream and then in the double quotes is a string.

This program averages. It says insert next to that the NumScores, which is this constant. It got the value from above. And then another string, and at the very end, there's this little endl, which is the end line insertion of what's called a stream manipulator. It says at the end of this, put a new line and start any subsequent text on the next line. In Java, that would look like a system.out.print lin of this program averages plus the NumScores plus that using the string [inaudible] and conversion stuff in Java.

It looks a little bit different in C++, but in the end result, it's a matter of printing out a bunch of numbers and strings intermingled with new lines. Nothing too fancy there. The next line is making that call to that function that we earlier told it about but we haven't yet seen the code for that makes a call to get scores and averages. It passes the constant NumScores and says that's how many scores that parameter has used to decide how many iterations to run the loop requesting the values. Then it returns the average of the values entered and then we take that number.

This just shows that in C++, you can declare variables anywhere, either at the start of a block, in the middle of a block or wherever they are at. We took that value, stored it in this variable and then used it in the next print statement. The last thing we have is a return zero. The signature for main in C++ always returns an integer. That integer is not all that useful in most situations. It tended to be a kind of return that tells the operating system who invoked the program whether everything completed successfully or not.

By default, the value that's used to show successful completion is zero, so unless you have some other reason, it is return zero that you'll always see at the end there. In reality, if you were to return something else, you won't see a lot of other differences. If you return -1 or 642, in most contexts, that number's almost ignored. It is there just to make tidy.

I'm about to flip to the next part, and this is the decomposed function that main made a call out to that does the averaging of the scores. We've got a comment on the top that tells a little bit about how it works. It's always a good idea to get some documentation in there that's helping the reader get oriented. That same matching of the prototype that was given earlier – same name, same argument, same things, but instead of ending with that semicolon, it goes into the body.

Sometimes we'll call these very similar terms – this one we will call the definition or the implementation of a function, and that earlier just introduction of it is called its declaration or its prototype. They should match, and they're just there to give warning of

something coming up later in the file. It initializes a sum to zero. It runs a standard four loop here. Like Java, we tend to count from zero as computer scientists for a long legacy reason. Instead of going from one to NumScores, we actually go from zero to NumScores minus one.

Each time through the loop, we are prompting the user to give us a score. We are reading their score with a GetInteger call, and we'll see what GetInteger is. GetInteger is a CS106 function that retrieves a number that the user types in the console. We add that into the sum and keep going, and after we've done the whole thing, at the very end, we're doing a division of the sum by the number of scores to compute the average score that was [inaudible] and return that.

Student: Did you say why we didn't have end line?

Instructor (Julie Zelenski):In this case, I didn't have an end line just because I think it makes it nicer for the user. If you say next score and end line, then you're typing on the line underneath it, and so if you don't put the end line there, it's like you type right next to the question mark, and so it actually it an aesthetic thing. If I put it there, it would say next score and then I would be typing underneath it, and instead, I'm typing right next to it. I hit the return and then the next one comes on the next line.

So [inaudible] – and I will talk about that a little bit later today, but it is – that's a good question but a little bit advanced, so just take my answer and then we'll get to that in maybe 20 minutes.

Student: So for every main function, you always have to end with return zero?

Instructor (Julie Zelenski):Pretty much. As I said, you could put return anything, but as good form, return zero is a way of saying I successfully completed my job and I'm done.

Student: Do you use a void return type for the main?

Instructor (Julie Zelenski):In C++, no. In C, it's a little more lenient about that, and it will allow that. In C++, [inaudible]. I have to decide if I can get this to you without hurting you.

Let me do a little coding with you guys just to play around with this and show you the complier. I'd like to do a little bit of this. I think it's really easy for me. I talk really fast. You might have noticed. When I get going on something, it's easier for me to put up a lot of code on a slide and say here it is. It's all fully formed. The reality is when you're running your programs yourself, it's not going to come to you fully formed on a slide all ready to go. You're going to have to figure out how do you stack through making this stuff work?

I'm going to do a little bit of development of a program that looks very similar to the one I just wrote, but I'm going to play around with it a little bit to get at some of the things

that are different about C++ and how the tools work. This is basically what an empty project starts me off with. I've got the include for the genlib, but I'll always have my int main and return zero and nothing else doing. I'm going to sit down. I'm going to start typing.

I'm going to say welcome. I'm going to move to – let's do this. We'll play with this for a second. There are some things that I can do. I can say welcome a lot of times. That wasn't really so hot there. You guys can help me when I fail to type correctly. I say welcome a lot of times. Let's go see and make sure this works. Look at that. It doesn't like what I did. Let's see what it has to say.

It says cout was not [inaudible] the scope. Endl was not declared in the scope. This is an error message you're going to see a lot of times in various other symbol lanes that will show up there, which is the C++ compiler's way of saying to you I don't read your mind. You used cout. You used endl. These things come from somewhere, but they're not known to me until you make them known to me. The way I make those known is that those are part of the standard IO stream, which the input/output streams facility for the C++ language, and once I tell it about those things, it's going to be much happier, and it's going to say welcome to me a lot of times.

In fact, I could say this – 106B rocks. I'll do this. I'll say how awesome equals get integer and I will – it may help if I move this up a little bit so we can see how much do you love 106B? Here, I won't use my endl, so we'll see how the prompt comes out. I'm going to get an integer from them, and I'm going to use that to write how awesome that many times. What does it not like about that?

Again, it says get integer not declared in the scope. This is going to be the bane of our existence today, which is remembering that there are headers out there that have features that we're going to be using, and getting familiar with the ones that we're gonna need – there are probably about ten total that we'll use again and again, and we have to get our bearings about which ones are where. The documentation for all of our CS106 headers is available on the website in a nice HTML-ized version, and the ones from the standard libraries are detailed in the book in chapter three.

How much do you love 106B? I could say well, I love it two times. I say no, I love it 1,900 and so. 10,000 times later, if you're not convinced now, I don't know what will convince you. We're seeing some stuff. Let's see if we can make that averaging thing work for us. I'll go back to having it say welcome. That's what the endl was doing for me there. The prompt is waiting at the end of that line. If I have the endl, it will actually be waiting on the next line.

Let me put a bogus message here. Then I'm going to say double average equals GetScoresAndAverage and then let's go ahead and – I'm going to blow off the constant for a little bit because I'm going to be mucking with it anyway, and I'm not going to use that constant for very long, so I'm not going to worry about it yet.

I put my prototype up here so it knows what I'm talking about, and then I'm going to print out average is. I've got a little bit of structure there. I can even go in here. Copy and paste is a really good idea. Once I've written the function header once, I don't really want to make a mistake. Something that's interesting to know is that you can always write code that doesn't do what you want to start with but that lets you test some of the other pieces around it.

In this case, I haven't finished fleshing out what GetScoresAndAverage is, but I can write code around it that says well, it passes in this number and it does some stuff. Right now, it just returns zero. It's totally bogus. It's not solving any of my problems, but it does allow me to test some other part of the code to make sure that connection and other parts are working before I go on to work on this part in isolation.

One of the things I'll try to show you whenever I'm doing code is to try to give you a little of the insight to how I approach thinking about the code. I do think that one of the things that really distinguishes somebody who works effectively and productively from someone who spends a lot more time doing it is often just strategies and tactics for how you attack your problem. I'm hoping that along the way, I can kind of share a little bit of the insights that I use to keep myself making forward progress.

I am going to put in that four loop that I'm looking for. I'm really not this bad of a typist in real life, but it's easy to – the use of the braces here is what's defining the block structure. Without the open curly brace close curly brace, the four loop and the if and the other statements only grab that next line as being part of the indented body of the four loop.

I'm going to go ahead with my next idea. I'm going to put that in the sum, and then I'm going to do some things that seem bogus. We're going to get there. I'm going to make some mistakes. Some of them I'll make on purpose. Some of them I'll actually be making accidentally. We'll see which ones I'm better at finding.

I have my GetScoresAndAverage. It asks them for the scores. It adds it into the integer and then doesn't return anything. First, let's see how the compiler feels about that. The first thing you'll notice is that compiled. That might alarm you just a little bit, because if you go back and look at this code, it certainly seems wrong. Anybody want to help me with one of the things that's wrong with it? It doesn't return anything. That seems a little shocking. It doesn't return anything at all. What is it then actually going to produce?

What is it doing that is a little bit bogus? Sum is not initialized. These are two examples of errors that Java doesn't let you get away with. You may very well remember from having typed in some code like this in Java that it will complain. If you have a variable and you declare it and you don't initialize it and you start using it, the compiler says not happening. You need to go back there and give that guy an initial value before you start reading it.

Similarly, you fall off the end of a function. It said it was going to return a double – you better return a double. Every code path needs to return a double. You can't bail out early in some cases and leave it fall off the end. It considers both of those big enough errors that it won't let you get away with them. C++ compilers are a little bit more lax. It's crack mom, I told you. It's like well, if you don't want to initialize that, go ahead. What value does it have then?

It doesn't make it zero. It doesn't make it empty. It doesn't do anything nice or clever or helpful. It's like whatever kind of contents were lying around in that particular location, now it's [inaudible] initial value. The result will be that we could get some pretty strange results. If I run this thing right now – let's see. I type in some numbers. I type in somebody got a nine, somebody got a 67, somebody got an 87. I type in a bunch of numbers, and the average is zero. That was interesting. Zero seems like it had some rhyme or reason to it, but there's no guarantee. I could run it again and I might get very different numbers.

I got zero the second time. Good luck for me. Let's go in and start fixing some of this stuff. The thing to note is the compiler isn't nearly as aggressive about being on your case about things, so it does require a little bit more self-monitoring on some of these things. Seeing this result and going back and figuring out how I got into this situation – let me get that guy an initial value, and I'll go ahead and put a return down here.

We'll see if I like how this works any better. I put in five, six, seven and six, and the average is six. It kind of looks good, right? It makes sense to me that the five and seven cancelled each other out. But if I do a little bit more deep testing where I put in numbers like five, six, five and six, my average is five. In this case, I would think that I would be getting something like five and a half.

I go back here and take a look at what I did. I took sum and divided it by NumScores. What have I failed to do? Cast that thing to get it out of there. The default – this is the same in Java as it is in C++ is that if you are combining two operands in an arithmetic expression, if they are of mixed type – one is double and one is int – it promotes to the richer type. If one of those was a double, it would convert the other one to a double and do the calculation in double space. As it is, sum is an integer and NumScores is an integer. It's doing integer division. Integer division has [inaudible] built into it, so if I divide ten by three, I get three and that remaining one third is just thrown away.

Even though the result is double, that happens after the fact. It does the calculation integer space – the ten divided by three. It gets the three-integer result and then it says oh, I have an integer here. I need to convert it to a double on my way out. It takes that and turns it into 3.0. It doesn't recover that truncated part that got thrown away. If I really want to get a real double result, I've got to make one of those a double to force the calculation into the double space.

The mechanism of C++ is a little bit different than it is in Java. In fact, the Java mechanism does work, but the preferred form in C++ looks a little bit like this where you

take the name of the type you're trying to convert it to. It looks a little bit like you're making a function call-passing sum to a function that will change it into a double. By doing this, I now have one of them being double. Int on the other side gets promoted. If I do my five, six, five, six, that truncated part is now part of that calculation and preserved and pulled out and printed.

Let me change this a little bit to something else. Right now, it assumes that there's a fixed number of scores that you want to do. Maybe what I have is a big stack of exam papers, and I don't actually know how many are in the pile, and I don't actually want to count them ahead of time. I just want to keep entering scores until I get to the bottom of the pile. I'm going to change this loop to allow me to enter numbers until I say I'm done. Let's take a look at what that's gonna look like.

I'm going to get the value here. I'm going to change my code a little bit. Then if the value is the sentinel, then I don't want to include it in the sum. Otherwise, I do. Maybe we'll do this. We'll say if the value doesn't equal the sentinel, then sum adds into there. I also need to make this thing stop, and so I'm going to have to rearrange my loop a little bit here, because if you think about what I'm going to try to do, I'm going to prompt and get a value and then either add it in the sum or quit.

Then I'm going to prompt and get a value and add it in the sum or quit. I have a loop where I need to rearrange my notion a little bit here. One way I could do this is something like this. I could say this. While value does not equal sentinel, then add it into the sum. I'm going to kind of change my code around. Watch carefully as I reorganize it.

If the value is not equal to the sentinel, then I'm going to add it into the sum and then I'm going to get the next and overwrite the previous value. I inverted that loop that was there a little bit. It says while the value is not the sentinel, add it into the sum and then get the next value and come back around.

The problem with this code as it stands is there's something a little bit wrong about value. It's not initialized. I need to initialize it to something. What do I need to initialize it to? What the user wants. This little piece of code down here needs to come up and be repeated up here. I need to get a value from them and then go into if it wasn't a sentinel add it and get the next one.

This is what's called a classic loop and a half construction where there's a few things I need to do and then I need to do some tests and decide whether to go on with that iteration. As it is, there's half of the loop that starts up. They call it priming the loop. That's a little bit unseemly. There's something about that that's a little bothersome. Even though it's a very small piece of code – two lines of code is not going to rock the world.

I do want to try to get us into the habit of thinking if we can combine that code and unify it, that's better. It's better than repeating it. Repeating it means that if I ever change something about it, I have to change it in two places. There's an opportunity for error to creep in that I can avoid if I can get it to where there's really just one and only one

version of the truth. I'm going to change this. It's going to use the break statement, which you may have had a little experience with.

I'm going to run a while true loop, which looks like it's going to run forever. I'm going to prompt to get the value and if the value equals the sentinel, then I'm going to use break and immediately exit the loop here without completing the bottom of this loop iteration. It causes it to move on to the statements after the loop. By doing it this way, I now have one prompt, then a check, and based on that check, I either finish this iteration and keep coming around or I immediately break out saying that was the clue that we're done with this.

I would say that it's a little bit of an advanced question, but it turns out that the do while loop is not really that much cleaner in the end because you still have to do the prompt and test and decide when to add it in. Do well loops are just so unusual that they cause everybody to slow down a little bit when you read them. If you can write it using a more [inaudible] construct, there is some advantage to that.

I got this guy together here, and then maybe I should actually tell the user – this might be a good thing to say. Tell them what the sentinel is. I have to type better than I do. NumScores equals zero, and then each time we get one, we increment. Let's go back. We'll change our call to use a more ordinary value like -1. We'll type in five, six, seven, eight, nine and then -1, and the average of those is seven.

It should seem very familiar in a lot of ways but there are a few details that need to be different. I'm going to show you one of the little C++isms while I'm here because it's something that comes up in the standard libraries and it's worth knowing. There are some minor conveniences in the way that C++ provides certain facilities that are not conceptually a big deal, but you do need to know about them because you're going to encounter them in various interfaces.

One is the notion of a default argument. In the prototype for a function, there are arguments to a function where it's some very large percentage of the time always going to want to be a certain value, but you still want to leave it open for the user to specify a different value in the cases where they don't want to use that standard value.

One way of doing that in C++ is to write your function using a default argument where I say int sentinel and I said equals -1. That is providing for if somebody makes a call to GetScoresAndAverage passing an argument, it's used instead. If they pass nothing, so they just have open paren close paren and they don't give a value for that argument, that means the assumption is to use that default. That means that most people who are making calls to GetScoresAndAverage can just drop this number and get the behavior of using -1 as the sentinel. In the case where for one reason or another -1 was one of the valid values that you could potentially enter, you could pick a different one.

We'll see that in the libraries. It's interesting. The mechanism of it is really quite simple. It's just a convenience to provide for. You can actually have more than one default

argument. The idea is that you can only leave off when you're making the call the last most argument, and then from there, other ones, because it has to figure out how to match them up. It matches the arguments left and right, and as soon as it runs out of arguments, it assumes everything from there has to be used its default argument for it to match that call.

It allows for – you could have three arguments of which you could specify two or one or three if they all had defaults if you needed that. It's not that common. Let me go back to my slides and tell you about a couple other mechanisms of some C++ features that are new to us that we want to know a little bit about. There are two types of user-defined types that are very simple to get your head around. You want to know what the syntax for them is and what they do.

It's the enumerated type or the enumeration where at the top of your program – up there where I was defining constants and doing #includes, this is where we put information for the compiler that is of use across the entire program. This was also where we would put new user defined types. In this case, I want to define a type direction T which can take on one of the four values, north, south, east or west. That whole package up there – that enum direction T north south east west is the way of defining the new type.

You're saying direction T now comes into the space as a name that can be used for variables, parameters, return types – any place you could have used int you can start using direction T. It has the expectation that variables that are declared in direction T will be holding one of those four values. It's like north, south, east and west got defined as constants, and they are, by default, assigned the values zero, one, two and three in order unless you do anything special.

You can use them – you can do things on enums that are very integer like. You can assign them. You can compare them. You could use less than and greater than. You can add them. There are things – they are largely implemented underneath the scenes as numeric types, but they're a nice convenience for making it clear that this thing isn't just any old ordinary integer. It's specifically – it should be kept into this constrained range. It means something a little different. It's just a nicety. It does not have a lot of heavy weight feature associated with it, but it's just a nice way of documenting that something is this kind of set up.

This one – the record of the [inaudible] type – much more broadly useful. It's just an aggregate where you can take a set of fields of same or different type, give them names and aggregate them together. You say here is student record, and the student has a name, dorm room, phone number and transcript. Aggregate it together into a new structure type.

In this case, the point T [inaudible], and so like this – this is the kind of thing you put up at the top of your code that says here's what's in a point T, the field names and their types, and a little point of noteworthy error producer is that there really does have to be a semicolon at the end of this. Similarly, there has to be a semicolon at the end of the enum for direction T.

If you forget to do that, there's a cascade of errors out of that that are really a little bit mystical the first time you see them. You learn to identify this quickly later on. It ends that declaration and then allows the composite to move on to the next thing, not assuming you're still doing more work with this. Once you have this type declared, you can make variables, return types, parameters, all that stuff, and then the access to the members within the fields within a [inaudible] looks like access to an object did in Java where you have the variable name on the left and then a dot and then on the right, the name of the field you're accessing, setting, and reading.

You can do things like P = Q which does a full assignment of one [inaudible] onto another, so it copies all the fields over from one to the other. It's something simple that does have a lot of usefulness. It makes a lot of sense when you have a program and you do group things together. Here's all the information about this part of my data structure – a student, a class, a dorm – all the information being aggregated together into one unit is actually a nice way to keep your data organized.

There are two point T variables. One is P. One is [inaudible]. I'm saying P.X. I'm saying the X field of the P variable is to be zero. At this point, the P.Y's field is nonsense. It's just garbage. I said P = Q, which basically says take the nonsense in Q and override it onto P and make P be as nonsensical as Q is in terms of its contents. Not very useful code.

The last thing I'm going to show you today is to talk a little bit about parameter passing. This is going to come up again and again, but this is just kind of a quick first mention of these things. Someone had asked earlier what is the parameter passing mechanism that's in play? The default parameter passing mechanism is what's called pass by value. It copies. If I have a function here binkie int X and Y, in the body of it, it tries to do something like double the value of binkie and reset Y to zero.

When I make a call to binkie and I had A set to four and B to 20, when I made the call to binkie, the pass by value really means that the X and Y parameters of binkie are copies of A and B. They are distinct. They are new integer variables, new space, new storage, and they got their initial values by taking the current values of A and B and copying them. Changes to X and Y affect binkie's context only. That four that came in here and got doubled to eight, that Y that got set to zero are live here, but when binkie exits and we get back to main, A and B still are four and 20.

It just did full copies of all the variables, and this is true for all types of [inaudible] and enums and ints and chars and all that. It's copying the data and operating on a copy. In most situations, that's actually fairly appropriate. You tend to be passing information in so it can do some manipulations.

In the situation where you really want to be passing in and having changes be permanent to it, there is an alternate declaration where you and an & to the type. Instead of being an int, it's an int&, and that changes this parameter from being a pass by value to a pass by reference or a reference parameter. In such a case, when I make a call to binkie, this first

argument will not be a copy. The second argument's still a copy because I haven't changed anything about it, but when I say binkie of A and B, what the binkie function is actually getting in that first argument is not a copy of the value four.

It's getting a reference back to the original A. For the Y parameter, it's getting a copy of 20. When it makes this change over here of trying to take X and double its value, it really did reach back out and change A. After this, A would be eight. B would still be 20. It allows for you to pass some data in and have it be manipulated and changed, so updated, adjusted and whatnot and then come back out and see those changes that is for certain situations very useful.

This mechanism doesn't exist in Java. There's not a pass by reference mechanism, so this is likely to seem a little bit bizarre at first. We will see this a lot, so this is maybe just our first introduction to this. We're going to come back to this more and more. One thing I will note is that the primary purpose of this is to allow you to kind of change some data. It also gets used in a lot of situations just for efficiency reasons where if you have a large piece of data you're passing in a big database, to copy it could be expensive.

Copying just so you could hand it off to a function to print it would be unnecessary, and so by using the reference parameter in those situations, you're actually just allowing it to avoid that copy overhead and still get access to the data. Sometimes, you'll see it used both for changing as well as efficiency.

I'm not going to show my last slide, but that's what we're going to be talking about on Monday. We'll talk about libraries. Looking at chapter three is the place to go in the reader. I'm going to be walking over to Terman momentarily, and I would love to have somebody come and hang out with me. Otherwise, have a good weekend. You have nothing to do for me. Enjoy it. It's the last weekend in which you will not be my slave.

[End of Audio]

Duration: 52 minutes

ProgrammingAbstractions-Lecture03

Instructor (Julie Zelenski):It's good to see you guys. Just a note about the handouts that are going out today so you aren't confused about our ability to count in sequence – six, eight, nine and ten are in the lobby. Five and seven we're going to bring at the end of the lecture, and I'm trying to avoid killing too many trees by what we do with five and seven. Five and seven are the handouts that tell you about the installing of the compilers and the debuggers of the two compilers, and there's two versions of them. There's a Mac version and a Windows version for the X code versus Visual Studio.

We printed an estimated amount of how many people we thought were going to be using the PCs and how many using the Mac, and so when we bring those in at the end and you're picking them up on the way out, be sure to get the one that you need and not take both. Take the ones for the product you're going to use. If you happen to believe you're really going to use both, it's fine to take both, but I figure that's pretty rare. If you know what platform you're on, take the one for yours and not both.

Those are the ones that tell you about how to get your compiler set up, which seems to be something a couple people this weekend were ready to get a jump on. Assignment 1 is also one of the handouts that's going out today. For assignment one, just to kind of come up to speed is that the first assignment here is not some big, complicated program. It's actually a set of small programs. Each is less than a page of code. They're designed to exercise some skills in isolation – learning how loops and variables work, learning how our graphics library works, and learning how strings work in C++ and files.

The more complicated part is going to come from learning how to express yourself in C++. For those of you who are a little unsure about whether your background is good and whether you're in the right place, this is actually a good testing ground. You should look at those problems and say oh, if I were solving this in my native language, I would be totally able to write this out without even stopping to think much. In C++, I'm definitely going to have some new things about learning how to convert what I know to how it works in C++.

If you do find these challenging to write, that's probably a sign that you're a little bit ahead of you. If you find them completely trivial and way too easy, you may want to think about 106X. One way to gauge that is to look at their first assignment. If you feel ready to do something like that, then maybe that's a place that would give you a little bit more of the challenge you're looking for.

We have all the section preference information in and we're doing the big matching. We will email out the section assignments. Our sections mostly meet on Thursday, but some are on Wednesday and Friday. You'll get an email tomorrow that tells you about where and when your section is meeting. If you have a conflict that has arisen post the decision making, we have a little bit of an ability to do some add late and rearrangement, but it's pretty tricky. If you can make the section that you're assigned to, that's going to help us the most. We will try to accommodate you if we can to make a switch for you.

What are we going to talk about today? We're going to talk about libraries and C++ and string and stream classes. There's some [inaudible] mostly chapter three and then the handout that went out last week about general C++ has some information that's useful. There's also a library reference I gave out today which is handout ten, which is nothing but an overview that reminds you of some of the features. One other thing that's going to be the theme for the next three lectures or so is going to be there's a lot of material in C++. There are these huge libraries that do lots of things.

The goal of these weeks is not to make it so that you are an expert on all the minutia but that you are comfortable with the basic facilities and you know where to look to find out more. If you're trying to figure out how the substring operation works on a C++ screen, what are the arguments? What do they mean? What are the cases I need to be worried about? You'll know where to start and how to find that information as opposed to memorizing the whole of the library. It's not really a feasible task and not even important.

Think about it as need to know. I'm going to get you the basics so you know what's out there and then as you start to write code, you'll learn the specific things you need to solve the problem at hand. Any questions administratively? I had one student come and hang out with me on Friday. I'm hoping that was because it was so last minute. This Friday, I don't have a meeting, so put it on your calendars now for Friday hanging out.

C++ libraries – the notion of a library is really nothing more than saying you've got some functionality that you want to provide to all users of C++ or all students enrolled in CS106. There is some reasonable grouping to these things. You have a bunch of operations that operate on strings or that allow you to do graphic works or allow you to do event handling or something. The library is the packaging device in C++ where you say here's a set of routines. Typically, it comes with two pieces. One is the interface or declaration or header file. It tells you about what routines are in there – what are their names? What are the prototypes? How do you use them?

It often contains good comments about the things that you would need to know as a client using that facility – how to use it effectively and correctly. There is the code that really implements it that when you make the call to a substring operation, how does it actually work? Well, there's some code that backs it that does the operation that gets called at runtime when you make a call to that function. The libraries that we're going to see this quarter form roughly two big groups.

There is the C++ standard library, so things that come with every C++ compiler. No matter where you continue coding in C++, you will always find things like the C++ string, the C++ IO stream, the file stream which is the F stream. There's a math header. There are headers that deal with all sorts of other facilities that are beyond what we're doing here. These are the ones that we'll see most commonly in the early part are string and stream.

The typical include for them is going to be the angle brackets. That's the sign to the compiler we're looking for something from the standard header locations. One way to remind yourself about how to distinguish these from our special libraries is that you're going to see these very terse and lowercase names. Part of the legacy of C and C++ was as a professional programmer's tool, they tended to value terseness over any kind of verbose and descriptive names, making it a little easier to type a little faster to get your point across.

Things like the cout, which is the console out stream, the get line, the [inaudible] call. There's the substring name of the function there. Those tend to be short and throw away vowels where they can. They tend to be all lowercase. Whenever you're looking at a routine, you might wonder where it comes from. If it has this capitalization scheme, it's likely to be something coming out of the standard libraries.

In addition to what we have present in the standard, we also have about seven libraries that we've included as part of CS106 mostly to make our lives a little bit easier. Things like the random library or the simple IO library – they actually layer on existing functionality that is already present in the standard libraries, but the way the functionality is expressed in the standard is just a little bit awkward or unhelpful for the task we need to do. We've provided a layer that cleans it up for you.

The graphics is a good example of where there is no graphics library included in standard C++. If you're working on graphics on Windows, you have access to a different toolkit than you do on the Mac or on Linux or some other platform. We have tried to abstract out a very simple graphics library that we can run on both Mac and Windows that then we provide one interface through that in turn talks to your platform in its native language to take those Windows and drawing things happen.

Our header files are always in the double quotes. We typically use a strategy of having capitalized verbose names. We don't throw away our vowels. We try to make a little bit of sense to describe the action that's being taken. Today, what I'm going to look at is I'm going to look at one of the 106 libraries here at the beginning because it's a nice, easy one to get our head around.

I'm going to go and look through the C++ string library and then I'm going to hopefully get a chance to even start talking a little about the C++ stream library. Along the way, there will be two additional 106 libraries that help out with string and stream that provide a little bit more functionality than what's already there.

Randomness comes up in all kinds of simulations in game playing. You want the computer to simulate some random behavior – flipping a coin, rolling a die or shuffling an array or a deck of cards. Computers actually aren't capable of true randomness in the sense that you might think in the real world, but they have what's called pseudo randomness behavior where it can generate numbers in a sequence that appears effectively random from the outside even though there actually is some determinism in how it operates.

There is a set in our library – there are four functions that form the CS106 random library that are used for all kinds of random behavior that you want. I note here that they are free functions, and by free functions, I mean functions that aren't on a particular class. They're actually globally accessible. You don't call them by sending [inaudible] particular object. They just exist at the top-level name space and you can just call them anywhere and any time. When you're ready to get some random behavior, you make a call to one of these routines.

There is an initialization routine for this library. The randomized call – that's called once and exactly once in your program, usually at the very beginning to set up a new random sequence. That's what's called seeds of the generator to get it started in a new place. Then, once you've made that call, you can intersperse any number of these calls to simulate certain random events.

The standard random number generator that's in C++ provides all of this through one call. We'll generate a random number from zero to the largest number possible, and then you can decide how to map that to other things. If you want the ability to flip a coin, you want to say half the numbers are odd and half the numbers are even. You could do something like generate a number and see if it's odd or even, or see if it's from the bottom half of the range versus the top half.

What these functions do is just provide that functionality and package it up for you neatly. You can say things like random integer low to high – you say one to five. It will give you a number from one to five inclusive. If you keep calling that, you should see an unpredictable sequence of the one, two, three, four, five coming in jumbled and mixing up. There are no real guarantees about what order you'll see it that will allow you to simulate random events.

The random real – same sort of idea but in this case using boundaries that are expressed in real numbers and returning in real number. Again, the bounds are inclusive, so it can't actually return the number low or high or anything in between. The last one is simulating a probability true/false value. Given the probability of 0.5, half the time it should return true and half the time it should return false. If you give it a probability of .25, one quarter of the time it will return true and the other three quarters of the time it will return false. It allows you to simulate coin flips or other random events where you have a [inaudible] distribution.

The point of a library is to take some set of facilities that are needed, package it up, have a vision of how they work together, a naming convention and design convention that makes them coherent, that they provide some convenience and they're complete. They cover all the bases. These three provide a pretty good range of different kinds of random events. There are still other things you might need to simulate, but you can typically do it in terms of using one of these.

You could also have left one out and have to simulate the others from it, but each of them has a client use that's pretty handy, so it actually has all three of them for your use.

[Inaudible] comes out of random.h in the CS106 library. The .h is just a convention for the extension for header files. .txt gets used on text files and .cpp gets used on source files. .h is for header files, which are descriptions of routines but no real code is typically in the .h file. It's an interface file they call it.

In Java, there isn't that distinction. Everything is all in one file, but the definition of a class serves as both the description for a client using it as well as the implementer implementing it. C++ has them separated. Let me look at C++ string as the next example of something we have to make use of in getting things done. The C++ string type is actually defined in a header file, and it's a library that's added into the language.

Unlike int and bool and double that are part of the language and can't be separated from it, string is kind of an add on that's defined through a library. It models a sequence of characters including everything – letters, numbers, digits, punctuation – and the string is defined as a class. In the same way that in Java you're used to the class being the pattern from which you can declare and initialize objects that you can then message and do things with, string in the C++ world is the same sort of deal.

You have a string class. You initialize string objects. You send messages to those string objects to ask them to do things for you. Asking a string to give you the character at a particular position or the number of characters or to insert some characters or change some characters within the body of the string are all done by messaging the string. A couple simple operations – I put a little bit of string code to get started.

The variable name is actually string itself. There is something a little bit different about string when you declare it and you don't initialize it relative to the things we know about primitives. When I say string S and I don't say anything else, you might assume then similar to the primitive types that S is garbage. It has some sequence of characters. In fact, string has what's called a default constructor, one that's invoked when you don't specify otherwise such that when you initialize a string with no other explicit information, it will assume you meant to set it up to be the empty string.

Making a call string S actually declares and initializes a string with no characters. If I were to ask it for its length, which is the way we ask for the number of characters in it that's being used right here – for example, S.length and then close the open paren there. It would return zero on the empty string. We can use square brackets like the array notation you're probably familiar with to access individual characters of the string.

Applying the square brackets to S – S sub I sometimes I'll call this accesses the [inaudible] character within the string. The character's index starting from zero – so if I have a ten character string, they actually are indexed zero through nine. The C++ string – the square brackets allow you to access that character both to read it and to write it. A C++ string is mutable. The Java string is immutable. Once you create a string, it has a certain sequence of characters, and although you can make a new string and overwrite that one, you can't go in and just manipulate the string in place and change its contents.

C++ you can do that. I initialized string in this case to the string literal or string constant CS106, and then I ran a loop over the index of the proper range of indices for this string, and then I used the two upper, which is a function from the standard C library that takes a character and returns its uppercase equivalent or unchanged if it's not a letter, and then [inaudible] S of I the result of two upper.

The effect of this was for each lowercase character in the string, we overrode it with its uppercase equivalent. Any other existing uppercase or punctuation characters were left unchanged. You can make assignment into that, which is something you cannot do with the Java string.

Student: Can you insert [inaudible]?

Instructor (Julie Zelenski):I certainly can. I'm going to show you that in about two slides. There are a whole set of member functions that then do these things. This one allows you to have the sequence of five characters – what if I want to put one in the middle? I'll use something called insert. If I want to take one out in the middle, I use something called replace or erase to pull it out and put something else in.

Many of the built in operators – things like equals and less than or less than or equal to, not equal, have extended meanings that apply to strings when they're used as the operands for those types. I can assign two strings using equals. If I say string S equals T as I'm doing right here, then whatever value T is, S becomes a copy of that. S and T have the same value, but they're not related in any important way going forward. We have two copies that both happen to have the same five characters.

For example, the first thing I did after this was change the first character of T to be J, so now T is jello. S is still hello. It was initialized from the same sequence, but they don't retain any kind of aliasing from that point forward. I'm able to compare two strings directly to see whether they're lexicographically equal or less than according to ASCII ordering. I can say if S = = T – in Java, that didn't do what you wanted. It did compile, but it didn't test the thing you were hoping for. In C++, it does do what you're expecting, which is to say take two strings and say do they have the same sequence of characters.

If I have assigned S to T, if I do S = = T, it's going to say yes, they have the same five characters in the same order. Once I've changed one of them, then they'll come up as not equal. I could do less than and less than or equal to to see in ASCII ordering which one proceeds the other to do sorting of strings. Just like you think of as the integer types in double touch, those operators have reasonable meanings applied to strings.

The plus and plus equals is what's called overloaded, so extended beyond its usual meaning for addition to do concatenation of strings. I can take S and I can add to it a character space at the end, so now instead of being just hello, it's hello space. I can also add strings to strings, so I can take T and use the shorthand plus equals, which takes jello and turns it into jello jello there, attaching another one on the end.

The concatenation for the C++ string only operates on strings and characters whereas in Java, there's this kind of automatic mechanism where things like doubles and integers are converted to string and added into the concatenation. That does not happen in C++. Concatenation is just for strings and characters. If you have something that's in numeric form and you want to add it into a string, you'll have to first convert it to a string. I'll show you a routine that does that a little bit later.

I would be happy to do that. Most of the things that I'm talking about actually are in handout four was well as repeated in handout ten and in the reader. There are a million places you can look for information on strings.

Most of the heavy lifting on the strings is done via these member functions. These are part of the string class, and so these are operations that apply to string receiver objects. They're not free functions. You can't call them outside of a usage where you're saying on some receiver string, apply this function using these arguments. For example, the length member function is applied to a string.

Just to note here, the word member function is vocabulary-wise the same thing as method. Java programmers tend to call the functions that are defined as part of a class methods. C++ programmers tend to call them member functions. They really mean the same thing, but I do try to use the word member function because we are a C++ class, and that is kind of the convention. I'll probably end up using both accidentally without even noticing it. Hopefully, it won't cause you too much grief there.

The member function here is saying str.function R is saying apply the function, send the message function to this particular string with these arguments and then get its answer back or have that operation happen. I can ask a string for its length in terms of an integer. It tells you the number of characters. I can ask a string to look for a particular character or string sequence substring within the characters that that string maintains right now. It will return the index of the first occurrence found, scanning from left to right or a string::end pause. It's a little bit of a funny return value, but it is the return value that says I didn't find it.

It's a string::end pause. It's an integer value that is distinct from any other valid index within the string itself to tell you it didn't find it. Both of these have a default argument on them. We talked a little bit about that last time. If I do not specify that second argument when I'm making a find call, it will assume that you want to start looking from the beginning. If I do specify it, then it will start from that position and scan from there to the end of the string. It's a way of targeting the place you're looking for a little more precisely than just starting from the beginning and going to the end.

C++ does allow what's called overloading. In this case, the function find that finds a char and the function find that finds a string both have the same name, and so that name can be used for multiple purposes as long as there's a sequence of arguments that distinguishes them so that when I make a call to find, it knows whether the first version or the second version by virtue of whether the first argument is a character or the first

argument is a string. That can be extended to other types. This is typically used when you have an operation that really has the same behavior but some slightly different sequence of arguments is required to invoke it.

It is not something you want to use a lot to make a bunch of similar named functions that don't have similar operations. It allows for a convenience when there are two or three variations of the same theme. They might all come under the same name by virtue of overloading.

Substr is something that given a receiver string and a position in a length will extract a new substring out of the middle of the string that was received. If I take the hello string and starting from position zero take two characters, I get the string he. It copies them. It's distinct from the original, and so all it did was get its initial sequence by copying characters from there. If I go in to change the hello string into jello, that he string stays he. They're not attached in any long-term way.

Insert, replace and erase are all of the family of something that I call modifiers or mutaters that change the receiver string. You can send these messages to a string to cause new text to get added into the string, text to be removed or text to be deleted and replaced with something else. Inserting – someone asks, well, how can I put new characters in the middle? Well, I put the position where I'd like them to go. If I say position zero and I say put the string I in there, then it would bump everything down and put I in the front and replace it. If it was hello, it would be I said hello. I inserted the string I said.

The replace at a position removes length characters starting at that position and then replaces it with that character. It's a way to take a chunk out and put something else in instead. Erase does a straight remove at a position. Take this number of characters and throw them away, deleting them from the string and making it shorter. All of these change the receiver string. When you say str.insert, str.replace or str.erase, after that call, str now actually has new contents based on what you've asked it to do about changing and mutating its contents.

Here's something I should tell you a little bit about C++ string relative to Java string. C++ is kind of an industrial strength language that's targeted at professional programmers. It does not make any guarantees to you about what happens if you misuse these calls. If you give it a position that isn't valid for this string or a length that isn't valid for the string, there is no contract in the C++ libraries that said this is what will definitely happen. It doesn't say oh, it's definitely going to throw an exception or throw some sort of error. It doesn't say it's just going to truncate it at the end.

It says that the library is free to do whatever is convenient for it up to and including just crashing. It does mean that as the programmer using these calls, it is a little bit more on you to be careful that you're using them correctly and making the numbers inbounds for the string in ways that will produce correct results. It might be that it will produce a nice error message, but there are no guarantees. You wouldn't want to come to depend on that. You want to just be careful about knowing what the right numbers are.

Unlike Java, which is very attentive to those things and on your case when you're a little bit out of bounds, in the name of efficiency, it tends to just breeze through that stuff. I'm going to show you a little bit of coding together just for fun. I like to sit and show you some things. If I were to do something like want to count the occurrences of a particular character within a string, I could write a loop that looks like this.

I could say int count = zero for – and this is a very ubiquitous loop for operating over a collection – in this case, the collection being the characters in there from zero to this length. If S sub I equals the character I'm looking for, we would increment the count and then return it. I put this down here in my code and do a little testing of looking for the character C in Chihuahua cheese crackers.

Let's take a look at that and see if we manage to count the number of Cs in my list. There are four, apparently. Let's go check and see if that comes up. It looks good. We did a little bit of counting. We're feeling okay about that part. Let me do something where for example I want to remove all of the occurrences from that. I'm going to write this two different ways to highlight a little bit about how things work.

I'm going to design a remove occurrences that given a character in a string will return to you a new string where all the occurrences of CH have been removed. Easy enough. It's not going to modify the original string. It's going to return a new one. Here's my strategy. The way to build these things up is I could go through the manipulations of trying to take the characters out in place and figuring out where I'm at, but often, the easier way to do this is to build up the result – decide when to append or concatenate a character from the original string and when to ignore it and go past it.

I can do something like this where it's like if the character I've just seen is not the one that I'm trying to avoid, then I can just add it into the result. When I'm done, I have the result. If I do this and I change this call down to remove occurrences – I'm counting on the fact that result is initialized to the empty string. I didn't actually say anything there. I could, for example, do this and that doesn't change anything about it and you might feel a little better about seeing that explicit initialization, but C++ programmers are very used to seeing uninitialized strings and knowing that that means they got the default initialization to the empty string.

When I didn't find the character I was looking for, I [inaudible] the result. I'm going to switch this up just a little bit. I'm going to change remove occurrences to instead of making a new string to actually modify the string that I have. I'm going to change my code down here to match what's going to happen. I'm going to set it to do this, and then I'm going to call remove occurrences C of S, and then I'm going to print out S afterwards.

In this case, I don't expect there to be a second string created. I expect us to go in and modify that string in place, truncating some of those characters and taking them out to make this work. I could kind of do this thing where I'm walking down the string character by character and then deciding whether to collapse over it. I'm actually going to

change my strategy entirely just so I have a little practice using some of the other routines, and I'm going to end up using the string find.

We'll start with this. I can actually do this. S.find of CH, and if I don't give that second argument, it's going to start from the very beginning and look all the way through and see if it finds it. I'm going to put a hold that result at a variable and I'm going to say while found equals the result of calling find and then I'm going to stick it in. This is a very C++ way of coding. It's tightly combining this up. In this case, I have an assignment and a comparison all in the test of the Y loop.

I'm making the call to ask the string to find a particular character, storing that result in an integer here so I can use it and then comparing that resultive string end pause. So the string end pause is a little bit of a funny C++ syntax there, but the way to read that is within the string class – string:: says within the string class, scope within the string. There's a particular constant called end pause, which is used as the return value in cases like find when it's looking for something. End pause being part of the class is a way of avoiding it conflicting and interfering with any other usages where you might have variables named end pause or similar functionalities.

It's tied to the string class through the scoping mechanism. I check and see if it's not string end pause, and then if it's not, then I go into the loop here and I can do an erase of one character at position. Erase takes the position in the count and removes the number of characters I specified from that position. Then, it will come back around. I have passed string by reference coming into here, and that's a very important part of what's happening here because these calls to erase that are modifying string – if I have not passed string by reference, they'd be operating on my copy.

I'd go through all the trouble of erasing all the Cs in my copy, but when I got back out to the main call, none of those effects would have been permanent. Passing by reference really means that what remove occurrences got was access to the original S. I should make these names – I'll call this my string out here so that we don't get any confusion about the two names. The my string variable in main is really being accessed by remove occurrences without a copy. It's reaching back out into main and making changes to the my string itself.

What does it not like about that? Oh, pause. I called it found. I've achieved the same thing. That's one of the things about the string library is it's so big and has so many different ways of doing things that often two people or ten people running the same task won't even come up with the same solutions. I could have used a replace where I replaced the character it found with the empty string.

I can build it up through concatenation. I can take it down with erase and replace. I could insert the other way around. There are a bunch of things I can do that in the end will achieve the same effect but show that there are a lot of ways to accomplish the same things. The library is pretty rich and has a wide variety of tools in it.

I'm going to make one change to this to show you how I can make it slightly more efficient. This is silly because strings are typically very short, so it doesn't really matter, but I'm going to do this and I'm going to use found as my index on subsequent calls. I can say starting at found from zero, do my search from zero and then found, and then any subsequent calls will pick up where I left off. The next time around through the loop, found is at the place where I found a previous occurrence of that character and it says starting from that position now, look forward and see if you see any more from here to the end.

For a very long string like this, it ends up doing a lot less work. It doesn't start at the beginning each time. It just picks up where the previous occurrence was found and goes from there to the end. It's a small change, but no big deal.

Basically, what you're saying is find needs to return something that says I didn't find it. It could return to you zero, one, two, three, all these indices. It actually needs to return to you, and it uses a special sentinel value that says I didn't find it. You might think that might be negative one or some other thing. A good programming form would be to have a constant for it so that you don't have any magic numbers embedded in your code. That constant is defined as part of the string class. Just the syntax for accessing that constant of the string class is using the string class name :: end pause.

It's basically the syntax for I have a constant that was defined within a class. How do I get to it? I use its class name, two colons and then the name of the constant. It's just C++ for something that in Java looks a little bit more like class name dot. Question?

Student: Sometimes I see a function declaration before [inaudible] and then the definition afterwards. Is that just a matter of preference?

Instructor (Julie Zelenski):It totally is. Probably I'm being a little bit lazy in class, which is if I put the function definition up here, then I can call it down here because it's already been seen. If I put it down here, then I need a prototype up there. The prototype means I have to be a little bit more careful. When I change the name, I have to change it in both places. If I change the argument, I have to change it in both places. The problem, of course, is that when you read the code, it probably reads a little better to say here's the main which makes calls to A, B and C.

Some of it has to do with it's a little bit harder to maintain in that form, but I think it's easier when you're done to read it. You're totally free to do it either way. You should probably pick a strategy and go with it. Maintain the prototypes is not really that much work once you get used to it, and I think in the end, it probably is a little bit cleaner. When I'm being lazy in class, I'm much more likely to just throw them up there to save myself some time. It's good to note that there are a lot of things that will slip by me if I'm not being careful.

Let me go back and pick up a few last details about string that I don't want to overlook before I move away from this. There are library functions that are need to know. I have

them sketched out in a couple places, and you can look at them and see what they do – knowing they're there and then learning about them as you encounter them is a fine strategy. There are a couple additions in our [inaudible] which is a 106 specific header file which are just some things that for one reason or another are a little bit harder or more annoying to do using the standard tools than we think is worth putting on your plate for now.

We have two convert to upper and lowercase that given a string just convert it to its upper and lowercase equivalent. There are some things that do conversion between string and integer and string and real when you have it in one form and you need it in the other. Here's something that just does that for you as part of the string library. It's just some simple things that you might find yourself needing and you just want to know they're there.

Here is something that is a little bit of a bummer. Part of the legacy of C++ being built on C means that every now and then, there's a little bit of a history in our deep, dark past that pops its head up in ways that are a little bit surprising. For string, it turns out there is a little bit of a weirdness here that I want to point out before you run into it the hard way. There is a notion of the old style C string.

The original C language didn't have a string class. It actually doesn't have any object [inaudible] features at all. It did have, though, some other more primitive handling of sequences of characters. This is a very common [inaudible] to have something. I've put in parenthesis what it actually is. It's [inaudible] an alternative. Don't worry about what that phrase means. That's just for those of you who have seen it a little bit before.

That would be fine. We have this better string object that has all these fancy features, so you'd think we could just use that and ignore the fact that the other one is there. It almost – 99 percent of the time, that's exactly how it's going to work. It does turn out that there are a few situations where this old style string pops its head up and gets a little bit in our way. One way that may be a little bit of a surprise is that the string literals are actually C strings.

When you see an open quote, some characters and then a closed quote, the compiler interprets that as a C style string. It also has a mechanism by which if you tried to use it in a context where you needed a C++ string, it will automatically convert it for you. It will take the old style string and make a C++ string out of it. That means that basically I can use them wherever I want and it will mostly work out.

There is a way you can deliberately force it, if you use what looks like the type case here. This is actually calling the string constructor, and you pass a string literal or string constant. It will turn it into a C++ string manually there. It's going to turn out that you might need to know this. There's also the other problem of what if I have it in one form and I want it in the old form? I have the old form. I want a new form. I have the new form. I want an old form.

There is a member function on the string class that will return to you an old style string from a new style C++ string, and it's called the C_str [inaudible]. They let you convert. Why do you care? It turns out there's one thing you'll definitely run into, which is when you're opening a file stream, you want to say this is the file on disk that you want to identify. It turns out that that library requires the use of a C string as the name. There was a little bit of an issue trying to get all the libraries to come together at the right time, and it turns out the stream library got finalized before the string library was done, and so it depended on what was available at the time, which was the old style string.

Even years later when they're both happily debugged and working, it is still the case that when you use the stream library, you have to describe the file you want by using the old style string. If you had a C++ string variable that held the name you wanted, you'll actually have to convert it. Converting in the other direction comes up in one case, and I'm going to show you this one.

It has to do with concatenation. The plus operator that does concatenation really wants to work on C++ style strings, so if one of your operands is a C++ string, it's all fine, as long as the left or the right side is a C++ string. The other side can be a string literal, a constant, another string, a character variable – all those things work fine. As long as at least one of the operands really is a true C++ string already, you're good.

That's almost always the case. But in the case where you somehow have two things on either side of the plus, neither of which is already a C++ string – typically, that means you have a C string on one or both sides, a character on one or both sides – you are not going to get concatenation. If you try to add two C style strings, it actually won't compile.

The sad thing about these two things, about taking a string literal and adding either a character constant or a character variable is that it does compile and it just does not do what you want at all. It does so in a silent but deadly way. I'm not going to tell you what it does, but if you are curious, you can come and talk to me and I'll lay it out for you. What I want you to come away with is this memory that when I'm using concatenation to be sure that one of the two operands is a C++ string. If you have to, force one. If you have a string literal and you want it to be a C++ string, then make it one to avoid running into this.

The mistake that you get from this is actually quite mystical and very confusing. Probably 95 percent of you would never run into this, and so mostly, I've just confused you for reasons that seem unclear, but for the five percent that are going to run into this, I'm really trying to do you a favor by giving you a heads up before it causes you a lot of grief later. Just a little bit of a legacy. C and C++ go back a long way, and as a result, we sometimes have little quirks we have to deal with even in the modern world.

Student: Is that the only way you can convert a C string into a C++ string?

Instructor (Julie Zelenski):Not exactly. A lot of times, it just happens automatically is the truth. In almost all situations – if you had a routine that expected a string argument and you passed the string literal, it will automatically convert. Mostly, you won't need to do this is the truth. It happens all the time behind the scenes without any effort on your part. This is the official way to say I've got a string and I really want to force it and I'm not waiting for the compiler to do it on my behalf.

In particularly, for example, in a situation like this, it doesn't realize what you really wanted to do was convert this and then to concatenation. It does something kind of goofy based on what the old meaning of taking a C string and adding a character to it was, which was not concatenation. That's a little moment of silence for old language C that comes back to haunt us a little bit like a ghost in the attic.

Student: What do you mean by a string literal?

Instructor (Julie Zelenski):A string literal just means a string constant. It's something in quotes.

Student: Without explicitly declaring it to be a string.

Instructor (Julie Zelenski):Yeah. A string literal is when you see open double quotes, some characters, close quote, that's a string constant or a string literal. In any situation where you see exactly that – not a string variable is basically what I'm saying there.

How do we do IO? How do we do input/output in C++? Let me first say that input/output is probably one of the more distinctive features of any language. C's IO, for example, looks very different than C++'s IO, which looks kind of different from Java's IO. These are areas where for some reason, even though they all do the same things underneath it all – they let you print stuff. They let you read stuff. They have some formatting features.

For one reason or another, these are the areas where they're widely divergent in their syntax and the way you express what you want to do. That makes them particularly annoying to learn is the truth. I know a lot of IO systems, and they're all very jumbled up in my head. At any given moment if you asked me how could I print a decimal number with three digits of precision in this language, I'm going to have to go look it up. My motto is look it up. Don't worry about memorizing these details, because they are very tied to any particular language and its formatting system.

That said, we're going to use a little bit of IO. We'll need to be able to read and write things to the console to interact with the user. We're going to do a little bit of file reading, reading numbers and strings from files, maybe even producing some files. We're going to use some very simple set of features. We're not going to go too deep. When you need to know more, there are great resources to go check into for that. I wouldn't in advance go make yourself an expert on any form of IO. Figure it out when you need to.

The IOs are actually handled in C++ using stream objects. There are stream classes. The O stream is the output stream that's used for writing. The I stream are the classes used for reading. Their variance, for example – the IF stream and the OF stream are the file equivalents of the input/output streams. Cout and cin are these two basically global variables, effectively, that give you access to the console output stream and cin for the console input stream. That means the little text window that pops up that you get to type and print things for the user to see and interact with.

The standard operators for reading and writing to the stream in the default sense are the <<, which is stream insertion, and >>, which is stream extraction. You stick things onto a stream and then retrieve things back from a stream that you're reading from. A very simple example of this would be I have the variables X and Y declared here. I asked the user to enter two numbers, and then I use extraction that says from the console input stream to pull an integer out followed by another integer, and then I repeat back what they said.

In its simplest form, the kind of things you can print out are very related to things you can read in. When I ask cin to read an integer here, it looks for a sequence of digits upcoming in the stream that form a valid integer which it assembles and puts into the value X. Then it looks for another one. It typically uses white spaces as an eliminator, so any returns, tabs or spaces will be skipped over in between. Anything that led up to X, it will skip over all the white space, look for some digits and then skip over any intervening white space, and look for some more digits to pull Y.

Of course, what's likely to happen here is users are bad typists. They make mistakes that when I go to read this, what happens if they've typed the letter A or 72A45. This causes a little bit of havoc because when it goes to extract that, it looks for some digits and it finds this thing and it doesn't match its expectation. That puts the stream in what is called an error or fail state, which then requires you digging around, realizing it went into a fail state, cleaning it up and resetting and starting over.

It's not that it can't be done, but it's a little bit annoying. We just made this task a little bit less onerous by providing in the simpIO library, which is our CS106 simple IO – it has get integer, get real and get line. They all read from the console, so reading from cin, and they deal with all that error handling. They make sure that the input given was well formed. If it's not, it reprompts and has them try again. It does that until they get an integer.

When you call get integer, you know that eventually, the user will have typed in a well-formed integer and you will get that value back when you make that call. You don't have to be worried about all the machinations to check for errors. Retry is actually bundled up behind that routine for you. Most of our console input will end up using these functions just for convenience. They save us a certain amount of hassle.

I would ask – if I wanted X and Y, I would say get integer one, get integer two. I'd have to call it twice. There's not a combined form of it. It saves us a lot of trouble. I can't do it

this way. I'd have to stop it after one anyway, check to see if it failed, if not, go back in. It's kind of misleading to even show this form, because that form assumes that the user is a perfect typist and never makes mistakes, which is in this day and age not too likely.

We'll talk more about file streams on Wednesday. On your way out, look for handout five and seven for Mac or PC depending on what you're using and good luck getting your compiler set up. I'll see you on Wednesday.

[End of Audio]

Duration: 48 minutes

ProgrammingAbstractions-Lecture04

Instructor (Julie Zelenski):Hey, good afternoon. Good afternoon. I've gotten a couple emails, which means that there's actually been compiler success, but I've also gotten a couple emails about compiler failures, so just a reminder that one of the tasks that you need to accomplish this week is getting the compiler installed where you're going to be working.

If it's in the clusters, that's actually been done for you, but if you're doing it on your own machine there's just a surprisingly vast array of things that can go wrong along the way that you don't want to be wrestling with at the last minute, so certainly try to get on that soon and then getting in touch with us if you run into some snags so that we can help you get past that and move on to running the code that you need read this week.

We posted a couple announcements on the webpage, one for X-Code and one for Visual Studio, and we will continue to do any updates that we think will be beneficial to the whole class. So do keep an eye on that and look for things that are coming out so that you can avoid things that we already know about and not spend too much time on them.

Sections start this week, but your Section emails should have been emailed to you yesterday, so you should know when and where your Section's meeting, who your Section Leader is. If you have – something's changed, and that makes that completely impossible to work for you we have kind of a limited ability to get you switched into something else, but it's a little bit dependent on where we have space.

So if you need something like that, get back to the sign-up page, there's an add/late option, and that will coordinate with the remaining spaces to get you someplace that works better for you. Today's topic is one of – we're going to talk more about [inaudible] I started that at the beginning of Monday and didn't get very far into it, so I'm gonna go through a little bit more of that today.

And then I'm going to start talking about CS106 class libraries, some of the facilities we're going to be using all quarter long, we're going to introduce today and Friday to get somewhere with those things. Much of this is just covered in the handout. Handout 14, which went out today is this big 20 something page handout, is the kind of reading material for the next two lectures, and then after that we will go back to start reading in the text.

We'll be picking up with Chapter 4 next week after the holiday. Any administration questions on your mind? How many people have actually successfully installed a compiler? Have stuff working – okay, so that's like a third of you, good to know. Remaining two thirds, you want to get on it.

Okay, so we started to talk about this on Monday, and I'm gonna try to finish off the things that I had started to get you thinking about; about how input/output works in C++.

We've seen the simple forms of using stream insertion, the less than less than operator to push things on to cout, the Console Output Stream.

A C-Out is capable of writing all the basic types that are built into C++, ants and doubles and cars and strings, right, by virtue of just sort of putting the string on the left and the thing you want on the right, it will kind of take that thing and push it out onto stream.

You can chain those together with lots and lots of those < < to get a whole bunch of things, and then the endl is the – what's called stream manipulator that produces a new line, starts the next line of text, a line beneath that. The analog to that on the reading side is the stream extraction operator, which is the > >.

And then when applied to an input stream it attempts to sort of take where the cursor position is in the input stream and read the next characters using the expected format given by the type of the thing you're trying to extract. So in this case what I'm saying, CN > > extract an integer here, X being an integer. What it's gonna look for in the input stream is it's going to skip over white space. So by default the stream extraction always skips over any leading white space.

That means tabs, new lines, and ordinary space characters. So scans up to that, gets to the first non-space character and then starts assuming that what should be there is a number, and so number being, a sequence of digit characters. And in this case, because it's integer, it shouldn't have a dot or any of the exponentiations sort of things that a real number would.

If it runs into something that's not integer, it runs into a character, it runs into a punctuation, it runs into a 39.5, what happens is that the screen goes into a fail state where it says, I – you told me to expect an integer. What I read next wasn't an integer. I don't know how to make heads or tails of this. So it basically just throws up its hand.

And so it – at that point the stream is – it requires you to kind of intervene, check the fail state, see that something's wrong, clear that fail state, decide what to do about it, kind of restart, and kind of pick up where you left off. It makes for kind of messy handling to have all that code kind of in your face when you're trying to do that reading, and that's actually why we've provided the things like get integer, get line and get wheel, and the simple I/O library to just manage that for you.

Basically what they're doing is in a loop they're trying to read that integer off the console. And if it fails, write resetting the stream, going back around asking the user to type in – give it another try, until they get something that's well formed. So typically we're just going to use these, because they just provide conveniences.

You could certainly use this, but it would just require more effort on your part to kind of manage the error conditions and retry and whatnot. So that's why it's there. The C++ file I/O; so the console is actually just a particular instance of the stream. Cout and cin are the string that's attached to the users interface console there.

That the same sort of mechanism is used to read files on disks, so text files on disks that have contents you like to pull into a database, or you want to write some information out to a file, you use the file stream for that. There is a header called fstream, standard C++ header in this case, so enclosed in < >, that declares the isstream and the osstream.

The input file stream for reading, the output file stream for writing. Declaring these variables; this [inaudible] just sets up a default stream that is not connected to anything on disc. Before you do anything with it you really do need to attach it to some named location, some file by name on your disk to have the right thing happen, to read from some contents, or to write the contents somewhere.

The operation that does that is open, so the isstream and the osstream are objects, so dot notation is used to send messages to it. In this case, telling the input stream to open the file whose name is "names.txt." The behavior for open is to assume that you meant the file in the current directory if you don't otherwise give a more fully specified path.

So this is almost always the way we're going to do this, we're just going to open a file by name. It's going to look for it in the project directory, where your code is, where you project is, so kind of right there locally. Now this will look for a file whose name is exactly names.txt, and then from that point the file positions, the kind of cursor we call it, is positioned at the beginning of the input stream.

The first character read will be the first character of names.txt, and as you move forward it will read its way all the way to the end. Similarly, doing an outopen, it opens a file and kind of positions the writing at the very beginning that will – the first character written will be the first character then when you finish.

And that file, they'll be written in sequence. So this is one of those places, actually, probably the only one that this direction is going to be relevant for. I talked a little bit last time about C-strings and C++ strings, and you might have been a little bit worried about why I'm telling you you need to know that both exist.

And so last time I talked a little about one way in which C-strings don't do what you think, in that one case of concatenation, and how you can do a – force a conversion from the old to the new. Now, I also mentioned that there was a conversion that went in the opposite direction. You had a new string, and you wanted the old one.

And one of the first questions you might ask is well why would I ever want to do that? Why would I ever want to go backwards? Why do I want to move back to the older yucky thing? This is the case that comes up; the open operation on isstream and osstream expects its argument to be specified as an old style string. This is actually just an artifact; it has to do with it – the group that was working on designing the string package. The group that was designing the string package were not in sync, and they were not working together.

The string package was finalized before the string package was ready and so it depended on what was available at the time and that was only the old style string. So as a result, it wants an old style string, and that's what it takes, and you can't give it a C++ string. So in double quotes – so this is the case where the double quotes are actually old style strings, in almost all situations gets converted on your behalf automatically. In this case it's not being converted and it's exactly what's wanted.

So if you have a name that's a string constant or a literal, you can just pass it in double quotes to open. If you have a C++ variable, so you've asked the user for what file to open, and you've used getline to read it into a string, if you try to pass that C++ string variable to open, it will not match what it's expecting.

I do need to do that conversion asking it to go .c_str to convert itself into the old style format. So that was sort of where I was getting to when I kind of positioned you to realize this was gonna someday come up. This is the one piece of the interface that will interact with this quarter that requires that old string, where you'll have to make that effort to convert it backwards.

Both of these operations can fail. When you open a file and [inaudible] – question here?

Student: So how hard [inaudible]?

Instructor (Julie Zelenski):You know it's obviously extremely easy to do it; the issue has to do with compatibility. They announced it this way, people wrote code that expected it this way and then you change it out from under them and all this code breaks that used to work. And so as a result of this [inaudible] compatibility an issue of once we kind of published it and we told people this was how it works, we can't really take it away from them.

And so part of that's – sort of part of what we're doing within C++2, which is things that used to work in C still need to work in C, and so as a result there's a certain amount of history that we're all carrying forward with us in a very annoying way. I totally agree that it seems like we could just fix it, but we would break a lot of code in the process and anger a lot of existing programmers.

So both of these open calls could fail; you might be able to – try to open a file and it doesn't exist, you don't have the permissions for it, you spelled the name wrong. Similarly trying to open it for writing, it's like you might not have write permission in the directory. And in either situation you need to know, well did it open or did it not?

There's not a return value from open that tells you that. What there is is a member function called .fail, that you can ask the stream at any point, are you in a fail state. So for operations that actually kinda have a chance of succeeding or failing in the string, you'll tend to actually almost write the code as a try it then check in .sale. So try to read this thing, check in .sale. Try to open this file check in .sale as your way of following up on did it work and making sure that you have good contents before you keep going.

If the in .open has failed, then every subsequent read on it will fail. Once the string is in a fail state, nothing works. You can't read or write or do anything with it until you fix the error, and that's the in .clear command that kind of resets the state back into a known good state, and then you have a chance to retry.

So for example, if you were trying to open a file that the user gave you a name for, they might type the name wrong. So you could try in .openit, check in .dot fail. If it failed, say no, no, I couldn't open that file, why don't you try again, get a new name, and then you'd clear the state, come back around and try another in .open to – until you get one that succeeds.

Once you have one of those guys open for reading or writing, there are three main ways that you can do your input/output. We have seen this form a little bit, this one with the insertion/extraction, these other two are more likely to be useful in the file reading state as opposed to interacting with the user state, and they have to deal with just breaking down the input more fine graindly.

Let's say this first one is reading and writing single characters. It might be that all I want to do is just go through the file and read it character by character. Maybe what I'm trying to write is something that will just count the characters and produce a frequency count across the file, tell me how many A's and B's and C's are in it, or just tell me how many characters are in the file at all.

In .get is the number function that you send to an input file stream that will retrieve the next character. If [inaudible] the next character from the stream it returns EOF when there are no more characters. EOF is the end of file marker, it's actually capital EOF, it's the constant that's defined with the class. And so you could read till EOF as a way of just getting them character by character.

Similarly there is a put on the other side, which is when you're writing, do you just want to write a single character. You could also do this with out << ch, which writes the character. This actually just does a put of the character, just kind of a matching function in the analog to get input that do single character io.

Sometimes what you're trying to do is process it line by line. Each line is the name of somebody and you're kind of putting those names into a database. You don't want to just assemble the characters by characters, and you don't know how many tokens there might be, that the white space might be that there's Julie Diane Zelenski, sometimes there might be Julie Zelenski, you don't know how many name pieces might appear to be there.

You can use getline to read an entire line in one chuck. So it'll read everything up to the first new line character it finds. It actually discards the new line and advances past it. So what you will get is – the sequence of characters that you will have read will be everything up to and not including the new line. The new line will be consumed though so that reading will pick up on the next line and go forward.

Getline is a free function. It is not a member function on the stream. It takes a stream as its first argument. It takes a string by reference as its second argument, and it fills in the line with the text of the characters from here to the next line read in the file.

If it fails the way you will find out is by checking the fail states. You can do a getline inline and then in .fail after it to see, well did it write something in the line that was valid? If it failed, then the contents of line are unchanged, so they'll be whatever nonsense they were. So it's a way of just pulling it line by line.

This name has the same words in it as rgetlineGL in the sympio, which shows that it's kind of a reasonable name for the kind of thing that reads line by line, but there is a different arrangement to how it's – what it's used for and how it's it used. So rgetline takes no arguments and returns a line read for the console. The lower case getline takes the file stream to read from and the string to write it into and does not have a return value.

You check in .fail if you want to know how it went. So write the entire line out there, [inaudible] a put line equivalence, so in fact you could just use the out stream insertion here, stick that line back out with an nline to kind of reproduce the same line your just read.

And then these we've talked a little about, this idea of formatted read and write, where it's expecting things by format. It's expecting to see a character, it's expecting to see an integer, and it's expecting to see a string. It uses white space as the default delimiter between those things.

So it's kind of scanning over white space and discarding it and then trying to pull the next thing out. These are definitely much trickier to use because if the format that you're expecting doesn't show up, it causes the stream to get new fail state, and you have to kind of fix it and recreate it.

So often even when you expect that things are going to be, let's say, a sequence of numbers or a name fall by number, you might instead choose to pull it as a string and then use operations on the string itself to kinda divide it up rather than depending on stream io because stream io is just a little bit harder to get that same effect.

And then in all these cases write in .fail. There is also – you could check out.fail. It's just much less common that the writing will fail, so you don't see it as much, but it is true for example, if you had wanted a disk space and you were writing, a write operation could fail because it had wanted a space or some media error had happened on the disk, so both of those have reasons to check fail.

So let me do just a little bit of live coding to show you that I – it works the way I'm telling you. Yeah?

Student: So the fail function, is it going to always be the stream that's failing and not the function that's failing?

Instructor (Julie Zelenski):Yes, pretty much. There are a couple rare cases where the function actually also tells you a little bit about it, but a general fail just covers the whole general case of anything I have just got on the stream fail so any of the operations that could potentially run into some error condition will set the fail in such a way that your next call to in .fail will tell you about it.

And so that's the – the general model will be; make the call, check the fail, if you know that there was a chance that something could have gone wrong and then you want to clean up after it and do something [inaudible].

So I'm gonna show you that I'm gonna get the name of the file from the user here, I'm going to use in .open of that, and I'm going to show you the error that you're gonna get when you forget to convert it, while I'm at it. And then I'll have like an in .fail error wouldn't – file didn't open. First I just want to show you this little simple stuff; I've got my ifstream declared, my attempt to open it and then my check for seeing that it failed.

I'm gonna anticipate the fact that the compiler's gonna be complaining about the fact that it hasn't heard about fstream, so I'm gonna tell it about fstream. And I'm gonna let this go ahead in compiling, although I know it has an error in it, because I want to show you sort of the things that are happening.

So the first thing it's complaining about actually is this one, which is the fact that getline is not declared in the scope, which meant I forgot one more of my headers that I wanted. Let me move this up a little bit because it's sitting down a little far. And then the second thing it's complaining about is right here.

This is pretty hard to see, but I'll read it to you so you can tell what it says; it says error, there's no matching function call and then it has sort of some gobbly gook that's a little bit scary, but includes the name ifstream. It's actually – the full name for ifstream is a lot bigger than you think, but it's saying that there's – the ifstream is open, and it says that it does not have a match to that, that there is no open call on the ifstream class, so no member function of the ifstream class whose name is open, whose argument is a string.

And so that cryptic little bit of information is gonna be your reminder to jog your memory about the fact that open doesn't deal in the new string world, it wants the old string world. It will not take a new string, and I will convert it to my old string, and then be able to get this thing compiling.

And so when it runs if I enter a file name of I say [inaudible], it'll say error file didn't open, some file that I don't have access for. It happens that I have one sitting here, I think, whose name is handout.txt. I took the text of some handout and then I just left it there. So let me doing something with that file. Let's just do something simple where we just count the number of lines in it. Let's say – actually I'll make a little function that – just to talk a little bit about one of the things that's a little quirky about ifstreams is that when you pass an ifstream you will typically want to do so by reference.

Not only is this kind of a good idea, because the ifstream is kind of changing in the process of being read. It's updating its internal state and you want to be sure that we're not missing this update that's going on. It's also the case that most libraries require you to pass it by reference. That it doesn't have a model for how to take a copy of a stream and make another copy that's distinct. That it really is always referring to the same file, so in fact in most libraries you have to pass it by reference.

So I'll go ahead and pass it by reference. I'm gonna go in here and I'm just gonna do a line-by-line read and count as I go. I'm gonna write this as a wild [inaudible], and I'm gonna say read the next line from the file into the variable, and then if in .fail – so if it was unable to read another line, the – my assumption here is gonna be that we're done, so it will fail as eof . It's the most common reason it could fail. It could also fail if there was some sort of more catastrophic error, you're leading a file from a network and the network's gone down or something like that.

In our case its right, the in .fail is going to tell us yeah, there's nothing more to read from this file, which means we've gotten to the end. We've advanced the count. Whenever we get a good line we go back around, so we're using kind of the wild true in this case because we have a little bit of work to do before we're ready to decide whether to keep going, in this case, reading that line.

And then I return the count at the end, and then I can then down here print it nom lines = mi call to count lines of n and l. Okay. Let me move that up a little bit. Last time I posted the code that I wrote in the editor here, and I'll be happy to do that again today, so you shouldn't need to worry about copying it down, I will post it later if you want to have a copy of it for your records, but just showing, okay, yeah, we're just a line by line read, counting, and then a little bit more of the how do you open something, how do you check for failure.

And when I put this together, what does it complain about? Well I think it complains about the fact that I told it my function returned void, but then I made it return it. And that should be okay now. And so if I read the handout.txt file, the number of lines in it happens to be 28. It's just some text I'd cut out of the handout, so there are 28 new line characters is basically what it's telling me there.

So I can just do more things, like I could use – change this loop and instead use like get to do a single character count. I could say how many characters were in there. If I used the tokenization and I said, well just tell how many strings I find using string extraction, it would kind of count the number of non-space things that it found and things like that.

Typically the IO is one of those errors I said where there's like a vast array of nuances to all the different things you can do with it, but the simple things actually are usually fairly easy, and those are the only ones that really going to matter to us as being able to do a little bit of simple reading and file reading/writing to get information into our programs.

How do you feel about that? Question?

Student: Sorry, why do have getline an empty string?

Instructor (Julie Zelenski):So getline, the one that was down here? This one?

Student: No, the one that –

Instructor (Julie Zelenski):Oh, the one that's up here. So yeah, let's talk about that. The getline that's here is – the second argument to getline is being passed by reference, and so it's filling in that line with the information it read from the file. So I just declared the variable so I had a place to store it and I said, okay, read the next line from the file, store the thing you read in the line.

It turns out I don't actually care about that information, but there's no way to tell getline to just throw it away anyway.

Student: Oh.

Instructor (Julie Zelenski):So I'm using it to just kinda move through line-by-line, but it happens to be that getline requires me to store the answer somewhere, and I'm storing it. Instead of returning it, it happens to use the design where it fills it in by reference. There's actually – it turns out to be a little bit more efficient to do a pass by reference and fill something in, then to return it. And the C++ libraries in general prefer that style of getting information back out of a function as opposed to the function return, which you think of as being a little more natural design.

There's a slight inefficiency to that relative to the pass by reference and the libraries tend to be very hyper-conscious of that efficiency, so they tend to prefer this slightly more awkward style. Question?

Student: Why in the main [inaudible] does the error open [inaudible] file didn't open with [inaudible] like print error: file didn't open?

Instructor (Julie Zelenski):You know it's just the way that error works. Error wants to make sure that you don't mistake what it does, and so it actually prefixes whatever you ask it to write with this big ERROR in uppercase letters, and so the purpose of error is twofold; is to report what happened and to halt processing.

And so when it reports that it actually prefixes it with this big red E-R-R-O-R just to say don't miss this, and then it halts processing there. And it's just – the error [inaudible] libraries function, which is your way of handling any kind of catastrophic I can't recover from this. And it's certainly something we don't want anybody to overlook, and so we try to make it really jump out at you when it tells you that.

Student: So this is in symbio?

Instructor (Julie Zelenski):It is in genlib actually.

Student: Oh.

Instructor (Julie Zelenski):So error's actually declared out of genlib.

Student: And can we use it – so it's global basically?

Instructor (Julie Zelenski):It is global. It's a telefree function, and you will definitely have occasion to use it. Right, it's just – it's your way of saying something happened that there's just no recovery from and continuing on would not make sense. Here's a – stop and help and alert the user something's really wrong, so you don't want to keep going after this because there's no way to kind of patch things back together.

In this case probably a more likely thing we'd do, is I should say give me another name, let's go back around and try again, would be a sort of better way to handle that. I can even show you how I would do that. I could say, well while true, enter the name, and maybe I could change this to be well if it didn't fail then go ahead and break out of the loop. Otherwise, just report that the file didn't open, and say try again.

And then the last thing I will need to do is clear that state. So now it's prompting, trying to open it. If it didn't fail it will break and then it will move forward to counting the lines. If it did fail it'll continue on through here reporting this message, and then that clear, very important, because that clear kind of gets us back in the state where we can try again.

If we don't clear the error and we try to do another in .open, once the string is in a fail state it stays in a fail state until you clear it, and no subsequent operation will work whatsoever. It's just ignoring everything you ask it to do until you have acknowledged you have done something about the problem, which in this case was as simple as clearing and asking to open again.

So if I do it this way I enter some name it'll say that didn't open, try again. And then if I say handout.txt, it'll open it and go ahead and read. All right, any questions about iostreams? We're gonna move away from this [inaudible], if there's anything about it you'd like to know I'd be happy to answer it.

So let me get us back to our slides, and I'll kind of move on to the more object-oriented features of the things we're going to be depending on and using this quarter. So the libraries that we have been looking at, many of them are just provided as what we call free functions. Global functions that aren't assigned to a particular object, they are part of a class, so asking for random integer, reading a line, competing the square root, gobs of things are there that just kind of have functionality that you can use anywhere and everywhere procedurally.

We've just started to see some things that are provided in terms of classes, the string of the class, that means that you have string objects that you're messaging and having them manipulate themselves. The stream object also is class, ifstream, ofstream, those are all

classes that you send messages like open to and fail to, to ask about that streams state or reset its state.

This idea of a class is one that's hopefully not new to you. Most of you are coming from Java have – this is pretty much the only mechanism for writing code for Java is in the context of a class. Those of you who haven't seen that as much, we're going to definitely be practicing on this in our – some simple things you need to know to kind of just get up to the vocabulary wise is class is just a way of taking a set of fields or data and attaching operations to it to where it kind of creates a kind of an entity that has both its state and its functionality kind of packaged together.

So in the class interface you'll say here is a time object, and a time object has an hour and a minute and you can do things like tell me if this time's before that time or what the duration starting at this time and this end time would – there would be all these behaviors that are like [inaudible] to do. Can you print a time, sure. Can I read a time for a file, sure.

As long as the interface for the time class provides those things, its kinda this fully flip – fleshed out new data type that then you use time objects of whenever you need to work with time. The idea is that the client use the object, which is the first role we're gonna be in for a couple weeks here, is you learn what the abstraction is. What does the class provide? It provides the notion of a sequence of characters, that's what stream does.

And so that sequence has all these operations; like well tell me what characters are at this position, or find this sub-string, or insert these characters, remove those characters. And internally it's obviously doing some machinations to keep track of what you asked it to do and how to update its internal state.

But what's neat is that from the outside as a client you just think well there's a sequence of characters there and I can ask that sequence of characters to do these operations, and it does what I ask, and that I don't need to know how it's implemented internally. What mechanisms it uses and how it responds to those things to update it state is very much kind of behind the abstraction or inside that black box, sometime we'll call it to kind of suggest to ourselves that we can't see inside of it, we don't know how it works. It's like the microwave, you go up and you punch on the microwave and you say cook for a minute. Like what does the microwave do? I don't know, I have no idea, but things get hot, that's what I know. So the nice thing about [inaudible] is you can say, yeah, if you push this button things get hot and that's what I need to know. [Inaudible] has become widely industry standard in sort of all existing languages that are out there. It seems like there's been somebody who's gone to the trouble of trying to extend it to add these object [inaudible] features and languages like Java that are fully object oriented, are very much all the rage now. And I thought it was interesting to take just a minute to talk about well why is it so successful? Why is object oriented like the next big thing in programming? And there are some really good valid reasons for why it is a very sensible approach to writing programs that is worth thinking a little bit about. Probably the largest sort of motivation for the industry has to do with this idea of taming complexity that certainly

one of the weaknesses of ourself as a discipline is that the complexity kinda can quickly spiral out of control. The programs that – as they get larger and larger, their interactions get harder and harder to model and we have more and more issues where we have bugs and security flaws and viruses and whatnot that exploit holes in these things. That we need a way as engineers to kind of tighten down our discipline and really produce things that actually don't have those kind of holes in them. And that object oriented probably means one of the ways to try to manage the complexities of systems. That instead of having lots and lots of code that [inaudible] things, if you can break it down into these objects, and each class that represents that object can be designed and tested and worked on independently, there's some hope that you can have a team of programmers working together, each managing their own classes and have them be able to not interfere with each other too much to kind of accomplish – get the whole end result done by having people collaborate, but without them kind of stepping on top of each other. It has a – the advantage of modeling the real world, that we tend to talk to talk about classes that kind of have names that speak to us, what's a ballot, what's a class list, what's a database, what is a time, a string, that – a fraction? These things kind of – we have ideas about what those things are in the real world, and having the class model that abstraction makes it easier to understand what the code is doing and what that objects role is in solving the problem.

It also has the advantage of [inaudible] use. That once you build a class and it's operations, the idea is that it can be pulled out of the – neatly out of the one program and used in another if the design has been done, and can be changed extended fairly easily in the future if the design was good to begin with.

So let me tell you what kind of things we're going to be doing in our class library that will help you to kind of just become a big fan of having a bunch of pre-written classes around. We have, I think, seven classes – I think there's eight actually in our class library that just look at certain problems that either C++ provides in a way that's not as convenient for us, or is kind of missing, or that can be improved on where we've tackled those things and given you seven classes that you just get to use from the get go that solve problems that are likely to come up for you.

One of them is the scanner, which I kind of separated by itself because it's a little bit of an unusual class, and then there's a bunch of container classes on that next line, the vector grid, staque, math and set that are used for storing data, different kinds of collections, and they differ in kind of what their usage pattern is and what they're storing, how they're storing it for you.

But that most programs need to do stuff like this, need to store some kind of collection of date, why not have some good tools to do it. These tools kinda let you live higher on the food chain. They're very efficient, they're debugged, they're commented, the abstraction's been thought about and kind of worked out and so they provide kinda this very useful piece of function [inaudible] kinda written to you ready to go.

And then I – a little note here is that we study these – we are going to study these abstractions twice. We're gonna look at these seven classes today and Friday as a client, and then start using them all through the quarter. In about a week or so after the mid-term we're gonna come back to them and say, well how are they implemented?

That after having used them and appreciated what they provided to you, it will be interesting, I think, to open up the hood and look down in there and see how they work. I think this is – there is an interesting pedagogical debate going on about this, about whether it's better to first know how to implement these things and then get to use them, or to use them and then later know how to implement them.

And I liken it to a little bit if you think about some things we do very clearly one way or the other in our curriculum, and it's interesting to think about why. That when you learn, for example, arithmetic as a primary schooler, they don't give you a calculator and say, here, go do some division and multiplication, and then later try to teach you long division. You'll never do it.

You'll be like, why would I ever do this, this little box does it for me, the black box. So in fact they drill you on your multiplication tables and your long division long before they let you touch a calculator, which I think is one way of doing it.

And, so – and for example, it's like we could do that with you, make you do it the kind of painful way and then later say, okay, well here's these way you can avoid being bogged down by that tedium. On the other had, think about the way we teach you to drive.

We do not say, here's a wheel and then they say, let me tell you a little bit about the combustion engine, you know, we give you some spark plugs and try to get you to build your car from the ground up. It's like you learn to drive and then if you are more interested in that you might learn what's under the hood, how to take care of your car, and eventually how to do more serious repairs or design of your own care.

Where I think of that as being a client first model, like you learn how to use the car and drive and get places and then if it intrigues you, you can dig further to learn more about how the car works. So that's definitely – our model is more of the drive one than the arithmetic one that it's really nice to be able to drive places first.

Like if I – we spent all quarter learning how to build a combustion engine and you didn't get to go anywhere, I'd feel like you wouldn't have tasted what – where you're trying to get, and why that's so fabulous.

So we will see them first as a client, and you'll get to do really neat things. You'll discover this thing called the map where you can put thousands, millions of entries in and have instantaneous look-up access on that. That you can put these things in a stack or a queue and then have them maintained for you and popped back out and all the storage of that being managed and the safety of that being managed without you having to kinda take any active role in that.

That they provide functionality to you, that you just get to – leverage from the get go, and hopefully it will cause you to be curious though, like how does it work, why does it work so well, and what kind of things must happen behind the scenes and under the hood so that when we get to that you're actually kind of inspired to know how it did it, what it did.

So I'm gonna tell you about the scanner and maybe even tell you a little bit about the vector today, and then we'll do the remaining ones on Friday, perhaps even carrying over a little bit into the weeks to get ourselves used to what we've got.

The scanner I kind of separated because the scanner's more of a task based object then it is a collection or a container for storing things. The scanner's job is to break apart input into tokens. To take a string in this case that either you read from the file or you got from the user, or you constructed some way, and just tokenize it.

It's called tokenizer parsec. That this is something a little bit like – strained extraction kind of does this, but strained extraction, as I said, isn't very flexible, that it doesn't make it easy for you to kind of – you have to sort of fully anticipate what's coming up on the string. There's not anyway you can sort of take a look at it and then to decide what to do with it and decide how to change your parstring strategy.

And scanner has a kind of flexibility that lets it be a little bit more configurable about what you expect coming up and how it works. So the idea is that basically it just takes your input, you know, this line contains ten tokens, and as you go into a loop saying, give me the next token, it will sub-string out and return to you this four character string followed by this single character space and then this four character line and space, and so the default behavior is to extract all the tokens to come up, to use white-space and punctuation as delimiters.

So it will kind of aggregate letters and numbers together and then individual spaces and new lines and tabs will come out as single character tokens. The parenthesis and dots and number signs would all come out as single character tokens, and it just kind of divides it up for you.

Okay. It has fancy options though that let you do things like discard those face tokens because you don't care about them. To do things like read the fancy number formats. So it can read integer formats and real formats, it can do the real format with exponentiation in it with leading minus', things like that, that you can control with these setters and getters, like what it is you wanted to do about those things.

You can it things like when I see an opening quote, I want you to gather everything to the closing quote, and so it does kind of gather phrases out of sequence if that's what you want. And so you have control over when and where it decides to do those things that lets you kind of handle a variety of kind of parsing and dividing tasks by using the scanner to get that job done.

So I listed some things you might need, if you're reading txt files, you're parsing expressions, you were processing some kind of commands, that this scanner is a very handy way to just divide that [inaudible] up.

You could certainly do this kind of stuff manually, for example, like using the find on the string and finding those faces and dividing it up, but that the idea is just doing that in a more convenient way for you than you having to handle that process manually.

This is what its interface looks like. So this is a C++ class definition. It looks very similar to a Java class definition, but there's a little bit of variation in some of the ways the syntax comes through in the class. The class being here is scanner, the public colon introduces a sequence of where everything from here until the next access modifier is public. So I don't actually have public repeated again and again on all the individual entries here.

It tells us that the scanner has a constructor that takes no arguments; it just initializes a new empty scanner. I'm gonna skip the destructor for a second; I'll come back to it. There is a set input member function that you give it the string that you want scanned and then there's these two operations that tend to be used in a look where you keep asking are there more tokens and if so, give me the next token, so it just kind of pulls them out one by one.

I picked just one of the space – of the particular advanced options to show you the format for them. There's actually about six more that deal with some other more obscure things. This one is how is it you'd like it to deal with spaces, when you see face tokens, should they be returned as ordinary tokens or should you just discard them entirely and not even bother with them?

The default is what's called preserve spaces, so it really does return them, so if you ask and there's only spaces left in the file, it will say there are more tokens and as you call the next token we'll return those spaces as individual tokens.

If you instead have set the space option of ignore spaces, then it will just skip over all of those, and if all that was left in the file was white space when you ask for more tokens, it will say no. And when you ask for a token and there's some spaces leading up to something it will just skip right over those and return the next non-space token.

There's a variety of these other ones that exist that handle the floating point and the double quote and other kind of fancy behaviors. There's one little detail I'll show you that's a C++ ism that isn't – doesn't really have a Java analog, which is the constructor which is used as the initialization function for a class has a corresponding destructor.

Every class has the option of doing this. That is the – kind of when the object is being created, the constructor is being called. When the object is being de-allocated or destroyed, going out of scope, the destructor is called. And the pairing allows sort of the

constructor to do any kind of set up that needs to be done and the destructor to do any kind of tear down that needs to be done.

In most cases there's not that much that needs to be there, but it is part of the mechanism that allows all classes to have an option kind of at birth and death to do what it needs to do. For example, my file stream object, when you – when it goes away, closes it file automatically.

So it's a place where the destructor gets used to do cleanup as that object is no longer valid. So a little bit of scanner code showing kind of the most common access pattern, is you declare the scanner. So at this point the scanner is empty, it has no contents to scan.

Before I start pulling stuff out of it, I'm typically gonna call a set input on it, passing some string. In this case the string I'm passing is the one that was entered by the user, using getline. And then the ubiquitous loop that says well while the scanner has more tokens, get the next token.

And in this case I'm not even actually paying attention to what those tokens are, I'm just counting them. So this one is kind of a very simple access that just says just call the next token as many times as you can until there are no more tokens to pull out. Way in the back?

Student:[Inaudible] I mean, like in the beginning when it says scanner, scanner, do we write scanner scanner = new scanner () or [inaudible]?

Instructor (Julie Zelenski):Yes. Not exactly. So that's a very good example of like where Java and C++ are gonna conspire to trip you up just a little bit, that in Java objects were always printed using the syntax of new. You say new this thing, and in fact that actually does an allocation out in what's called the heap of that object and then from there you use it.

In C++ you actually don't have to put things in the heap, and in fact we will rarely put things in the heap, and that's what new is for. So we're gonna use the stack to allocate them. So when I say scanner scanner, that really declares a scanner object right there and in this case there are no [inaudible] my constructor, so I don't have anything in parenths.

If there were some arguments I would put parenths and put the information there, but the constructor is being called even with out this new. New actually is more about where the memory comes from. The constructor is called regardless of where the memory came from.

And so this is the mechanism of C++ to get yourself an object tends to be, say the class name, say the name of the variable. If you have arguments for the constructor, they will go in parenths after the variable's name.

So if scanner had something, I would be putting it right here, open parenth, yada, yada. So that's a little C++/Java difference. Oh, that's good. Question over here?

Student: When do we have to use the destructor?

Instructor (Julie Zelenski):So typically you will not ever make a call that explicitly calls the destructor. It happens for you automatically. So you're – [inaudible] you're gonna see it in the interface as part of the completeness of the class it, here's how I set up, here's how I tear down. When we start implementing classes we'll have a reason to think more seriously about what goes in the destructor. But now you will never explicitly call it.

Just know that it automatically gets called for you. The constructor kinda gets automatically called; the destructor gets automatically called, so just know that they're there. One of the things that's – I just want to encourage you not to get too bogged down in is that there's a lot of syntax to C++.

I'm trying to give you the important parts that are going to matter early on, and we'll see more and more as we go through. Don't let it get you too overwhelmed, the feeling of it's almost but not quite like Java and it's going to make me crazy. Realize that there's just a little bit of differences that you kinda got to absorb, and once you get your head around them actually you will find yourself very able to express yourself without getting too tripped up by it.

But it's just at the beginning I'm sure it feels like you've got this big list of here's a thousand things that are a little bit different that – and it will not be long before it will feel like your native language, so hang in there with us.

So I wanted to show you the vector before we get done today and then we'll have a lot more chance to talk about this on Friday. That the other six classes that come in [inaudible] class library are all container classes.

So containers are these things like they're buckets or shells or bags. They hold things for you. You stick things into the container and then later you can retrieve them. This turns out to be the most common need in all programs. If you look at all the things programs do, [inaudible] manipulating information, where are they putting that information, where are they storing it?

One of the sorts of obvious needs is something that is just kind of a linear collection. I need to put together the 100 student that are in this class in a list, well what do I do – what do I use to do that? There is a build in kind of raw array, or primitive array in C++. I'm not even gonna show it to you right now.

The truth is it's functional, it does kinda what it sets out to do, but it's very weak. It has constraints on how big it is and how it's access to it is. For example, you can make an

array that has 10 members and then you can axe the 12th member or the 1,500th member without any good error reporting from either the compiler or the runtime system.

That it's designed for kind of to be a professional's tool and it's very efficient, but it's not very safe. It doesn't have any convenience attached to it whatsoever. If you have a – you create a ten number array and later you decide you need to put 12 things into it, then your only recourse is to go create a new 12 number array and copy over those ten things and get rid of your old array and make a totally new one, that you can't take the one you have and just grow it in the standard language.

So we'll come back to see it because it turns out there's some reasons we're gonna need to know how it works. But for now if you say if I needed to make a list what I want to use is the vector. So we have a vector class in our class library that just solves this problem of you need to collect up this sequence of things, a bunch of scores on a test, a bunch of students who are in a class, a bunch of name that are being invited to a party.

And what it does for you is the things that array does but with safety and convenience built into it. So it does bounds checking. If you created a vector and you put ten things into it, then you can ask for the zero through 9th entries, but you cannot ask for the 22nd entry, it will raise an error and it will use that error function, you will get a big red error message, you will not bludgeon on unknowingly.

You can add things and insert them and then remove them. So I can go into the array and say I'd like to put something in slot zero, it will shuffle everything over and make that space. If I say delete the element that's at zero it will move everything down. So it just does all this kind of handling of keeping the integrity of the list and its ordering maintained on your behalf.

It also does all the management of how much storage space is needed. So if I put ten things into the vector and I put the 11th or the 12th or the – add 100 more, it knows how to make the space necessary for it. Behind the scenes it's figuring out where I can get that space and how to take care of it.

It always knows what count it has and what's going on there, but its doing this on our behalf in a way that that rawray just does not, that becomes very tedious and error prone if it's our responsibility to deal with it. So what the vector is kind of running, it's an instruction. And this is a key word for us in things that we're going to be talking about this quarter is that what you really wanted was a list.

I want a list of students and I want to be able to put it in sorted order or find this person or print them. The fact that where the memory came from and how it's keeping track of is really a tedious detail that I'd rather not have to deal with.

And that's exactly what the vector's gonna do for you, is make it so you store things and the storage is somebody else's problem. You use a list, you get an abstraction. How that – there's one little quirk, and this is not so startling to those of you who have worked on a

recent version of Java, is in order to make the vector generally useful, it cannot store just one type of thing.

That you can't make a vector that stores [inaudible] and service everyone's needs, that it has to be able to hold vectors of doubles or vectors of strings or vectors of student structures equally well. And so the way the vector class is actually supplied is using a feature in the C++ language called templates where the vector describes what it's storing using a placeholder.

It says, well this is a vector of something and when you put these things in they all have to be the same type of thing and when you get one out you'll get the thing you put in, but I will not commit to, and the interface saying it's always an integer, it's always a double. It's left open and then the client has to describe what they want when they're ready to use it.

So this is like the Java generics. When you're using an array list you said, well what kind of things am I sticking in my array list, and then that way the compiler can keep track of it for you and help you to use it correctly.

The interpart of this kinda looks as we've seen before. It's a class vector, it has a constructor and destructor and it has some operations that return things like the number of elements that you can find out whether it has zero elements, you can get the element at index, you can set the element at index, you can add, insert and remove things within there.

The one thing that's a little bit unusual about it is that every time it's talking about the type of something that's going into the vector or something that's coming out of the vector, it uses this elem type which traces its origin back to this template header up there, that is the clue to you that the vector doesn't commit to I'm storing ants, I'm storing doubles, I'm storing strings, it stores some generic elem type thing, which went the client is ready to create a vector, they will have to make that commitment and say this vector is gonna hold doubles, this vector is gonna hold ants, and from that point forward that vector knows that the getat on a vector of ants returns something of n type.

And then add on a vector of nts expects a perimeter of n type, which is distinct from a vector of strings or a vector of doubles. So I'll show you a little code and we'll have to just really talk about this more deeply on Friday. A little bit of this in text for how I make a vector of [inaudible] how I make a vector of strings, and then some of the things that you could try to mix up that the template will actually not let you get away with, mixing those types.

So we'll see this on Friday, so don't worry, there will be time to look at it and meanwhile good luck getting your compiler running.

[End of Audio]

Duration: 52 minutes

ProgrammingAbstractions-Lecture05

Instructor (Julie Zelenski):Good afternoon. Things that you want to know kind of administratively about what's going on; assignment 1 is dues next Wednesday, and because next Monday is MLK Jr. Day there'll be no lecture, so in between now and then is five days, and in case anything comes up that we want to let you know about, we're going to be using the web page.

In particular, there are a couple installation snags that we've already put up there, as well as a fix for the sample run being on a slightly different set of parameters than you think. And so you just want to keep an eye on that so in case anything comes up that might be important for you in completing that assignment, that you have access to the things that we're announcing to you there.

How many people have installed their compiler at this point? I've gotten a lot of email, but not all that I'm expecting. Okay, that's good. Those of you who have not, that's definitely something to do sooner rather than later. Although it seems like it should be the task that goes without trouble, software is never without its quirks, so depending on what OS you're using, what compiler, what configuration you're in, you may find that more challenging than you imagined, so starting on it sooner rather than later gives you a chance to email us and get help rather than kind of fighting against the compiler to the very last minute.

Today's topic is we're going to continue talking about the cs106 class [inaudible] today. Hopefully I'll get through vector grid/stacking queue. That will leave map and set for next Monday when we come back. And then after that we're going to move on to going back to the textbook and picking up at Chapter 4 doing recursion.

So the handout, handout 14, that massive thing I handed out last time is the material for this lecture and next one, and then we go back to the reader. I put a little note here about [inaudible] pointers. There is a raw built in array in a pointer type in C++, and they both have their utility and purpose. They're covered in Chapter 2 of the reader, but we're actually not gonna deal with that topic right now. We're gonna go ahead and build on vector and grid and these other things that in some ways do the things that arrays do and more.

And we will read this at the more raw built-in primitive types a bit later in the quarter when we have a good use for them. So at this point you can read those sections if you're curious, but it's not gonna be on our agenda for another couple weeks.

Today after lecture, I'll be walking over to the Truman Café in the basement and hanging out, so if you have some free time, you're not running off to go skiing, I would love to have you join me. Typically we kinda gather right here at the end of lecture and then we walk over together, so if you want to be sure you're in on it, just stay wherever I am and follow me. I'll – [inaudible] some questions that hold me up here for a little while before I make it over there.

Okay, anything administratively you'd like to ask about? How many people have completed assignment 1, and done the whole thing? All right, you get a gold star. All right, you guys want to be him, is what it is, because this guys is getting to go skiing guilt free. You guys if you're going skiing won't be guilt free, and you'll be working late to finish it off, and he is sleeping easy.

How many people have at least one or two of the problems done? Okay, that's a good number. We're still making progress. So I had just started to talk a little about this idea of a template, which is the C++ equivalent of the Java generic, and I want to refresh on the rules about how do you use a template.

The thing about templates is they're a very useful and practical component to have in the language, but they do have a little bit of issues with resolve to when you make mistakes with them, kinda having it reported and how you learn about them, and so they can be a little bit of a tripping point despite their vast utility.

Let me just remind you about what it means to use a template; is that you use include the interface file as usual. We're trying to use a vector to hold some sequence of things, so we include the vector.H. The name vector by itself without specialization doesn't tell the compiler everything it needs to know.

When you're trying to [inaudible] declare a vector, pass a vector as a parameter, return one, and any of those situations where you would have wanted to say it's a vector, you have to say what kind of vector, it's a vector holding character, it's a vector holding location T's, it's a vector holding doubles.

And that just applies everywhere you use the name vector, the name vector by itself doesn't mean anything. It always has to have this qualification or specialization on it. And then once that you've committed that to the compiler, the vector from that point really behaves in a type safe manner. A vector character is not the same thing as a vector holding doubles.

The compiler keeps those things separate. It doesn't let you co-mingle them. If you expect a vector of care, and you say that as your parameter, then you will have to pass one that has the same element type in it, that passing a vector double is not this same thing, and so trying to put a double into a vector of characters or retrieve an integer out of one that's holding student structures, is going to give you compiler errors, which is really a nice feature for maintaining type safety.

So the vector class I had just started talking about, and I'm gonna just kinda pick up and review the things we had started to talk about on Wednesday and go through it, which is what is the vector good for? The vector is good for any collection of things. You need to store a list is kind of the abstraction that it's trying to model.

I have a list of students in this class, I have a list of problems that are being assigned, I have a list of classes I'm taking this quarter, you know, whatever those things are, a list

of scores on an exam, and the vector manages the needs of all of those kinds of lists. You say what kind of thing you're storing in it. Every element has to be the same type, so that's what I mean by homogeneous, that all the elements are double or they're all students. You can't have doubles and students co-mingle.

It's linear in the effect that it kinda lays them out in a line. It indexes them from zero to the size minus 1. Each one has a place in the line and there are no gaps in it, so it actually is sequenced out. And it doesn't – a lot of things that make for a really convenient handling of the list as an abstraction, it knows its size at all time. You can ask it what the size is, it'll tell me how your elements have been stored in it.

Now if you ask for an element by index it bounds checks to make sure that you gave it a valid index for the range of size that it's currently holding. It handles all the storage for you. If you put ten elements and to put an eleventh, if it doesn't have space it goes and makes space for you. This all happens without you doing anything explicit, so as you add things, as you remove things, it handles sizing and changing whatever internal storage needs as needed to accommodate what you asked you.

It has convenient operations for inserting and removing; where you want to put something in a slot, it will move everything over to make space for it or to shuffle it down to close over that space. It also does what we call a deep copy; a deep copy is sort of a CS term for if I have a vector holding ten numbers, and I assign that to another vector, it really does make a new vector that has the same ten numbers.

That deep copy means it's more than some sort of shallow, like they're sharing something. They really are creating a clone of it, so taking that same – however big that vector is, whether it has a hundred or a thousand or two entries, it makes a whole copy, a parallel copy that has the same size and the same entries that was based on taking the original input and reproducing it in a new vector.

And that happens when you do assignment from one vector to another, it happens when you do a cast by value of a vector into a function that takes a vector by value, or when you return a vector from a function, so in all those cases it's doing kinda a full deep copy that is unlike those of you had a little experience working with a built in array, know that it doesn't have those behaviors, and that comes as a little bit of a surprise.

The vector behaves just like the primitives in the sense that there's no special knowledge you need to know about how it's – the assignment affects other copies of the same vector. So your typical usage is you create an empty vector, you add a new insert; remove to kind of jostle of the contents. You can access the elements once they're in there using member function setat and getat that allow you to change the value of the location, or get the value.

There's also an operator bracket. We'll see how you can actually just use the syntax of the vector name and then the bracket with the index to access a particular element, useful for all sorts of things. Question here?

Student: Yeah, can you make a multi-dimensional vector?

Instructor (Julie Zelenski):You can make a vector or vectors. The next class I'll talk about is a grid, which is kind of just a tooty thing that is already built, and you can also build vectors of vectors, and vectors of vectors of vectors to build to the higher and higher dimensions. And there's a little bit more syntax involved in doing that, but it [inaudible] the same basic functionality kind of applies in higher dimension.

So this is the basic interface of the vector, supplied as a template, so all the way through it, it refers to elem type as what you get at, what you set at, what you add and you insert all of the kind of values that are going in and out of that vector are left using this place holder rather than saying it's explicitly a double or a string or something, making no commitment about that, just leaving it open.

And then that template typed in elem type is the way that the whole class is introduced to say this is a pattern from which you can create a lot of different vector classes. Let me show you a little something on the next slide that helps to point this out. So here I have, in blue, put all the places where the elem type shows up.

I put the type name parameter introduced, and it says within the body of the class I'm using elem type as a place holder, and the four places it's showing up here, the getat the setat the add and the insert, that when I go to create a vector as a client, I'll say vector of double. Every place where there was elem type and on the vector name itself has been annotated or specialized to show that what's really going in out of this thing is double.

So this now created a class vector, angle bracket, double. The constructor and the destructor match the class name now, and the getat, the setat, the add, the insert, all have been locked down. We've made that commitment, we said what we're storing here really is vectors of doubles, and that means that the add number function for vector double takes a double parameter.

The getat returns a double value, and that way once the compiler has done this transformation that subsequent usage of that vector double variable will be consistent with this filling in of the placeholder, so it doesn't actually get confused about other types of vectors you may have been creating or working with. Question?

Student:[Inaudible]

Instructor (Julie Zelenski):No, these empty functions are really just a convenience off of size. You could always check size equals zero, and so it actually doesn't really add anything to the interface. It just turns out you do that so often in many situations, you want to know if this thing is totally empty, that as a convenience it offers two different ways to get at that same information.

So you're totally right, it's redundant. Sometimes you'll see that for example the standard template library has a bunch of the same things. The string has a length number function.

It also has a size number function. They do exactly the same thing. It's just because sometimes people can remember size, sometimes people remember length. There's also an empty – it's not called is empty, but it's just empty.

The [inaudible] is the length or size zero, and so all of those things are kind of written in terms of each other, but they allow different ways to get at the same information. Anything else? You got a question? Here we go.

Student: Is there a remove all method?

Instructor (Julie Zelenski):There is actually a clear method, and I should have put that up there. Actually, I think there's a few. In each of these cases I've excerpted a little bit for the most mainstream operations, but there is a clear operation that takes no arguments, returns no argument, that just takes the vector to an empty state.

So here's a little bit of code that shows some of the common use of vector and how you might do stuff with it that just gets you familiar with, okay, what does the syntax look like? So I'll look at this top function together, this is make random vector. You give it a parameter, which is the size, and it will fill a vector of integers with a sequence of random numbers up to that size.

You say I'd like a length ten vector filled with random numbers, it'll make a ten number vector stuffing in random numbers generated using the random library here. So you'll see the declaration here, so I included vector [inaudible], the compiler knew what I was using. I specialized when I declared that vector, and so the constructor for vector creates a new empty vector of that type, in this case vector of integer, and then numbers.add sticking in a bunch of numbers, and then I'm returning it.

So it's totally valid to actually return a vector as the return value coming out of a function. It'll take that, however many numbers I put in there, ten length vector, and make a full copy of it. And then when I'm down here and I'm saying nums equals make random vector, it actually copied that ten number vector into the variable beings stored in main.

So now I have a ten number thing with some random contents. The next thing I did with it was pass it to another routine that was going to print it, and there I am getting that vector, and this time I'm accessing it in here by reference.

This is just to show you that in typical because of the deep copying semantics it does mean if I didn't pass by reference, it means the full contents of the vector would get copied when I passed it to some function. There's no harm in that per se, other than the fact that it can get inefficient, especially as the vector gets larger, it has hundreds and thousands of entries, that making a full copy of those can have some overall efficiency effects on the program.

Passing by reference means it didn't really copy; it just kinda used the copy that was out here by reference, so reaching out and accessing it out of [inaudible]. So in here using the size to know how many elements are in the vector, and then this is what's called an overloaded operator, that the square brackets that you have seen in the past, user array access and the languages you know, can be applied to the vector, and it uses the same sort of syntax that we put an integer in that tells you what index and indices from zero to size minus one are the valid range for the vector, and accessed that integer and printed it out.

So anything you would have done on any kind of array, reading a bunch of contents from a file, printing these things out, rearranging them into sorted order, inserting something into sorted order, all those things are operations that work very cleanly when mapped onto what the vector provides.

One of the really nice things is unlike most array, like the built in array of C++, you don't have to know in advance how big the vector's gonna be, it just grows on demand. So if you were reading a bunch of numbers from a file, like you are in your assignment 1, you don't have to have figured out ahead of time how many numbers will be there so that I can allocate the storage and set it aside. You just keep calling v.dot add on your vector, and as needed it's just making space.

So if there's ten numbers in the file, there's a hundred, there's a million, it will just make the space on demand and you don't have to do anything special to get that access. Question?

Student:[Inaudible]

Instructor (Julie Zelenski):The square brackets.

Student: No, the [inaudible].

Instructor (Julie Zelenski):Oh, the angle brackets.

Student: Yeah, they turn the side of it and then – no?

Instructor (Julie Zelenski):No, no. The angle brackets are used in this case just for the vector specialization. The square brackets here are being used for – I'm accessing a particular member out of the array by index.

Student:[Inaudible]

Instructor (Julie Zelenski):So yeah, what happens in – so if you look at make random vector, it created an empty vector, so the typical usage [inaudible] is you create it empty and then you add things to grow it. So you actually never in here actually have to say how big it's going to be. It just – on demand as I call numbers.add, it got one bigger each time through that loop, and if I did that 100 times, that'll have a hundred entry vector.

So there isn't actually a mechanism where you say make it a hundred in advance. You will add a hundred things; it will have length a hundred, if you inserted a hundred things. The add me insert both cause the size of the vector to go up by one, remove caused to go down by one.

Student:[Inaudible] brackets to access a specific element and write to it and it's not yet at the – will it automatically fill in [inaudible]?

Instructor (Julie Zelenski):No, it will not, so the sub – the square brackets can only access things that are in the vector already. So you can overwrite it if it's there, but if you have a ten-member vector, and you go to say V sub 20, it will not sort of invent the ten members in between there and make that space. So the things that are valid to access to read from are the same ones that are valid to write to, so you can use the square bracket on either side of the assignment operator to read or to write, but it has to still be something that's already in there.

If you really want to put a hundred zeros into it, then you need to write a loop that puts a hundred zeros into it using add. Way in the back.

Student:[Inaudible]

Instructor (Julie Zelenski):Only for efficiency in this case. I'm not changing it, so if I was planning on going in here and multiplying everything by two or something, I would do that pass by reference to see those changes permanently affected. This actually isn't making any changes to the vector, it's just reading the contents of it, so it could be pass by value and have the same effect, but I wouldn't see any change in the program, it would just run a little slower if I did that.

We will typically though – you will see that kinda just by habit we will almost always pass our collections by reference, because of the concern for efficiency in the long run. So it – even though we don't plan on changing it, we'll use that to save ourselves some time. Anything about vector?

Student:[Inaudible]

Instructor (Julie Zelenski):The second to the last line before printing, the one right here where I'm going the – so this is declaring vector [inaudible], so making the new variable, and it's assigning it the return value from calling the make random vector function. So it's actually declaring and assigning it in one step where that assignment caused a function to be called that stuffed it full of random numbers and returned it.

Student: Ten in that case means what?

Instructor (Julie Zelenski):Ten in this case is just the size. It's the [inaudible] make me a random vector containing ten values, so that's – the ten in this case is just how many things to put in the array.

Student:[Inaudible]

Instructor (Julie Zelenski):Well it will make ten random numbers and stick them into the vector, so when you get back you'll have vector of size ten that has ten random entries in it. If I said a hundred I'd get a hundred random entries.

Student:[Inaudible] function, which library has it?

Instructor (Julie Zelenski):It is in random.H, so a lower case random.H, which is R cs1061. Okay. Now let me reinforce this idea that templates are type safe, and that if you misuse them you will get errors from the complier that help you to alert yourself to the mistakes that you've made. If I make a vector specialized to hold [inaudible] is really an integer vector, I make a vector to hold strings I call words, that I could add integers into the first one, I could add words to the second one, but I can't cross those lines.

If I try to take nums and add to it the string banana, it will not compile, so it has a very strong notion of the add operation on this kind of vector accepts this kind of thing. So if it's a vector of strings, the add accepts strings. If it's a vector of [inaudible] add accepts integers, and the crossing of that will cause compiler errors.

Similarly, when I'm trying to get something out of a vector, that return value is typed for what you put in it. If you have a vector of strings, then what you return is strings, not characters. Or trying to do, kind of take one vector; one vector is not equivalent to another if their base types are not the same. So a vector of doubles is not the same thing as a vector of integers. And so if I have a function that expects one or tries to use on, it really [inaudible] a vector of in, a vector of in is not the same thing as a vector of doubles, and the complier will not let you kind of mix those things up.

So it provides pretty good error messages in these cases. It's a, here's how you've gotten your types confused.

Student:[Inaudible] double bracket number and then –

Instructor (Julie Zelenski):Yeah, so if this said vector angle bracket [inaudible] then it would fine, then I would just be making a copy of the nums into a new variable S that had a complete same content that nums did. So that would be totally fine. I can definitely do assignment from one vector to another if they are of the same type, but vector in is not the same thing as vector double which is not the same thing as vector string, and so it's – basically it means that – what the template is, is a patter for which you can make a bunch of classes, and on demand it makes new classes, the vector double, the vector in, the vector string.

And each of those is distinct from the other ones that have been created.

Student: Can I change the types of nums in the expression and then –

Instructor (Julie Zelenski):You cannot type [inaudible], so it's not like I can type cast it down, they really are just different things and they're stored differently, ints are a different size and doubles, so there's a bunch more things that it will just not do automatically in that case. If I really wanted to try to take a bunch of integers and put them into a vector or doubles, I would end up having to kind of do a one by one, take each int, convert it to a double and stick it into a new vector to get that effect. Somebody in the back had something going on?

Student: Same question.

Instructor (Julie Zelenski):Same question, okay, so then we're good. Let me tell you a little bit about grid, which is just the extension of vector into two dimensions. Somebody asked about this a minute ago, which is like, well can we do this? We can still [inaudible] vectors of vectors as one way of getting into two dimension, but often what you have is really a rectangular region, where the number of rows and columns is fixed all the way across, in which case it might be convenient to have something like grid that you just specify how big you want, how many rows, how many columns, and then you get a full 2D matrix to hold those values.

So it is something that you set the dimensions of the constructors, you make a new grid on int that has ten rows and ten columns. There actually is a number function resize that lets you later change the number of rows and columns, but typically you tend to actually – once you set it, it tends to stay that way.

You have access to each element by row and column, so you say I want the to getat this row, this column, it will return you the value that's been stored there. And I put it over here; it says elements have default [inaudible]. So if you say I want a ten by ten grid of integers, then it does create a full ten by ten grid of integers. If you ask it to retrieve the value at any of those locations before you set them, they have the same contents that an integer declared of the stack would have, which is to say random.

So they're not zero, they're not negative one, they're not anything special. They're whatever – kind of if you just have int setting around, what is its default value? For the primitive types that just means it's random. For some of the more fancy types like string, it would mean they have the default value for string, which is the empty string.

So if I'm in a 2D grid of strings, then all the strings would be empty until I explicitly did something to set and assign their value. So there's a getat setat pair that looks very much like the vector form that takes, in this case, two arguments of the row and the column. There's also an operator parens that lets you say grid of parans, row column and separated by comma.

I'll show that syntax in just a second. It does do full deep copying, the same way the vector does, which is if you have a ten by ten grid which has a hundred members, when you pass or return it by value, or you assign it, it has a full copy of that hundred member grid.

Lots of things sort of fit into the world of the grids utility, any kind of game ward, you're doing battleship, you're doing a Sudoku, you're doing a crossword puzzle, designing a maze, managing the data behind an image or any kind of mathematical matrix, or sort of table tend to fit in the model for what a grid is good for.

This is the interface for grid, so a template like vector was. It has two different constructors; one that is a little bit of an oddity. This one creates a zero row, zero column grids, totally empty, which then you would later most likely be making a resize call to change the number of rows and columns. That might be useful in a situation where you need to create the grid before you kind of have information about how to size it.

You can alternatively specify with a constructor the number of rows and calls from the get go and have it set up, and then you can later ask it for the number of rows and calls and then you can getat and setat a particular element within that grid using those. There's also an operator – I'm not showing you the syntax in these for the operator open parens, just because it's a little bit goopy, but I'll show you in the client usage that shows you how it works from that perspective.

So this is something let's say like maybe I'm playing a tic tac toe game and I was gonna use the grid to hold the three by three board that has x's and o's in it, and I want to start it off by having all of them have the space character. So I'm going to using characters at each slot, I create a board of three three, so this is the way you invoke a construction in C++ that takes argument.

If you have no arguments, you don't have the parens or anything, you just have the name, but if it does take one or more arguments, you'll open the paren and list them out in order. In this case, the three and three being the number of rows and columns. And so at this point I will have a grid that has num rows three, num columns three, it has nine entries in it, and they're all garbage.

Whatever characters that were left over in that place in memory is what actually will be at each location. And then I did a nested four loop over the rows and columns, and here I am showing you the syntax for the access to the row column in kind of a shorthand form where I say board of and then parens bro , column = space. Equivalently that could have been the member function setat board. Setat row column space to accomplish the same result.

So this is still like vector sub I, it can be used to read or to write. It does bounced checking to make sure that row and column are greater than or equal to zero and less than the number of rows or number of columns respectively. So it raises an error if you ever get outside the bounds of the grid.

And then this return at the end just returns that full grid. In this case, the nine by entry, three by three back to someone else who's gonna store it and so something with it.

Student:[Inaudible]

Instructor (Julie Zelenski):There is actual one distinction between a vector and a vector of a grid that's kinda interesting, the grid is rectangular, it forces there to be exactly the same number of rows and column kind of for the whole thing. That a vector – if you created a vector of vector, you could individually size each row as needed. This could have ten, this could have three, and so like a vector vector might be interesting if you had something – well they call that kind of the ragged right behavior, where they don't all line up over here.

But if you really have something that's tabular, that there is a rectangular shape to it, the grid is going to be a little bit easier to get it up and running, but the vector vector's certainly would work, but if you were doing an array of class lists where some classes have ten and some classes have four hundred, but if you tried to do that with a grid you'd have to size the whole thing to have a much larger row dimension than was needed in most cases.

Student: So there's no square bracket operator?

Instructor (Julie Zelenski):There is not, and there is kinda an obscure reason for that, and if you are curious, you can come to the café afterwards, and I'll tell you why it is, but some of the syntax you may have seen in the past has two square brackets, you say sub row, sub column, and to have that work on a class in C++ doesn't – it's not as clean. So it turns out we use the parens, because it turns out as a cleaner was of accomplishing the thing we wanted, which was a short hand access into there.

So it doesn't actually use the square bracket operator. Question here?

Student:[Inaudible] when you created support three by three and you said it was forced to be a square, so why would you ever have to enter the second?

Instructor (Julie Zelenski):It's forced to be rectangular; it's not forced to be square. It could be three by ten, so I could have three rows by ten columns, but it does mean that every row has the same number of columns, but the row and number of rows and columns doesn't have to be equal, I'm sorry. [Inaudible] if I made squares, the wrong word really rectangular is it.

So then I'm gonna very quickly tell you these last two and they're actually even easier to get your head around, because they're actually just simplified versions of something you already know, which is to take the vector and actually limit what you can do with it to produce the stack in the queue class.

It seems like kind of a strange thing to do if you already had a class that did something, why would you want to dumb it down, but there actually are some good reasons that I'll hopefully convince you of that why we have a stack and a queue.

So what a stack is about is modeling the real world concept of a stack. If you have a stack of papers or a stack of plates you come and you put new things on the top of that and then

when it's time to take one off you take the top one. All right, so you don't dig through the bottom of the stack, you don't reach over to the bottom. So if you go up to get a plate in the cafeteria you just take the one that's on the top. When they put new clean ones they stick them on the top.

So this could be – you could take a vector and model exactly that behavior where all the edits to that – all the additions and remove operations have to happen on the top or one end of the vector, and that's basically what a stack does. Is it provides something that looks like a vector but that only allows you access to the top end of the stack.

It has the operation push which is to say put a new thing on the top of the stack. It has the operation pop, which is remove the top most element on the stack, and there's actually a peak operation that looks at what's on the top without removing it. There is no access to anything further down. If you want to see at the bottom or what's in the middle, the stack doesn't let you do it.

It gives a kind of a little window to look at this information that's restricted. That seems a little bit strange, why is it when you want – like sometimes you could do that with a vector by always adding to the end and always forcing yourself to remove from the end, but then just ignoring the fact that you could dig through the other parts of the vector.

And there are a few really good reasons to have the stack around, because there are certain things that really are stack based operations. A good example of is like if you are doing the web browser history when you can walk forward, but then you can back up. You don't need to be able to jump all the way back to the end, you're just going back in time.

Or if you undoing the actions in a word processors, you type some things and you undo it, that you always undo the last most operations before you undo things before that. And having it be a vector where you can kind of did through it means there's a chance you could make a mistake.

You could accidently pull from the wrong end or do something that you didn't intend. By having the stack it kinda forces you to use it the way you said you planned on, which is I'm only going to stack on the end, I'm going to remove from the end.

So it lets you model this particular kind of limited access vector in a way that prevents you from making mistakes, and also very clearly indicates to someone reading your code what you were doing. So for example, stacks are very well known to all computer scientists. It's one of the classic data structures. We think of, for example, the function calls as a stack.

You call main which calls binky which calls winky, well winky comes back and finishes, we get back to the binky call, we go back to main, then it always goes in this last in, first out, that the last thing we started is the first one to undo and go backwards to as we work our way back down to the bottom of the stack.

And so computer scientists have a very strong notion of what a stack is. You declare something as a stack and that immediately says to them, I see how you're using this. You plan on adding to the end and removing from that same end. So you can do things like reversal sequence very easily. Put it all on the stack, pop it all off, it came back in the opposite order you put it on. You put ABC on you'll get CBA off.

If I put 5 3 12 on, I'll get 12 3 5 off. So anytime I needed to do a reversing thing, a stack is a great place to just temporarily throw it and then pull it back out. Managing any sequence of [inaudible] action, the moves and again, the keystrokes in your edits you've made in the word processor, tracking the history when your web browsing of where you've been and where you want to back up to are all stack based activities.

So if you look at stack, it actually has an even simpler interface than vector for that matter. It has the corresponding size n is empty, so you'll see a lot of repetition throughout the interface where we could reuse names that you already know and have meaning for, we just reproduce them from class to class, so there's a size that tells you how many things are on the stack, and is empty, that tells you whether the size is zero, and then the push and pop operations that add something and remove it.

And they don't allow you to specify where it goes; it is assumed it's going on the top of the stack, and then that peak that lets you look at what's on the top without removing it. Yeah?

Student: So if you [inaudible] that had nothing in it, would you just get –

Instructor (Julie Zelenski):You get an error. So these guys are very bullet proof. One of the things we tried to do in designing our interface was give you a simple model you can follow, but also if you step out of the boundaries of what we know to be acceptable, we really stop hard and fast. So if you try to pop from an empty stacker, peak at an empty stack, it will print and error and halt your program.

It won't make it up, it won't guess, it won't –

Student:[Inaudible]

Instructor (Julie Zelenski):Well, so the stack knows its size and it [inaudible] is empty, so when you're unloading a stack you'll typically be in a loop like, well the stacks not empty, pop. So there's definitely ways you can check ahead of time to know whether there is something there or not, and it's all managed as part of the stack.

But if you blow it and you try to reach into the stack that's empty, it won't let you get away with it. And that's really what you want. That means that they run a little slower than the counterparts in the standard library, but they never let you make that kind of mistake without alerting you to it, where as the standard library will actually respond a little bit less graciously. It would be more likely to just make it up.

You tell it to pop and the contract is, yeah, I'll return you something if I feel like it and it may be what – it may be something that actually misleads you into thinking there was some valid contents on the stack and causes the error to kinda propagate further before you realize how far you've come from what its real genesis was.

So one of the nice things about reporting the error at the first glance is it gives you the best information about how to fix it. So here's something that just uses the stack to invert a string in this case, right, something a user typed in.

So I prompted for them to enter something for me, I get that line and then I create a stack of characters. So the stack in this case is being created empty and then I take each character one by one from the first to the last and push it onto the stack, and so if they answered Hello, then we would put H-E-L-L-O on the stack.

And then print it backwards, well the stack is not empty I pop and print it back out, so I'll get O-L-L-E-H back out of that and the loop will exit when it has emptied the stack completely. Stack [inaudible] just is a simpler form of that. The queue is just the cousin of the stack.

Same sort of idea is that there's certain usage patterns for a vector that form kind of their own class that's worth kinda giving a name to and building an abstraction around, the queue. The queue instead of being last in first out is first in first out. So the first thing you add into the queue is going to be the first one you remove.

So you add at the front – you add at the back and you remove from the front, it models a waiting line. So if you think of lets say the head of the queue and tail, or the front or the back of the line, that A was placed in the queue first, that operation's called n queue; n queue A, n queue B, n queue C, n queue D, and then when you d queue, which is the remove operation on a queue, it will pull the oldest thing in the queue, the one that was there first who's been waiting the longest. So removing the A and then next d queue will give you that B and so on.

So it does what you think of as your classic waiting line. You're at the bank waiting for a teller. The keystrokes that you're typing are being likely stored in something that's queue like, setting up the jobs for a printer so that there's fair access to it, where first come, first served, and there's a couple kind of search [inaudible] that actually tend to use queue as the way to kind of keep track of where you are.

So again, I could use vector for this, just making a deal with myself that I'll add at one end and I'll always remove the zero with element, but again, the effect is that if somebody sees me using a vector, that they'd have to look at the code more closely to see all my access to it, when I add, and when I remove to verify that I was using it in a queue like manner, that I always add it to the back an remove from the front.

If I say it's a queue, they know that there is no other access than the n queue d queue that operates using this FIFO, first in, first out control, so they don't have to look any closer at

my usage of it to know what I'm up to. The other thing that both stack and queue have which won't be apparent now, but will when we get a little further in the course, is that by defining the queue extraction to have kind of less features, to be what seems to be less powerful and have sort of a smaller set of things it has to support, also has some certain advantages from the implementation side.

That if I know somebody's always going to be sticking things on this end and that end, but not mucking around in the middle, then I can make certain implementation decisions that support the necessary operations very efficiently, but don't actually do these things well because they don't need to, in a way that vector can't make those trade offs.

Vector doesn't know for sure whether people will be mucking around with the middle or the ends or the front or the back, and so it has to kinda support everything equally well, that stack and queue has a really specific usage pattern that then we can use to guide our implementation decisions to make sure it runs efficiently for that usage pattern.

So the same constructor or destructor in size is empty, that kind of all our linear collections to, and then it's operations which look just like push and pop but with a slight change in verb here, n queue and d queue, that add to the back, remove from the front, and then peak is the corresponding look at what's in the front. Like what would be d queued, but without removing it from the queue, so just see who's at the head of the line.

And so a little piece of code I stole from the handout which sort of modeled a very, very simple sort of like if you're getting access to the layer and you're getting help, that you might ask the user, to kinda say well what do you want to do next, do you want to add somebody to the line or do you wanna service the next customer, and so we have this way of getting their answer.

And if they said, okay, it's time to service the next customer then we d queue and answer the question of the first person in the queue which will be the one who's been there the longest, waiting the longest. And if their answer was not next, we'll assume it was a name of somebody just stick onto the queue, so that actually adds things to the queue.

So as we would go around in this loop it would continue kind of stacking things up on the queue until the response was next and then it would start pulling them off and then we could go back to adding more and whatnot, and at any given point the queue will have the name of all the in-queued waiting questions that haven't yet been handled, and they will be pulled off oldest first, which is the fair way to have access in a waiting line.

So just the – very similar to the stack, but they – LIFO versus FIFO managing to come in and come out in slightly different ways. So once you have kind of these four guys, right, you have what are called the sequential containers kind of at your disposal. Most things actually just need one of those things. You need a stack of keystrokes, right, or actions or web pages you visited; you need a queue of jobs being queued up for the printer.

You need a vector of students who are in a class, you need a vector of scores on a particular exam, but there's nothing to stop you from kind of combining them in new ways to build even fancier things out of those building blocks, that each of them is useful in it's own right, and you can actually kind of mash them together to create even fancier things.

So like I can have a vector of queue of strings that modeled the checkout lines in a supermarket, where each checkout stand has it's own queue of waiting people, but that there's a whole vector of them from the five or ten checkout stands that I have, and so I can find out which one is the shortest line by iterating over that vector and then asking each queue what's your size the find out which is the shortest line there, and picking that for the place to line up with my cart.

If I were building a game, lets say, where there was a game board that was some grid, and that part of the features of this game which you could stack things on top of each location, then one way to model that would be a grid where each of the elements was a stack itself of strings.

So maybe I'm putting letters down on a particular square of the board, and I can later cover that letter with another letter and so on, and so if I want to know what is the particular letter showing at any particular grid location, I can dig out the row column location, pull that stack out and then peek at what's on the top. I won't be able to see things that are underneath, and that might be exactly need in this game, is that you can only see the things in top, the things underneath are irrelevant, until maybe you pop them and they are exposed.

So we just layer them on top of each other, we can make them as deep as we need to. It's not often they need to go past probably two levels, but you can build vectors of vectors of vectors and vectors of queues of stacks of grids and whatever you need to kind of get the job done.

There's one little C++ quirk that I'm gonna mention while I'm there, which is that the vector of queue of string actually has a closer – a pair of those closing angle brackets that are neighbors there, where I said I have a queue of string, and that I'm enclosing that to be the element stored in a vector, that if I put these two right next to each other, which would be a natural thing to do when I was typing it out, that causes the compiler to misunderstand what we want.

It will lead the grid stack string, and then when it sees the string, the next thing coming up will be these two greater than signs. It will read them together, so it effect tokenizing it as oh, these next two things go together, and they are the stream extraction operator, and then it just all haywire – goes haywire from there.

I think that you're about – you were in the middle of declaring a type and then all of a sudden you asked me to do a stream extraction. What happens, right? Sad, but true, and it will produce an error message, which depending on your compiler is more or less helpful.

The one is X-code is pretty nice, it actually says, closing template, there needs to be a space between it.

The one in visual studio is not quite as helpful, but it's just something you need to learn to look for, is that you do actually have to plant that extra space in there so that it will read the closer for one, and then the second closer without mingling them together.

There is an on deck proposal which shows you that C++ is a live language, it's evolving as we speak, but in the revision of C++ as being planned, they actually want to fix this so that actually it will be fine to use them without the space and the right thing will happen, and by changing it in the standard, it means the compiler writers will eventually kind of join to be spec compliant, will actually have to change their compilers to handle it properly, where as they now have little incentive to do so.

And then I just put a little note here that as you get to these more complicated things there might be some more appeal to using typedef, which is a C++ way of [inaudible] shorthand. You can actually do this for any type in C++. The typing, if you were typically something it would go up at the top of the program. I say typedef and then I give the long name and then I give the new short name, the nickname I'd like to give to it.

So I can say typedef into banana and then all through my program use banana as though it were an int. Okay, probably not that motivating in that situation, but when you have a type name that's somehow long and complicated and a little bit awkward to reproduce throughout your program, you can put that type name in place and use the shorthand name to kind of add clarity later.

So maybe I'm using this to be some information about my calendar, where I have the months divided into days or days divided into hours, so having kinda of a two layer vector here, that rather than having vector of vector of ints all over the place I can us calendar T to be a synonym for that once I've made that declaration.

Let me show you just a couple of things back in the compiler space before I let you guys run away to go skiing. One of the things that you're likely to be working on in this case is that you're learning a new API, API is called Application Programming Interface, it's the way you interact with the libraries, knowing what routine does what and what it's name is and what it's arguments are, and that's likely to be one of the things that's a little bit more of a sticking point early on here, is just kind of saying, oh I know this exists in a random library, but what's the name of the function, is it random numbers, is it random integers, is it random it?

And being familiar with the ways you kind find out this information. So let me just give you a little hint about this; one is that you can open the header files, so I just opened in this case, the grid.h header file, and I can look at it and see what it says. It says, oh, here's some information, it actually has some sample code here, and as I scroll down it'll tell me about what the class is, and then it has comma's on each of the constructor and member

function calls that tells me how it works and what errors it raises and what I need to know to be able to use this call correctly.

And so for any of our libraries, opening the header file is likely to be an illuminating experience. We try to write them for humans to read, so they actually do have some helpfulness. If you go to open a standard header file, they're not quite as useful. For example, let's go look at I stream and keep going.

You'll get in there and you'll see, okay, it's typedef template car basic I stream and then there's some goo and then there's a bunch of typedef's and then it gets down to here, there's a little bit of information that tells you about what the constructor might do and stuff.

You can read this, but it's not – it doesn't tend to be actually targeted at getting the novice up to speed about what's going on. There is some information here, but in those cases you may be better of using one of our references, going back to the reader, or looking at one of the websites that we give a point or two on the reference handout that just kind of tries to extract the information you need to know as a client rather than trying to go look in here, because once you get in here, oh what is gettake, and it's like what is all this stuff with these underbars and stuff, it's not the best place to learn the interface, I think, from their header files.

The other place that I will note that we have is up here on the website there is a documentation link, let me show you where I got to that just to remind you, is up here in the top, over on this side, and what this is, is actually this is our header files having been run through something that generates a webpage from them, so it has the same information available in the header files, but it's just organized in a clickable browsable way to get to things.

And so if you dink this down and you look at the class list, you can say, yeah, tell me about queue, I'd like to know more about it and then it'll give you the public member function. This projector's really very fuzzy, I wonder if someone can sharpen that, that tells you that here's the constructor, here's the n queue d queue peak, there's a clear operator that empties the whole thing and then there's some other operations that are involved with the deep copying that are actually explicitly named out.

And then if you go down, you can say, well tell me more about n queue, you can come down here, it'll tell you the documentation we had that's been extracted from the header files tells you about what the arguments are, what their turn value is, what things you need to know to use that properly.

And so you'll probably find yourself using one or both of these, like going and actually reading the header files or reading the kind of cleaned up pretty printed header files just to get familiar with what those interfaces are, what the names are, and how you call them so that when you're working you know, and have a web browser kind of up aside to help you navigate that stuff without too much guess work.

And so I just wanted to show you that before I let you go. Anybody questions about that? Hopefully you should feel a little bit like, hey, that starts to be useful. By Wednesday I'll show you map and set and then you'll realize there's a lot of really cool things you can do with all these objects around in your arsenal.

So have a good weekend, enjoy the holiday, come to Truman, I'll see you Wednesday.

[End of Audio]

Duration: 48 minutes

ProgrammingAbstractions-Lecture06

Instructor (Julie Zelenski):Hello. It's the mad rush. Welcome to Wednesday. As you can see by the growing mound of paper up here on the front of my desk, Assignment 1 is coming in today, the first of the sequence of tasks that I'm gonna set you to doing this quarter. And Assignment 2, the handout – I think Jason brought it – it is in the back for you to pick up on the way out.

The files actually aren't up for Assignment 2 yet, so you may wanna hold off jumping right on that until a little bit later today. There's been a little bit of a – my household's in a little bit of upheaval because my younger son got strep this weekend, which he thought the best thing to do with would be to give to me, so right now there's a little bit of strep. So you probably wanna stay away from me because I don't think you want it any more than I want it. And it also means that I'm gonna need to cancel my office hours today. I don't feel that good, and I need to get home to my kids, too. Okay, so let's just ask a little question about Assignment 1. I always like to do this when the assignments come in because it's good for me to kinda know what the experience was like for you, as well as kinda you to see in context how everything went for everybody. So let me just ask how many people think they managed to do the whole thing, start to finish, five problems in less than ten hours?

Less than ten? Okay, so that's actually like a good chunk of you. That's good. That's less than I would've expected. There's certainly some hurdles to kinda getting everything worked out in C++ that in general I would – the people who raised their hands, I'm assuming what you're telling me there is that the coding was very straightforward, like if you had to write that in Java, you probably could've done it in half the time, and probably most of the thing that got you tripped up was how do I say this in C++. What header file do I need? What is the compiler telling me? How many people think it took them between 10 and 15 hours? Okay, another equally large group. So that's kind of my target zone. So you guys are kinda right where I'm thinking it should be. Let me ask about 15 to 20? Okay, a little bit smaller group here. And then let's say more than 20. Anybody wanna admit to that? You can tell me later if you'd rather not. So that seems pretty good. I do expect that the assignments – at the beginning here, our first three assignments are a little bit like problem sets where we give you some small things to work on that give you some skill drilling.

The first big kind of comprehensive program is the one that goes out in the fourth week, but they do build up a little bit. We get some more mastery, and we kinda keep moving. And then Assignment 4 will be kind of a heftier thing, so kind of in terms of planning, I think, we believe that the range is probably 10 to 20 hours, but the early weeks tend to be a little bit more on the light side, and then they do kinda pick up steam. Any comments on the assignment that you wanna share that I should hear about?

Student:[Inaudible].

Instructor (Julie Zelenski):Somebody said they spent three hours, just getting this burnt and that, and whatnot. Was that the Mac or the PC that was so – the PC. We'll try to negotiate with Microsoft for an easier install pack. They've been trying to help us, but it turns out it's just not really set up to make it easy, and then the fact that there's a lot of different OSes that it needs to coordinate with, it's kind of like most PC installs. It's not actually trivial.

Student: One thing I was thinking is I'm sure a lot us made a text file to test the prompt in Problem 5. Maybe it would've been nice to have just a standard one rather than testing.

Instructor (Julie Zelenski):I think that's a good point, although I will say that in general I think learning to be able to make your own test data is actually really valuable, and the idea that knowing exactly what you put in it is almost sort of better than having me give you a test file which then you have to go look at, and figure out what's in it, and figure out what the results would be. If you stack the deck with 10, 20, 30, 40, 50, you'll know what to expect. And so in some ways, it actually kind of gives you I think better control.

I think sometimes what I found out when I give students test data, they don't tend to make up any other test data. If I give you no test data, then you'll have to make up some, and then that gets you thinking about how to create good test cases. But that said, it certainly would be trivial to do so. That's probably not where you spent your most time, though, I bet. How do you feel about C++ now after the assignment relative to kinda before you got into it? Are you feeling okay about your skills transferring? The compiler being a little bit of a change for you, the language.

Hopefully, you feel like – one thing I'd like to hear you saying is that you're confident about your programming skills coming with you, that you're not finding yourself having to relearn things that you knew how to do before. You're just learning how to express it a little bit in different syntax, and knowing how to divide things into functions, and test them, and write loops, and use variables should all seem very familiar, so hopefully that's not coming out too surprising. All right. So let me keep moving forward. We're gonna talk about the map and the set classes which are the remaining classes of the class library. Hopefully, I'll do most of that today. If not, what we won't finish today, we'll do on Wednesday.

And then we're gonna pick up and talk about recursion, and at that point we're gonna move back to the text, and we're gonna cover Chapters 4, 5, 6, and 7 just pretty much straight in order, so if you're looking to read ahead. I do think this is actually some of the best parts of the reader is the chapters – especially on the recursion chapters, 4, 5 and 6. So Eric Roberts, who wrote the original text that I edited and sort of worked on, I did the least editing in 4, 5, and 6, so they in some sense are most true to Eric's original work, and I think they are some of the strongest conceptual help. And recursion being kind of a tough thing to get your head around, I think you're gonna want second sources, so this is a good time to read ahead in the text, come to lecture having seen a little bit of it from that perspective, and ready to hear it from my perspective, and hopefully together we'll move toward mastery of that topic. So we've seen vector, grid, stack, and queue, which is

what we were doing on Friday, and these four are the group we call the sequential containers where they're storing and retrieving things on the basis of sequence. You place things LIFO into the stack, last in first out. You put things in the vector by index, retrieved by index. There's a sequence that's being maintained behind the scenes to store and retrieve the data.

All of these containers have the property that they don't really examine your elements. They just store them. So you can put things into a stack, but it never really looks at them. It never looks at the strings you put in or the students that you put in the vector. It just holds onto it, and then when asked to give you something back on N queue – I mean a D queue operation or a get at or something like that, it retrieves what you earlier placed there. These are not very fancy. They do things that are very useful, but the bang for the buck is mostly in terms of just modeling a particular behavior like the stack and the queue, and then allowing you to kind of use it easily without error. The associative containers, which is what map and set belong to, are the ones where you're really getting a lot of power out of them, where they do something that actually is hard, or inconvenient, or inefficient for you to do manually. They're not about sequencing. They're about relationships.

The set is a little bit like a vector in the sense that it's a collection of things without duplication, but it has fast operations for checking membership, adding things and removing things, checking to see if something's in the set in a way that the vector you'd end up kind of trudging through it to see do I already have this value in here. There's no easy way to do that with a vector as it stands. And so set gives you this kind of access for is this in the set, is it not, as well as a bunch of operations that allow you to do high level combinations like take this set and union it with another, or intersect it, or subtract this one from that, and get the mathematical properties of sets expressed in code. The map, even fancier in some sense, is a key value pairing that you want to be able to do some kind of look up by key. I wanna have a license plate number, and I wanna know what car that's assigned to, and what its make, and model, and owner is. The map is exactly the thing you want for that where you have some sort of key, or some sort of tag, or identifying ID that you wanna associate some larger thing with, or some other thing with for that matter, and be able to look that up, and make that pairing, and retrieving that pairing information quickly. Both of these are designed for very efficient access. It's gonna be possible to do these lookups, and additions, and combinations in much faster than you would be able to do if you were doing it manually. We're gonna peek under the hood in a couple weeks and see how that implementation works out and why, but for now what's pretty cool as a client is you just get to do neat things with it, stick stuff into it, check things, build things on top of it, taking advantage of all the power that's there through the kind of simple interface without having to learn what crazy internals behind it are. It's not something we have to worry about in the first pass.

So we'll talk about map. As I said, it's a collection of key value pairs sort of modeling a dictionary, a word and its definition. It could be that it's the student ID and it's their transcript. It could be that it's an index which tells you what a word – and what page as it appears in this text. So there's a key. There's a value. You have operations that once you

create a map where you can add a new pair, so you add a key and a value together and stick them in there. If you attempt to add a different value for an existing key, it will actually overwrite. So at any given time, there's exactly one value associated per key. You can access the elements that you stored earlier saying I've got a key, give me the associated value. I can check to see whether a key is contained in the map, just if I wanna know is there some entry already there. And a couple of the fancy features we'll get to in a second have some shorthand operators, and some access that allow you browsing. Some examples of things that maps are really good for – so maps turns out to be just ubiquitous across computer science problem solving. You probably saw the hashmap in your 106A class, and you got a little bit of work out of that toward the end of the quarter.

This guy, there's just a thousand uses for which a map is a great tool. A dictionary is an obvious one, a thesaurus where you have a word mapped to synonyms, the DMV which has license plate tags that tell you the three cars I've owned in my life and what their license plate numbers were – any sort of thing that looks like that fits in the map's goal. Let's take a look at what the interface actually looks like. This is a little bit simplified. I removed a couple of the advance features I'm gonna get to in a second. The class name is map. It's templatized on the value type. If you look at the add operation, it takes a string type for the key and a value type for the value, and that means that the map is templatized on only one of the half of the pair, that the pair is always a key which is a string mapped to some other thing, whether it's a student structure, or a vector of synonyms, or a string which represents a definition. That value type can vary based on what the client chooses to fill in the template parameter with, but the keys are always strings.

I'll kinda get to why that is a little bit later but just to note it while we're here. So things like remove, and contains key, and get value that all operate on a key always take a string, and then what they return or access is templatized by this value type. So there's one little thing to need to know a little about how the interface works. When you create a map, it is empty. It has no pairs. If you ask it for the size, it tells you the number of pairs in it. If you ever add something to a key that was already present – so if earlier I'd said Julie's phone number is this, and then I later went in and tried to add under Julie another phone number, it will replace, so there will not be entries for Julie. There's always exactly one. So add sometimes is incrementing the size, and other times the size will be unchanged but will have overwritten some previous pair. Remove, if there was a matching value will decrement the size. If actually there wasn't a matching entry, it makes no changes. Contains key can tell you whether something's in there. And then get value has a little bit of a quirk in its interface that you'll need to be aware of. If you ask it to retrieve a key that doesn't exist, it's got a little bit of a problem there. If you say give me the phone number for Julie, it's supposed to return to you the previously associated value, something of value type. Well, if you ask it to get a value for something and there was no pair that matched that, it has a little bit of a conundrum about what to return. For example, if this table were storing numbers – maybe you were storing the latitude and longitude of a city. Well, when if you asked it to return the value for a city it doesn't know about, what is it supposed to return? What is the default – some sort of sentinel value that says no latitude, no longitude. It says, "Well, what is that?" If you were getting strings – if you had definitions, you might return the empty string. If you were returning

numbers, it might be that what you wanna return was zero or negative one. There is not for all types of values some clear identifiable sentinel that would be appropriate to return.

So what happens actually in get value is if you ask it to retrieve a value for something that doesn't exist, it throws an error message. There is no other sensible return value it knows of, and so it treats it actually as a drastic situation. That means if you are probing the table for something that you traditionally wanna be checking contains key if you're not positive that it's already in there. If you do know, it's fine to go ahead and get value through, but if you are querying it, it's a two-step process to verify it's there before you go get it. So I got a little bit of code. I'm gonna go actually just type this code rather than show it to you prewritten because I think it's easier to learn something from when I actually type it, so I'm gonna go ahead and just show you. So what I have here is a little bit of code I wrote before you got here which is just designed to take a particular input text file and break it apart using stream extraction into words, in this case tokens that are separated by white space using the default stream extraction. So it has a while true loop that keeps reading until it fails, and in this case it just prints them back out.

What I'm gonna do is I'm actually gonna gather those words, and I'm gonna put them into a map, so this'll be my plan. So let's build a map that's gonna map words to counts, so the number of times a word occurs in a document will be stored under that key. So the first time I see "the," I'll put a one in, and the next time I see a "the," I'll update that one into a two, and so on. And when I'm done, I should have a complete table of all the words that occurred in the document with the counts of the number of times they occurred. So let me go ahead and I'm gonna pass it up here by reference so I can fill it in with something, and then what I'm gonna do here – I'm gonna say if M contains key word – so this means that there was an existing entry. I'm gonna add to it under the word – let's take a look at what we did here.

If it was already there, then I'm gonna get the value that was previously stored underneath it, add one to it, and then stick it back into the table, so that add is gonna overwrite the previous value. If it was four, it's gonna retrieve the four, add one to it, and then overwrite it with a five. In the case where it didn't contain that key, so no occurrence has been seen yet, we go ahead and establish a new entry in the able using m.add of the word with the count one. Yeah?

Student: Could you try the remove number function if there's no value there to develop the [inaudible]?

Instructor (Julie Zelenski):It does not, actually. So remove can actually – partly it has to do with a little bit about can you do something reasonable in that case. Remove in this case can say there's nothing to remove. Get value just can't do anything reasonable. It's supposed to return you something. It's like there's nothing to return. I can't make it up, and that's why that's considered a more drastic situation. Question?

Student: You passed the map on the wrong line.

Instructor (Julie Zelenski):Oh, you're right. I totally did. Thank you very much. That was gonna give me quite an error when I got that. There we go. Okay, fixed that. Thank you very much. And so then maybe when I'm done I could say num unique words, and then m.size. So if we let this guy go, it's gonna complain all over the place because it says what's this map of which you speak. Then we'll tell it here's a header file you'd like to derive. Okay. So there were 512 unique words in this document. It's the text of some handout we had earlier that I just quickly put in a text file. So this shows sort of the basic usage pattern will typically be I'm sticking stuff in there, and intentionally updating and changing, and then I in this case was using a count at the end. I'm gonna note one funky little thing about the map while I'm here because map has a shorthand access that allows you to retrieve the value associated with a key using a syntax that looks a little bit bizarre, but it's kinda becoming common – is to instead of saying m.get value word, that you say m of square brackets of word. So that looks a little bit like an array access, and the first time I saw this I have to say I was just shocked, and I thought, "Oh my god. What an abuse of the notational system," but it has become so ubiquitous in languages now that I actually don't consider it so frightening anymore. It's basically saying reach into the table, and instead of using an index to identify which element you're doing, you're using a key and saying reach into the bucket that's tagged with this index and get the value back out. So this is effectively an m.get value of word but using this syntax.

There's something that's a little bit wacky bout this syntax though. It's not exactly equivalent to get value in that it doesn't mind if you try to access something that doesn't exist. And what it will do is its strategy is if you access a word m of some word where this is not an existing key, it will create a pair, it will tag it with the key you asked for, and then it will kind of leave the associated value as to whatever the default value for that type is. So in this case it's almost like you declared an int variable on the stack, what do you get? It turns out you'll get garbage. You'll get kind of an uninitialized number which is of very little value, but the side effect of then having set this pair was up that we could immediately go in, and write on top of it, and make the assignment into the one. And so whatever junk condits were there were only temporary, and then I immediately overwrote it with that. In effect, that also means I can do things like this where I do m.word plus equals one, and now I'm using it both for the add and the get value part. So the m bracket form kind of handles both the things you think of as being add – set adding something into there, overwriting or adding a new value, as well as just retrieving it when you wanna read it like a get value.

So you'll see both of these used, but they do have a little bit of a quirky behavior that using the get value without the existing key throws an error. Using it this way actually kinda sets up an empty error that the assumption is that you're using it in this context where you're trying to immediately create it and overwrite it. Question?

Student: Does the case not matter for map? If it's –

Instructor (Julie Zelenski):Case does matter. If I had the word capital T "the" and lowercase "the," those are different words, so it really does – basically what you think of a string equals comparisons at the base level. If I wanted it to be case insensitive, then I

would need to actually go out of my way to convert the keys to uniform case to guarantee that they would match other forms of the word. So let me go back – and that's a little bit of code there. So let me just review a little bit about where we're at. That shorthand operator that we have – I said I would explain why is it that keys have to be string type. That if we're trying really hard to build these general purpose containers, it seems like it would be extra convenient if we could say the key is an integer, and the mapping is – we could use our student ID numbers, and it's a student's record that's there. Why does it have to be string type?

Well, it turns out that it really does use the known structure of strings to store them efficiently. It's capitalizing on certain properties that strings and strings only have to decide how to store those things so that it can actually do this very fast lookup that does not generally apply to other types of things. I couldn't say the key is a student structure and you have to match student structures. It's like there's not easy efficient ways to match any kind of value type, but there are certain things you can do with strings and strings only that it's capitalizing on.

So in the case where you have something and you're trying to use a key that isn't already in a string type, the recommended way to get around that is you have to figure out a way to convert it to a string. So if you have an int, a student ID number that you wanna use, or a phone number, something like that, you just convert it into a string form. You take the integer; you make it into a string. If you have a first name and a last name, and you wanna use them both, then you concatenate them together, and use that string as the key – just ways of building a string out of what you have. It's a little bit of a hack, but it solves the problem without too much trouble in most situations. The other question that comes up is what if I have more than one value for a key? The thesaurus example I gave earlier, the key is the word. The thing I want to map it to is the synonyms, that list of words. If I did add a word with each synonym, each subsequent synonym would just replace any previous pairing I'd put in.

If what I really want is a one to many relationship, one word many options, then what I can do is just use vector as the value, so build a map containing vectors containing strings. So I'll get a nested template out of that, and then when I wanna add a new synonym to the map, it's a matter of pulling out the existing map that's there and adding it to the end of that vector, so it actually is kind of a two step add to get it into the vector and get that vector into the table. So the last thing that's missing is one that I'll go back to my code and we can do together. Once I've put the information in the table, I hope you believe me that I can get it back fast. I can check to see if there's an existing entry and do the lookup by key, put new entries in, replace it, overwrite – all of that in the interface so far seems just fine. But let's say I just wanna see that whole table. I wanna print my class list. I wanna see the thesaurus or the dictionary just spewed out, or just actually walk through it and look for new words. As it stands, the interface doesn't give you that access. They're not indexed. They're not in there under Slot 0, Slot 1, or Slot 2. It's not like a stack or a queue where you can just dequeue them back out to see all the things you put in there. They kinda go in and so far it's the roach motel. You can put them in, and if

you know who you're looking for, you can get them back, but there's no way to kind of scan the contents.

What we're gonna see is that map provides access to the elements using a tool called an iterator. This is the same tool that Java uses, so if you've seen that, you're already ahead of the game. Where you would like to see the entries in it, and rather than giving you a whole vector or some other kind of random access version of it, it provides you with a little intermediary. In this case, this class is called iterator. You create an iterator on the class. Here, let me just go write the code. It's better to see that. So if I got here and I said I wanna see what they are, what I do is I create an iterator. The map's iterator name is a little bit wacky. There are iterators on both the map and the set, and to distinguish them because they're similar but not exactly the same thing, the iterator type is actually declared within the map class itself. So it's actually map of the type in angle brackets colon colon iterator is the name of the iterator for an iterator that walks over a map containing integers. We get one of those from the map by asking for it with the iterator member function. So that sets you up an iterator that's positioned to kind of work its way through it. The map iterator makes no guarantee about what order it will visit them, but it will visit them all. It uses a syntax that looks like this.

And I say iter.hasNext – are there more things to retrieve? So the iterator's kind of keeping track of where we are as we're working our way through the map. If there are more keys we haven't yet visited, haven't yet retrieved from the iterator, it will return true in which case the call to iter.next will retrieve the next key to be visited, and then advance the iter, so it will have moved past that and kind of moved that to the already visited side, and now it will continue to work on the things that are unvisited. As I keep doing that, iter.next is both retrieving it and advancing it, so you typically wanna call that once inside the loop to get the value, and then check that hasNext again to see whether there's more and keep going.

It gives you just the key. It doesn't give you the pair. The pair is just an easy call away, though. So I can say key equals and then M of key. And so using the shorthand syntax or the M – the get value, either way once I have a key, it's very easy to look up the value. In fact, it's so efficient, it means there's actually really no harm in just getting the key and going back in to get the value separately. It's still works out just fine. So when I do this, this should actually iterate through, show me every entry in the map, and the count for it. So one thing I will note is the sequence by which it's traveling through seems to be effectively random. You see my reasonable experience few following somewhere using the numbers 1, 4, 2, so there is no pattern. There is no rhyme or reason.

It turns out it is actually internally you're doing something kinda sensible, but it does not guarantee as for the iterator anything predictable to you about which sequence it will visit them in. It says I will get them all before it's all done and said, but don't count on seeing them in alphabetical order, reverse alphabetical order, in order of increasing value or anything clever like that. It will get to them all. That's all you can be sure of. That is the [inaudible] that's up here. Question?

Student: On the previous page, on the previous slide, you mentioned something about [inaudible] finding the [inaudible], you would have to use a vector for that?

Instructor (Julie Zelenski):Yeah. So if I had a one to many in that case, so maybe it was synonym to vector of strings, then I could use the iterator to go through and say how big is this vector compared to the other vectors I've seen so far. So I keep track of let's say the [inaudible] so far. So you'd run an iterator that – so let's just say I wanna find the most frequent word. That's about the same operation. Let me come back over here.

If I just wanna see the most frequent, I could do this. I could say string max, and I'll say int max count equals zero. So the max is empty here. When I get this thing, I say if M of key is greater than my max count, then I want my max to be the key, and my max count to be M of the key. So I've started right with no assumptions about what the max is, and any time I see something that's greater than the max I have so far, I update that.

So if this were a vector, I would be checking to see that the size of a value is bigger than the size of the previous vector. When I'm done, I can say max equals max count. It turns out "the." The most common word in there? What a surprise, right? It shows up 42 times.

So the same sort of [inaudible] would be used if there was some other thing I wanna do. So what the iterator is just giving us is a general-purpose way of walking through all the values. And then what you wanna do with them when you get there is your business. Do you wanna print them? Do you wanna put them in a file? Do you wanna compare them to find the top two, or the smallest, or the biggest, or whatever is up to you. So it's just giving you access to that data that you stuck in there and get it back out.

Student: So what does iter.next do?

Instructor (Julie Zelenski):Iter.next is – think of it as an abstraction. The idea is that it's almost like it's got a little pointer. It's kind of like if I were in charge of walking through this class and returning each student, it is a pointer that given a certain student will return you the student I'm currently pointing at, and then move to another random student. And at any given point when I'm done them all, that hasNext returns false.

And so next does two things, which is return to you the next key that you haven't yet seen, but advances kind of the iterator as a side effect so you can now test for are there any more, and if so, the next call will return a different one. So every time you call, it returns a new unvisited key in the map until they're all gone.

Student:[Inaudible] or the next one?

Instructor (Julie Zelenski):Well, it returns the next one in the iteration. You can imagine that it's almost like a caret that's between characters at any given point. It will return the next character, but if you call it again, it will return a new one. So it never returns the same value unless there was the same thing being iterated over. So it advances and returns in one step. That is key. There are some iterators that don't have that design.

They tend to return the current one, and then you side effect, the STL iterators for example. In this one is kind of more of a Java style iterator. It does both. So if you needed to use the value a couple times now, you'd probably store it in a local variable, and then use it throughout that thing, and then only call iter.next once in each loop. There is an alternate way of doing something to every pair in a map. I'm actually just not gonna show it to you because I think actually it's something that's overkill. So there is something you can read about in Handout 14 and look at which is called the mapping function. I'm gonna actually just skip it because I think it confuses the issue more than it helps. When you know how to do iteration, it's a great way to get access to all things in a map, and since the other function is actually just a more complicated way to get at the same functionality, I won't bore you with it.

Then let me tell you about set. Once you have set, you've got it all. So map I think is the heavy lifter of the class library. The set is kind of a close second best in terms of just everyday utility value. What set is is a little bit like a vector in some ways, but it has the added features that there is no sequencing implied or maintained by it. If you have the set that you've added 3 and 5 to, if you add 5 and 3 in the other order, you end up with the same set when you're done. Those two sets are equal. They have the same contents, so it doesn't consider the order of insertion as important at all. It's just about membership. Does it contain these values? There are never any duplicate elements. If you have a set containing 3 and 5, and you attempt to add 3 again, you don't get the set 3, 3, 5. You just get 3, 5. So kind of like the mathematical property that it derives its name from, there I just existence. Is the number in or not? And adding it again does not change its status if it was already there.

So your typical usage is you make an empty set. You can use add, remove, and contains to operate on individual elements. Add this one. Remove that one. Contains – remove has the same behavior that it did on the map which is it kind of silently fails if you ask it to remove something that wasn't there. If you ask add to add something that was already there, it silently doesn't change anything, so both of those operations just kind of quietly deal with the cases that might be a little bit unusual. And then contains will give you information about does this number exist in the set. All of those are very efficient, can be done many, many times a microsecond without taking any processor time at all, so that means it's very handy for just throwing things in a set and then later wanting to know if it's there, as opposed to walking your way through a vector piecemeal to see if something was there.

So for example, a vector doesn't have anything like this. If you wanna know if Julie is in your list of students, and you have them in a vector, the only way to know is to walk through it Sub 1, Sub 2, Sub 3 until you find it or you run out of entries to check. The contains operation does a blinding fast sort of check and can almost immediately tell you is Julie in there or not, whether your entries are a thousand, a million, a billion, pretty much immediate access to dig it out. It also offers these high level operations, and these actually are where sets are particularly valuable is this idea of unioning, intersection, subtracting, checking to see whether two sets are equal, so whether they have all the same numbers, or whether one has a subset of another – that allow you to model – a lot of

times the thing you want to take your code to do is something that in the real world needs a subset or an intersect kind of operation.

You'd like to know if the courses you have taken meet the requirements you need to graduate. Well, if the requirements to graduate are a subset of what you take, whether they're a proper subset or not, if they're all in there, then you can graduate. If you wanna know if you and I are taking any classes together, or have any requirements that we both need to do, we can take the requirements, see what we have, subtract what's left, intersect that to find out what we have left to see if there's any classes we could take together and both satisfy requirements. Any kind of compound Boolean queries like when you're trying to do searches on an index, you wanna see pages that have this word, and this word, or that word, and not that word are very easily modeled in terms of sets. You have the set that have A and the set that have B. If you intersect them, it will tell you about which things have both. If you wanna see the "or," you can use the union and find out which things have either.

Sometimes you use sets simply for this idea of coalescing duplicates. If I just wanted to see the set of words that were in my file, and I don't care about how many times a word occurs, I could just dump them all into a set, and then when I'm done, I will know what the 512 unique words are, and I will have not had to chase down did I already have a copy of that word because the set's add just automatically coalesces with any existing entry that matches. So let me show you its interface. This is probably the biggest of all our classes in terms of number of member functions there, and features of them. I'm not gonna talk about the constructor just yet because it has something scary in it that we're gonna come back to which is a little bit fancy in terms of how it works. There's a default argument over there. I'll note that, so that means that actually if you don't specify, in some situations the set doesn't need that argument and can kind of figure out for itself what to do. So we can create a set. We can ask for the size. Is empty – just checking to see if the size is zero. The things that operate on a single element here, adding, removing, containing – and then ones that operate on the set as a whole comparing it to another set.

That set there is one of the few places where you're allowed to use the name set without the qualification. That's because we're in the template, and in this situation, this set without qualification is assumed to be whatever set is currently being built. So if I have a set of int, the equals method for set of int expects another set of int as the argument. Same for is subset, and union, and intersect, so it actually kinda matches it out correctly all the way through, so you can only union sets of integers with other sets of integers. That should make sense to you.

These operations return the Boolean. These guys actually destructively modify the receiver. So you have a set that you're a messaging with a union with. You give it another set. It will join into the receiver set, so the receiver set will be enlarged by whatever new members are contributed by the other set. Intersect could potentially shrink this set down to just those elements that are in common with this other one. And then subtract takes all the elements that are in there that are present in here and removes them. So you can think of those as just modeling what the [inaudible] formula for those

operations are. It also uses an iterator. Like the map, once you stick them in there, they're not indexed. They're not by number. There's no way to say give me the nth element. There's no sequencing to them, so you use an iterator if you wanna walk through a set and see what's in there. So let me write you a little set code. Any questions about its basic interface?

Student: Why would somebody use a vector over a set?

Instructor (Julie Zelenski):Sometimes you actually do care about ordering, or you do want duplicates. You might have a bunch of names where maybe some people are gonna have the same names, or the ordering really is – that you wanna know who's in what room in a dorm, you might be using the index to say it's in Room 100. That's at what index?

So definitely when you care about that index, and where you also want that random access that I know a particular number, and I wanna see what's in that box. There really isn't that same feature in a set. There's no way to get the nth member. If you just wanted to pick a random member out of a set, the only way to do it would be to open up the iterator and take a random number of steps forward, whereas in a vector you can say just pick from zero to N minus one.

So there are definitely things that vector can do that set doesn't, and it just depends on the task which you have. Do you need duplicates? You're definitely stuck with vector. Do you care about random access? Do you use the index in some interesting way? Do you care about recording the sequence, like knowing who was first or last that you added? Set doesn't track those things in the same way. So let me write a little code that uses a set, and let's write this. I'm gonna write something that tests the random number generator. Every time I do this it alarms me, but I think it's good to know. I'm gonna keep a list of the numbers I've seen, and I'm going to write a for loop that then – I don't know. Let's just run it until we see a duplicate.

I'm going to generate a number between 1 and 100. If seen.contains that number, then I'm gonna break. Let me do a count. No, they'll be fine. Otherwise, I'm going to add it. And then I'm gonna print found seen.size or repeat. So what you might be hoping is that if random number generator was still truly random but still a little bit predictable – in this case, if I asked it for the numbers between 1 and 100, you'd like to think it would take about 100 iterations before it would get back to a number that's seen. But certainly given it's random, it's not actually picking and choosing without replacement. They're just kinda bopping around the space 1 to 100, and it may very well light back on a number it's already seen before it would get around to some of the other numbers. So when you run this, you're kinda curious to see what is it like. Let's find out. Every time I do this I always think, "Wow. The random number generator really not that random," but we'll see what today says.

I'm including random to get access to random integer. And I'm going to put my set here. And I'm gonna add one called randomize here at the top. That randomize initialized the

library so that we have a new random sequence for this particular run which will allow us to have different results from run to run. It said 24 numbers before repeat. Okay. Let's run that again. Seven before repeat. Let's try it again. 20 before repeat. So showing you a little bit about the – seven again. Six. Ten. Like is it ever going to get close to 100? No. So it just tells you that it bops around, but it actually does not – it is pseudorandom, which is one of the words that computer scientists use to say, "Yeah, don't count on it being really random." I would not wanna run my casino on the basis of numbers being generated by your average C++ random library. There you go. So in this case, just using it for containment so that each time through I can add that new number and then check the contains. Both those operations act operating very quickly. If I did this instead with a vector, I could accomplish the same thing, but it'd just be more manual. I'd stick it on the end, and if I wanted to see if it was there, contains is not an operation vector supports, so I'd have to walk through the vector from zero to the length minus one, and try to do the matching myself, which gets slower and slower as the vector gets longer. It's just a hassle to write that code, so having contains around saved us a little bit.

If I want to print my sets, why don't go ahead and just at the bottom show that the iterator gets used on the set, has the similar kind of formed name as the map, so the iterator – in this case capital I is a nested class declared within the set. I ask the set to give me its iterator while there are things left to pull out, then I will pull out it and print it. And so there's the sequence of let's say 20 or so numbers.

So this highlights something that's a little bit distinctive about the iterator as it applies to the set is that those numbers – 15, 16, 22, 24, 37 – are coming out in increasing order all the way down to the bottom. And if I run it again, I will get the same effect, but perhaps on a shorter list. That the sets iterator is not as unpredictable as the map iterator was – that the set iterator – that in fact part of how the set is able to supply its operations so efficiently is that internally it is using some notion of sorting. It is keeping track of things by ordering. In this case, the increasing order is the default strategy for how it lines things up – and that the iterator takes advantage of that. It is internally storing in sorted order. It might as well just use the iterator to walk them in sorted order, and that tends to be convenient for somebody who wants to process this set. So as a result, you can count on that, that when you're using a set, that the iterator will put out the values from kinda smallest to largest. So for example, for strings it would be the lexicographic ordering, that kind of slightly alphabetical thing based on ASCII codes that it would produce them in. In numbers, they'll go from smallest to largest as a convenience, and it happens to be kinda nice for just browsing it in alphabetical order. Often it's something that's handy to be able to do.

We head back over here. And so the code we just wrote, printing the set, doing the random test, I can apparently reproduce my own code on demand. And then here it just shows a little bit of some of the fancier features you can start going once you have sets in play. So if I were keeping track of for the friends that I know who they consider to be their friends, and who they consider to be their enemies – of course, those are very small lists I'm sure for you indeed, but if I were putting together a party, what I might wanna do is say let's invite everybody friends with and everybody I'm friends with, but then

let's make sure we don't invite anybody that one of us doesn't like, that's on our enemy list. And so by starting with a copy of the friends that one has, unioning that with the two's friends, right now I've got the enlarged set which includes – note that I did a set string result equals one friends. That actually does a complete deep copy, so the map and the set just like our other containers know how to copy themselves and produce new things. So I got a new set that was initialized with the contents of one's friends. I took that set and destructively modified it to add two's friends, so now that result has an enlarged circle that – and then I'm further destructively modifying it to remove anyone who was on my enemy list and your enemy list to get us down to kind of the happy set of folks that either of us likes and neither of us dislikes to do that. So things that you can imagine if you had to do this with vectors would start to get pretty ugly. Walking down your list and my list to see who's matching, to see who you have, I don't, to make sure that everybody got one and only one invitation. The set is automatically handling all the coalescing of duplicates for us.

And then allowing you this kind of high level expression of your actions is very powerful. If in the end what you're trying to model is this kind of rearrangement, then being able to union, subtract, intersect really helps the clarity of your code. Somebody can just read it and say, "Oh, I see what they're doing here," doing set operations to get to where we wanna be. Question?

Student:[Inaudible] the same enemies, so when you subtract one's enemies, and then you [inaudible] again with two it's not there.

Instructor (Julie Zelenski):It turns out subtract will say remove these things if they're present. So if you and I have some of the same enemies, after I've removed them from your list, I'll remove them from my list. It turns out they won't be there, but it doesn't complain. Like the remove operation, if you ask it to subtract some things, it doesn't get upset about it. It just skips over them basically. The set though is a little bit more quirky than the others in one kind of peculiar way. I'm gonna get you – foreshadow this. I'm actually gonna do more serious work on this on Friday – is that the other containers kinda store and retrieve things. They're putting them in buckets, or in slabs, or boxes, or whatever, and returning them back to you. But set is actually really has a much more intimate relationship with the data that it's storing that it really is examining the things that you give it to make sure for example that any existing copy is coalesced with this one, that when you ask it to go find something it has to do some kind of matching operation to say do I have something that looks like this.

It's doing this sorting internally, as I said. It's keeping them ordered. Well, it needs to know something about how to do that ordering, what it means for two elements to be the same, what it means for one element to precede another or follow another in some ordering. Okay. But set is written as a template. Set takes any kind of elem type thing. How is it you compare two things generically? That actually is not an easy problem. I'll show you a little bit off this code, and then we'll talk about this on Friday more – is that one way you could do it is you could assume that if I take the two elements, I could just use equals equals and less than. If I just plug them into the built-in operators and say,

“Tell me. Are they equal? What does equal equal say? Is one of them less than the other? Tell me that.” Using the built-in operators will work for some types of things you would store. It’ll work for ints. It’ll work doubles. It’ll work for characters.

So all the primitive types have relational operator behavior. Even things like string, which is a sort of fancier type also has behavior for equals equals and less than. But you start throwing things like student structures into a set, and I say take two students and say if they’re equal equal, we’re gonna run into some trouble. So I’ll show you that error message, and we’ll talk more about what we can do about it when I see you again on Friday. If you have something you need to talk to me about, now would be a good time before I run away, so if you wanted to see me today at office hours.

[End of Audio]

Duration: 49 minutes

ProgrammingAbstractions-Lecture07

Instructor (Julie Zelenski):That it felt it was too long, I asked too much of you, and it decided the best thing I could do for you was just to throw a page out of the list. So you will notice the handout that we gave out on Wednesday, it goes straight from Page 4 to Page 7 I think or something like that. It doesn't really quite make sense when you read it because it's in the middle of talking about one problem, and then suddenly it's talking about something else. We photocopied the missing page and it's in the lobby on the way out. You can also just get it from the web. The PDF that's up there's complete. You can print the middle page out by yourself or just grab the whole PDF to take a look at, but you do wanna have that handy when you start working on the assignment because it will be quite mysterious if you try to read it as is with that missing page.

What are we going on to talk about? I'm gonna talk about functions as data a little bit and client callbacks, and hopefully a little bit of introduction to recursion. We won't get very far in recursion today, just enough to get some terminology and see a couple simple examples, but recursion is gonna be sort of the entirety of next week, so we'll have lots of time to soak this in and become masters of it. And that reading that matches with that is basically Chapter 4, 5, and 6 in order, which is all the recursion chapters in the text.

As you can probably tell, I'm still working on getting rid of my strep. Technically, I'm not contagious. Apparently, once you've been on antibiotics for 24 hours, you're no longer spreading the germs, but I'd just as soon not want you to hang out with me today more than you need to. I'm not sure it's good for you or for me for that matter, but I will be actually in my office for a while after class, so if you were hoping to come see me on Wednesday, and you couldn't because I canceled my hours or you just have something on your mind, feel free to walk back with me and talk a little bit today. Anything administratively? Way in the back?

Student:[Inaudible] that handout [inaudible]?

Instructor (Julie Zelenski):Oh, it will. He's asking about solutions to Section 2, and we're – I bet you Jason could have it up before the end of the – he could put it up right now. How about that? We try to hold them back until actually after people have had section, just because it helps to kind of increase the mystery, keep the suspense, but then sometimes we forget to kind of patch together. So if we ever do that, send us an e-mail. Jason's actually your section go-to guy actually, so if you actually have questions about the section materials, he is the most on top of that, more so than I am. Anything else? Okay, so thank you for coming out in the rain. I appreciate you trudging over here and risking your health and life. So I'm gonna do a little diversion here to explain this idea of functions as data before I come back to fixing the problem we had kind of hit upon at the very end of the last lecture about set. So what I'm just gonna show you is a little bit about the design of something in C++ that is gonna helpful in solving the problem we'd run into. What I'm gonna show you here is two pieces of code that plot a single value function across the number line. So in this case, the function is that it's the top one is applying the sine function, the sine wave as it varies across, and then the square root

function which goes up and has a sloping curve to it, and that both of these have exactly the same behaviors.

They're designed to kind of go into the graphics window, and they're just using a kind of a very simple kind of hand moving strategy. It starts on a particular location based on what the value of sine is here, and then over a particular interval at 0.1 of one tenth of an inch, it plots the next point and connects a line between them. So it does a simple kind of line approximation of what that function looks like over the interval you asked it to plot. The same code is being used here for plotting the square root. And the thing to note and I tried to highlight by putting it in blue here was that every other part of the code is exactly the same except for those two calls where I need to know what's the value of square root starting at this particular X. So starting at Value 1, what is sine of one? What is square root of one? As I work my way across the interval, what's a square root of 1.1, 1.2, 1.3 and sine of those same values? And so everything about the code is functionally identical. It's kind of frustrating to look at it, though, and realize if I wanted to plot a bunch of other things – I also wanna be able to plot the cosine function, or some other function of my own creation – across this interval that I keep having to copy and paste this code and duplicate it.

And so one of the things that hopefully your 106A and prior experiences really heightened your attention to is if I have the same piece of code in multiple places, I ought to be able to unify it. I ought to be able to make there be a plot function that can handle both sine and square root without actually having to distinguish it by copy and pasting, so I wanna unify these two. And so there is – the mechanism that we're gonna use kinda follows naturally if you don't let yourself get too tripped up by what it means. Just imagine that the parameter for example going into the function right now are the start and the stop, the interval from X is one to five. What we'd like to do is further parameterize the function. We'd like to add a third argument to it, which is to say and when you're ready to know what function you're plotting, here's the one to use. I'd like you to plot sine over the interval start to stop. I'd like you to plot square root over that. So we added the third argument, which was the function we wanted to invoke there. Then we would be able to unify this down to where we had one generic plot function.

The good news is that this does actually have support features in the C++ language that let us do this. The syntax for it is a little bit unusual, and if you think about it too much, it can actually make your head hurt a little bit, think about what it's doing. But you can actually use a function, a function's name and the code that is associated with that name, as a piece of data that not just – you think of the code as we're calling this function, and we're moving these things around, and executing things. The things that we tend to be executing and operating on you think of as being integers, and strings, and characters, and files. But you can also extend your notion of what's data to include the code you wrote as part of the possibilities. So in this case, I've added that third argument that I wanted to plot, and this syntax here that's a little bit unusual to you, and I'll kind of identify it, is that the name of the parameter is actually FN. Its name is enclosed in parentheses there. And then to the right would be a list of the arguments or the prototype information about this function, and on the left is its return value. So what this says is you

have two doubles, the start and stop, and the third thing in there isn't a double at all. It is a function of one double that returns a double, so a single value function that operates on doubles here.

That is the syntax in C++ for specifying what you want coming in here is not a double, not a ray of doubles, anything funky like that. It is a function of a double that returns a double. And then in the body of this code, when I say FN here where I would've said sine, or square root, or identified a particular function, it's using the parameter that was passed in by the client, so it says call the client's function passing these values to plot over the range using their function in. So the idea is that valid calls to plot now become things like plot – and then give it an interval, zero to two, and you give the name of a function: the sine function, which comes out of the math library, the square root function, also in the math library. It could be that the function is something you wrote yourself, the my function. In order for this to be valid though, you can't just put any old function name there. It is actually being quite specific about it that plot was to find [inaudible]. It took a double, returned a double. That's the kind of function that you can give it, so any function that has that prototype, so it matches that format is an acceptable one to pass in.

If you try to pass something that actually just does some other kind of function, it doesn't have the same prototype, so the get line that takes no arguments and returns a string just doesn't match on any front. And if I try to say here, plot the get line function of the integral two to five, it will quite rightfully complain to me that that just doesn't make sense. That's because it doesn't match. So a little bit of syntax here, but actually kind of a very powerful thing, and it allows us to write to an addition to kind of parameterizing on these things you think of as traditional data, integers, and strings, and whatnot. It's also say as part of your operations, you may need to make a call out to some other function. Let's leave it open what that function is and allow the client to specify what function to call at that time. So in this case for a plot, what function that you're trying to plot, let the client tell you, and then based on what they ask you to plot you can plot different things. All right. Way in the back.

Student: Is there a similar setup for multivariable functions [inaudible]?

Instructor (Julie Zelenski):Certainly. So all I would need to do if this was a function that took a couple arguments, I would say double comma double comma int. If it returned void, [inaudible] returned. Sometimes it looks a little bit like the prototype kinda taken out of context and stuffed in there, and then those parens around the function are a very important part of that which is telling you yeah, this is a function of these with this prototype information. Behind you. No? You're good now. Somebody else? Over here?

Student: Is that FN a fixed name [inaudible]?

Instructor (Julie Zelenski):No. It's just like any parameter name. You get to pick it. So I could've called it plot function, my function, your function, whatever I wanted. Here in the front.

Student:[Inaudible] Java?

Instructor (Julie Zelenski):So Java doesn't really have a similar mechanism that looks like this. C does, so C++ inherits it from C. There are other ways you try to accomplish this in Java. It tries to support the same functionality in the end, but it uses a pretty different approach than a functions as data approach. [Inaudible]?

Student: Can you pass like a method, like an operator?

Instructor (Julie Zelenski):So typically not. This syntax that's being used here is for a free function, a function that's kind of out in the global namespace, that level. There is a different syntax for passing a method, which is a little bit more messy, and we won't tend to need it, so I won't go there with you, but it does exist. It just as it stands does not want a method. It wants a function. So a method meaning member function of a class. Okay, so let me – that was kind of just to set the groundwork for the problem we were trying to solve in set, which was set is holding this collection of elements that the client has stuffed in there. It's a generic templative class, so it doesn't have any preconceived notion about what's being stored. Are the strings? Are they student structures? Are they integers? And that in order to perform its operations efficiently, it actually is using this notion of ordering, keeping them in an order so that it can iterate an order. It can quickly find on the basis of using order to quickly decide where something has to be and if it's present in the collection.

So how does it know how to compare something that's of unknown type? Well, what it does is it has a default strategy. It makes an assumption that if I used equals equals and less than that would tell me kinda where to go. And so it's got this idea that it wants to know. We'll give it two things. Are they the same thing? In which case, it uses kinda the zero to show the ordering between them. If one precedes the other, it wants to have some negative number that says well this one precedes it, or some positive number if it follows it. So it applies this operation to the [inaudible] that are there, and for strings, and ints, and doubles, and characters, this works perfectly well. And so that's how without us going out of our way, we can have sets of things that respond to the built in relational operators without any special effort as a client. But what we can get into trouble with right is when equals equals less than don't make sense for type.

So let me just go actually type some code, and I'll show you. I have it on the slide, but I'm gonna – wow, this chair suddenly got really short. [Inaudible] fix that. Okay. We'll go over here because I think it's better just to see it really happening, so I'm gonna ignore this piece of code because it's not what I wanted. But if I make some student T structure, and it's got the first and last name, and it's got the ID number, and maybe that's all I want for now – that if down in my main – can't find my main. There it is. I'm gonna need that piece of code later, so I'm gonna leave it there. If I make a set, and I say I'd like a set of students – my class – and I do this. And so I feel like I haven't gotten [inaudible]. I made this structure. I say I'd like to make a set of students, that each student is in the class exactly once and I don't need any duplicates, and I go through the process of trying to compile this, it's gonna give me some complaints. And its complaint, which is a little

bit hard to see up here, is there's no match for operator equals equals in one, equals equals two, and operator less than in one equals two.

So it's kind of showing me another piece of code that's kind of a hidden piece of code I haven't actually seen directly, which is this operator compare function. And that is the one that the set is using, as it has this idea of what's the way it should compare two things. It says well I have some of this operator compare that works generically on any type of things using equals equals and less than. And if I click up here on the instantiate from here, it's gonna help me to understand what caused this problem. The problem was caused by trying to create a set who was holding student T. And so this gives you a little bit of an insight on how the template operations are handled by the compiler, that I have built this whole set class, and it depends on there being equals equals and less than working for the type.

The fact that it didn't work for student T wasn't a cause for alarm until I actually tried to instantiate it. So at the point where I said I'd like to make a set holding student T, the first point where the compiler actually goes through the process of generating a whole set class, the set angle brackets student T, filling in all the details, kinda working it all out, making the add, and contains, and whatever operations, and then in the process of those, those things are making calls that try to take student T objects and compare them using less than and equals equals. And that causes that code to fail. So the code that's really failing is kind of somewhere in the class library, but it's failing because the things that we're passing through and instantiating for don't work with that setup. So it's something we've gotta fix, we've gotta do something about. Let me go back here and say what am I gonna do. Well, so there's my error message. Same one, right? Saying yeah, you can't do that with a [inaudible].

Well, what we do is we use this notion of functions as data to work out a solution for this problem that if you think about kinda what's going on, the set actually knows everything about how to store things. It's a very fancy efficient structure that says given your things, keeps them in order, and it manages to update, and insert, and search that thing very efficiently. But it doesn't know given any two random things how to compare them other than this assumption it was making about less than and equals equals being a way to tell. If it wants to have sort of a more sophisticated handling of that, what it needs to do is cooperate with the client – that the implementer of the set can't do it all. So there's these two programmers that need to work in harmony. So what the set does is it allows for the client to specify by providing a function.

It says well when I need to compare two things, how about you give me the name of a function that when given two elements will return to me their ordering, this integer zero, negative, positive that tells me how to put them in place. And so the set kind of writes all of its operations in terms of well there's some function I can call, this function that will compare two things. If they don't specify one, I'll use this default one that maps them to the relationals, but if they do give me one, I'll just ask them to do the comparison. And so then as its doing its searching and inserting and whatnot, it's calling back. We call that calling back to the client. So the client writes a function. If I wanna put student Ts into a

set, then I need to say when you compare two student Ts, what do you look at to know if they're the same or how to order them. So maybe I'm gonna order them by ID number. Maybe I'm gonna use their first and last name. Whatever it means for two things to be equal and have some sense of order, I supply – I write the function, and then I pass it to the set constructor.

I say here's the function to use. The set will hold on to that function. So I say here's the compare student structure function. It holds onto that name, and when needed it calls back. It says I'm about to go look for a student structure. Is this the one? Well, I don't know if two student structures are the same. I'll ask the client. Here's two student structures. Are they the same? And then as needed, it'll keep looking or insert and add and do whatever I need to do. So let's go back over here. I'll write a little function. So the prototype for it is it takes two elem Ts and it returns an int. That int is expected to have a value zero if they are the same, and a value that is negative if the first argument precedes the second. So if A is less than B, returns some negative thing. You can return negative one, or negative 100, or negative one million, but you need to return some negative value. And then if A [cuts out] some ordering, so they're not equal [cuts out] later, it will return some positive value: 1, 10, 6 million.

So if I do [cuts out] say I use ID num as my comparison. Based on [cuts out] are the same, if they are I can return zero. And if ID num of A is less than the ID num of B, I can return negative one, and then in the other case, I'll return one. So it will compare them on the basis [cuts out] figuring that the name field at that point is nothing new. And then the way I use that is over here when I'm constructing it is there is [cuts out] to the constructor, and so that's how I would pass [cuts out] add parens to the [cuts out] as I'm declaring it, and then I pass the name. Do I call it compare student or compare students? I can't remember. Compare student. Okay. And this then – so now there's nothing going on in the code – causes it to compile, and that if you were to put let's say a see out statement in your comparison function just for fun, you would find out as you were doing adds, and compares, and removes, and whatnot on this set that you would see that your call kept being made. It kept calling back to you as the client saying I need to compare these things. I need to compare these things to decide where to put something, whether it had something, and whatnot.

And then based on your ordering, that would control for example how the iterator worked, that the smallest one according to your function would be the one first returned by your iterator, and it would move through larger or sort of later in the ordering ones until the end. So it's a very powerful mechanism that's at work here because it means that for anything you wanna put in a set, as long as you're willing to say how it is you compare it, then the set will take over and do the very efficient storing, and searching, and organizing of that. But you, the only piece you need to supply is this one little thing it can't figure out for itself, which is given your type of thing, how do you compare it.

For the built in types string, and int, and double, and car, it does have a default comparison function, that one that was called operator compare. Let me go open it for you [inaudible] the 106. So there is a compare function dot H, and this is actually what

the default version of it looks. It's actually written as a template itself that given two things it just turns around and asks the built in operators to help us out with that. And that is the name that's being used if I open the set and you look at its constructor call. I had said that I would come back and tell you about what this was that the argument going into the set constructor is one parameter whose name is CMP function that takes two elem type things – so here's the one in the example of the two argument prototype – returns an int, and then it uses a default assignment for that of operator compare, the one we just looked at, so that if you don't specify it, it goes through and generates the standard comparison function for the type, which for built-ins will work fine, but for user defined things is gonna create compiler errors.

So you can also choose if you don't like the default ordering works. So for example, if you wanted to build a set of strings that was case insensitive – so the default string handling would be to use equals equals and less than, which actually does care about case. It doesn't think that capital [inaudible] is the same as lower case. If you wanted it to consider them the same, you could supply your own. A compare case insensitively function took two strings, converted their case, and then compared them. And then when you establish a set of strings, instead of letting the default argument take over, go ahead and use your case insensitive compare, and then now you have a set of strings that operates case insensitively. So you can change the ordering, adapt the ordering, whatever you like for the primitives, as well as supply the necessary one for the things the built-ins don't have properties for. So then that's that piece of code right there.

All right. So does that make sense? Well, now you know kind of the whole range of things that are available in the class library. All right, so we saw the four sequential containers, the vector, stack, queue, and the grid that you kinda indexed ordering, and kinda allowed you to throw things in and get them back out. We went through the map, which is the sort of fancy heavy lifter that does that key value lookup, and then we've seen the set, which does kind of aggregate collection management, and very efficient operations for kind of searching, retrieving, ordering, joining with other kind of sets, and stuff that also has a lot of high utility. I wanna do one quick little program with you before I start recursion just because I think it's kinda cool is to talk a little about this idea of like once you have these ADTs, you can solve a lot of cool problems, and that's certainly what this week's assignment is about. It's like well here are these tasks that if you didn't have – So ADTs, just a reminder – I say this word as though everybody knows exactly what it means – is just the idea of an abstract data type. So an abstract data type is a data type that you think of in terms of what it provides as an abstraction. So a queue is this FIFO line, and how it works internally, what it's implemented as, we're not worried about it at all. We're only worried about the abstraction of enqueue and dequeue, and it coming first in first out.

So we talk about ADTs, we say once we have a queue, a stack, or a vector, we know what those things do, what a mathematical set is about. We build on top of that. We write code that leverages those ADTs to do cool things without having to also manage the low level details of where the memory came from for these things, how they grow, how they search, how they store and organize the data. You just get to do real cool things.

So you probably got a little taste of that at the end of 106A when you get to use the array list and the hashmap to do things. This set kinda just expands out to fill out some other niches where you can do a lot of really cool things because you have these things around to build on. So one of the things that happens a lot is you tend to do layered things, and you'll see a little bit of this in the assignment you're doing this week where it's not just a set of something, it's a set of a map of something, or a vector of queues, a map of set. So I gave you a couple of examples here of the things that might be useful.

Like if you think of what a smoothie is, it's a set of things mixed together, some yogurt, and some different fruits, some wheatgrass, whatever it is you have in it. And that the menu for a smoothie shop is really just a bunch of those sets, so each set of ingredients is a particular smoothie they have, and then the set of all those sets is the menu that they post up on the board you can come in and order. The compiler tends to use a map to keep track of all the variables that are in scope. As you declare variables, it adds them to the map so that when you later use that name, it knows where to find it. Well, it also has to manage though not just one map, but the idea is as you enter and exit scopes, there is this layering of open scopes. So you have some open scope here. You go into a for loop where you open another scope. You add some new variables that when you look it actually shadows then at the nearest definition, so if you had two variables of the same, it needs to look at the one that's closest. And then when you exit that scope, it needs those variables to go away and no longer be accessible. So one model for that could be very much a stack of maps where each of those maps represents the scope that's active, a set of variable names, and maybe their types and information is stored in that map.

And then you stack them up. As you open a scope, you push on a new empty map. You put things into it, and then maybe you enter another scope. You push on another new empty map. You stick things into it, but as you exit and hit those closing braces, you pop those things from the stack to get to this previous environment you were in. So let me do a little program with you. I just have this idea of how this would work, and we'll see if I can code something up. So I have – let's go back over here. I'm going to – this is the piece of code that just reads words, so that's a fine piece of code to have here. I have a file here – let me open it up for you – that just contains the contents of the Official Scrabble Players' Dictionary, Edition 2. It's got a lot of words it. It's pretty long. It's still loading. Let's go back and do something else while it's loading.

It happened to have about 120,000 words I think is what it would be right now. So it's busy loading, and I have this question for you. There are certain words that are anagrams of each other. The word cheap can be anagrammed into the word peach, things like that. And so I am curious for the Official Scrabble Players' Dictionary what – so if you imagine that some words can be anagrammed a couple times, five or six different words just on how you can rearrange the letters. I'm curious to know what the largest anagram cluster is in the Official Scrabble Players' Dictionary.

So I'd like to know across all 127,000 words that they form little clusters, and I'd like to find out what's the biggest of those clusters. Okay. That's my goal. So here's what I've got going. I've got something that's gonna read them one by one. So let's brainstorm for

a second. I want a way to take a particular word and kinda stick it with its other anagram cluster friends. What's a way I might do that? Help me design my data structure. Help me out. [Inaudible]. I've got the word peach. Where should I stick it so I can –

Student: You could treat each string as like a set of –

Instructor (Julie Zelenski):So I have this string, which represents the letters. I've got the word peach. I wanna be able to stick peach with cheap, so where should I stick peach in such a way that I could find it. And you've got this idea that the letters there are a set.

They're not quite a set, though. Be careful because the word banana has a couple As and a couple Ns, and so it's not that I'd really want it to come down to be the set BAN. I wouldn't wanna coalesce the duplicates on that, so I really do wanna preserve all the letters that are in there, but your idea's getting us somewhere. It's like there is this idea of kind of like for any particular word there's the collection of letters that it is formed from. And somehow if I could use that as an identifier in a way that was reliable – anybody got any ideas about what to do with that?

Student: If you did that a vector, each letter and then the frequency of each letter in the word?

Instructor (Julie Zelenski):So I could certainly do that. I could build kind of a vector that had kinda frequencies, that had this little struct that maybe it was number of times it occurs. Then I could try to build something on the basis of that vector that was like here is – do these two vectors match? Does banana match apple? And you'd say well no. It turns out they don't have the same letters. I'm trying to think of a really easy way to represent that. Your idea's good, but I'm thinking really lazy. So somebody help me who's lazy.

Student: Could you have a map where the key is all the letters in alphabetical order and the value is a vector of all the –

Instructor (Julie Zelenski):Yeah. That is a great idea. You're taking his idea, and you're making it easier. You're capitalizing on lazy, which is yeah – I wanna keep track of all the letters that are in the word, but I wanna do it in a way that makes it really easy for example to know whether two have the same – we'll call it the signature. The signature of the word is the letter frequency across it.

If I could come up with a way to represent the signature that was really to compare two signatures quickly to see if they're the same, then I will have less work to do. And so your idea is a great one. We take the letters and we alphabetize them. So cheap and peach both turn into the same ACEHP. It's a nonsense thing, but it's a signature. It's unique for any anagram, and if I use a map where that's the key, then I can associate with every word that had that same signature. So let's start building that. So let me write – I wanna call this the signature. Given a string, it's going to alphabetize it. I'm going to write the dumbest version of this ever, but I'm just gonna use a simple sorting routine on this. So

smallest is gonna small index – we'll call it min index. That's a better name for it. Min index equals I, and then I'm gonna look through the string, and I'm gonna find the smallest letter that's there and move it to the front. That's basically gonna be my strategy. I've got the wrong place to start, though. I'm gonna look from I to the one.

So if S sub J is less than S sub min index, so it is a smaller letter, then the min index gets to be J. So far, what I've done is I've run this loop that kind of starts in this case on the very first iteration and says the first character in Slot 0, that must be the min. And then it looks from there to the end. Is there anything smaller? Whenever it find anything smaller, it updates the min index, so when it's done after that second loop has fully operated, then min index will point to the smallest alphabetically character in the string that we have. Then I'm gonna swap it to the front. So S sub I with S sub min index, and I'll write a swap because swap is pretty easy to write. See about how we do this, okay. So if I – I like how this works, and then let me stop and say right here, now is a good time to test this thing. I just wrote signature. It probably works, but it potentially could not. And I'm just gonna show you a little bit of how I write code. This is good to know. As I say, I put some code in here that I plan on throwing away. Enter word – and then I throw it through my function. I think I called it signature, didn't I? Signature S – so the idea being if it doesn't work, I wanna find out sooner rather than later. It doesn't like my use of get line. Is that because it's not included? Yes, it's not. So let's go get the right header files. This is half the battle sometimes is figuring out what headers are needed to make your compiler happy.

So now it's up here at enter word, and I say what does cheap come out as? It goes into the debugger. That's good. So it wasn't right. Now we're gonna get to find out what does it not like about that. It says – did we forget to return something? Let's go look at our code. So it's complaining about – oh yeah, I can see where we're gonna get in trouble here. It's complaining that the return – it was saying that I'm trying to print something and it looks like garbage, that what I'm trying to print didn't make sense at all. I could say well that's funny. Let's go look at signature.

Student:[Inaudible] pass the [inaudible].

Instructor (Julie Zelenski):That's probably a good idea. We ought to fix that while we're there. Let me leave that bug in for a minute because I'm gonna fix my first bug, and then I'll come back to that one. So it turns out what it's really complaining about though is it has to do with – I said well what does signature return. Somehow, what's being returned by signature is being interpreted as total crap when it got back to main, and there's a very good reason for that because I never returned anything. So maybe if I had been less cavalier about the fact that it was giving me a warning over here that said control reaches the end of non-void function, but I was being lazy and I didn't look – was that I didn't pay attention to the fact.

So let's leave my other bug in that you've already pointed out because this is exactly what it's like when you're doing it. You enter words and you say cheap, and it says cheap, and you're like no. And then you're like about how about an. Hey look, that's in

alphabetical order. And then you spend all your time thinking about words for a while. Well, tux. That's in alphabetical order. It seems to work. You could do that for a while, but then you're like this whole cheap, not good. Okay, so I come back here and my friend who's already one step ahead of me has pointed out that my swap is missing the all important pass by reference that as it is it what swapping the copies that got passed to the function, but of course nothing was happening back here in signature land. So if I fix that, and I come back in here, I'm gonna feel better about this. Oh, my gosh. And even tux still works. And if I say banana, I should get a bunch of As, a bunch of B and an N, so there's various cases to test out if you have multiple letters, and if you have letters that are the same letters like apple so that it doesn't lose your duplicates, and it seems to come out right. So given this, the word peach should come down to the same signature as cheap, and so that seems to indicate we're on the path toward building the thing we wanted to build. So I have this read file thing that I have left over from last time that reads each word, and I wanna change my vector into a map of set of string. What capitalization do you not like?

Student:[Inaudible].

Instructor (Julie Zelenski):Yeah. So it turns out this file happens to all be lower case, but there's no harm in doing this. That way even if they weren't, it'll take care of that problem for us if we wanted it to.

Student:[Inaudible].

Instructor (Julie Zelenski):I want lower case. Oh yeah, I do. Well, your case. You guys are so picky. All right. Here's my deal. I'm gonna take my map, and this is gonna be a line of code that's gonna make your head spin. Just go with it. This is the do all craziness. Okay. So I've got this map, and what I wanna do is under the signature of this word, I want to look up the set of strings that's associated with it and tack this one in. And the add in this case with the set, I know that's gonna do non-duplication. In fact, the file doesn't contain duplicates, but if it did, I certainly don't want it to record it twice anyway, so I might as well do this. Now this form of this is a heavy lifter for a small piece of code. The signature then went and converted into the ACEHP form. I used that as the key into the table. If it was already there, it's gonna retrieve me an existing set that I'm gonna just go ahead and add a word onto. If it wasn't there, the behavior for the brackets is to create kind of a new empty value for that. So it'll use the key and create a default value for type.

Well, the default value for set of string – so if you just create a set without any other information in the default constructor will always create you a nice clean empty set. So in fact, it will get me exactly what I want which is to put it in there with an empty set that I will immediately add the word into. So after I do this, I should have this fully populated map. And then I'm gonna do this just as a little test. I'm gonna say num words when I'm done to feel a little bit confident about what got put in there. How about I call it – that's a good idea. It's so fussy. C++ never does what you want. I think I called this thing OSPD2.txt that has the words in it. And then I need to declare the variable that I'm

sticking all this stuff into is a set. So go in, load stuff, doing it's thing, the number of words 112,882. Okay. That's close enough. I can't remember the number that's in there, but that sounds like a fine approximation of it. So I feel like it did sort of manage to do something for it. And I can actually do this if I just wanna get a little glimpse of it is to use my iterator to look at something that's in there. Wait, is map a set of string? Iterator -- file iter.hasNext. I'm gonna say key equals iter.next, and I'm going to print that key, and I'm going to print the size of the set because I'm at this point – I should see gobs of printing come out of this thing.

It takes a little bit of a while to process the thing, and then see gobs and gobs of stuff going by. It looks like a lot of things are ones if you can imagine reading them as they go by because a lot of words are really just not anagrams of anything else. But some of the shorter ones have sort of a better chance. So you can find out here at the end that there are EEIKLPST. I don't know what that is. Leakiest? No. I don't know what that word is. I should write it in for any of them. This dictionary has a bunch of really crazy words in it too, so it makes it especially challenging. What is that? I don't know. That one almost looks like beginners, but it's got an F in it. It's the F beginners, a very famous word. You guys have probably heard of it. So I've seen that, and now I wanna do the thing where I will pick the largest one. I'd like to know. Somebody should tell me. Int max size [cuts out] max key, so I'll set this to be zero, and max key is that. And then I'm gonna do this. If the size of this key is greater than my max size, then it's gonna get to be the new key. So after I did all of that then [cuts out] key. And then I probably wanna see what they are, so why don't I go ahead and take a look. Ah, I have to go to the type, though. [Inaudible] to equals M of key – max key, I guess, dot iterator, [inaudible] IT has next, CLIT.next [inaudible]. So it went back. It found the maximum, in this case using the size of the sets as the distinguishing feature. And then max is AEPRS, which it's got a big old list of about 12. I think that's 12, actually. Maybe 13.

So now you know. You can impress your friends at parties. This is the kind of thing you can win bar bets on. Oh, yeah. What's the size of the largest anagram cluster? Everybody wants to know this kind of stuff. I can't believe you guys can sleep at night without actually knowing this. And what's neat to know though [inaudible] just to point out a couple of things that – you can use a little decomposition on this code, but there's kind of a very small amount of things we're having to do. For example, one of the things that's really powerful, things like the map where we can just figure out how to key the things to store the collection right under that key, then looking it up and adding something is a very efficient sort of direct operation, just building on these things and it going through and doing all the work of storing them, sorting them, making it efficient for us to retrieve them and look them up such that I can process 100,000 words in the blink of an eye, and then go back through, look at them all, find the biggest one, get my information out.

Student: When you make the call to M.size, is that the number of words? [Inaudible].

Instructor (Julie Zelenski):That is the number of keys.

Student: Keys, right. So that's not actually [inaudible].

Instructor (Julie Zelenski):Yeah. So it doesn't know anything about everything else that was [inaudible], but in fact that's why it's [inaudible]. I know the dictionary has about 127,000 words. It turns out they form about 112 unique signatures, and so there's actually another 20,000 words that are clung onto some existing signature. That's the number of unique signatures across the dictionary, not the number of words, so that's probably the wrong name to call it.

Student: For the M signature word thing where [inaudible] of the default just to create a new stack, that works as well for vectors [inaudible]?

Instructor (Julie Zelenski):Yeah. It works for anything if you were just to declare it on the stack and the right thing happened, so vectors, set, maps. All those things do. But the primitive types like int and double, it doesn't. So it would work for string. String is an empty string.

So for some of the fancier, more modern types tend to actually know how to just default construct themselves into a good state, but the primitives don't do that. So if you were having a map of ints and you wanted to have them start at zero, you need to really start them at zero. You can call just M sub this. It would be garbage, and it would just be operating with garbage from that way forward. All right. Well, we're good. What I'm gonna give you is the eight minute discussion of recursion that whets your appetite for the things we're gonna be doing next week.

So recursion is one of those things I think that when you haven't yet had a chance to explore it first hand and other people tell you about it, it has sort of an awe inspiring sort of mystery, some fear, and whatnot. So first off, I wanna kinda shake that fear off. It is a little bit hard to wrap your head around the first time you see it, but we're gonna have a whole week's worth of time to spend on it, so we're gonna try to give you a lot of different ways to think about it, and different problems to see to kinda help you do it. And I think once you do get your head around it, it turns out then you'll discover how infinitely powerful it is, that there is kind of a simple idea in it that once you kinda fully get your head around, you can explore and solve lots of different problems using this just one technique again and again.

So in itself, it's a little bit mysterious at first glance, but then once you kind of master it, you'll be amazed at the kind of neat things you can do with it. So it is certainly what I'd consider an indispensable tool in a programmer's tool kit. The kind of problems you can solve using the techniques you have so far is fundamentally limited. And part of what we need to do in this class is expose you to these new ways of solving harder, more sophisticated problems that the old techniques don't work for. One of the cool things about recursion is it actually lends very simple, elegant, short solutions to problems that at first glance seem completely unsolvable. That if you can formulate a structure for [inaudible], you will discover that the code is not long to write. It's not tedious to write. The tricky part is to figure out how to express it, so it's more of a thinking puzzle than it is a coding puzzle. I certainly like thinking puzzles as much as coding puzzles if not more.

The general sort of idea is that you are going to try to solve a problem – instead of sort of breaking it down into component tasks like if I need to make dinner, I need to go to the store and buy things, and I need to come home, and chop them, and get the recipe. You think of what your standard decomposition is all about – breaking down your tasks into A, B, and C, and D, and then you add them all together to get the whole task done. Recursion has this kind of very different way of thinking about the problem, which is like well if I needed to get Task A done, and I had Task A prime, which was somehow a lot like the task I was trying to solve, but it somehow was a little bit simpler, a little bit easier, a little bit more manageable than the one I started out to solve, and if I had that solution – so if somehow I could delegate it off to some minion who works for me, and then I could use that to solve my problem, then my job would be made much easier by using that result of solving a similar problem that's a little bit [inaudible].

Okay, that seems a little bit wacky. Let me give you sort of an example of how this might work. So your standard problem, I said yeah, it's like you do these dissimilar sub problems. Let's imagine I had this goal where I wanted to survey the Stanford student body. I don't want just like a haphazard most of the people involved. Let's say I really wanted to get input from every single person on campus whether they think having cardinal as your mascot is a ridiculous choice or not. So let's imagine I really wanna hear from all 10,000 students. Now I can stand out in White Plaza with a big note pad and try to accost people and sort of work my way down the list. And then I'd be there for eons and never solve my problem. Instead, what I decide to do is I say well I'm gonna recruit some people to help me because I'm lazy as we've already established, and I would like to get some other people to join in my quest to answer these burning questions and to solve the survey.

So what I do is I round up ten people let's say, and I say would you help me, and I decide to divide the campus into kind of ten partitions. And I say if you could survey all the people whose names begin with A, B, and C, that would really help. And if you could do [inaudible] Gs, and if you would do – and if I divide up the alphabet that way, give each of them two or three letters, and I say if you would go get the data, it'd be really easy for me to do my job then. If I just took all their data [inaudible]. Well, being the kind of lazy person that I am, it's likely that the ten people I recruit would have similar lazy qualities because lazy people hang out with other lazy people. And so the person who was in charge of A, B, C, the first thing they do is turn around and find ten more friends, and then they divide it up and say could you do the AA through AM and so on. If they divide it into these pools of one tenth of what they were responsible for, and say you can go get the information from these people, and if they did the same thing – so if everybody along the road. We started with 10,000. Now each person had 1,000 to survey. They asked their friend to do 100. Their friend asked ten people to do ten. And then at some point, the person who has ten says well I just need to ask these ten people. Once I get their data, we don't need to do anything further.

So at some point the problem becomes so small, so simple, even though it was kind of the same problem all along. I just reproduced the same problem we had, but in a slightly more tractable form, but then I divided it around. Divide and conquer sometimes they call

this to where I spread out the work around a bunch of people to where each person's contribution is just a little part of the whole. You had to find the ten volunteers around underneath you and get their help in solving the problem, but nobody had to do much work, and that's kind of a really interesting way to solve a problem. It sounds like a very big problem of surveying 10,000 people, but by dividing and conquer, everybody has a little tiny role to play. You leverage a lot of people getting it done, and there is this self-similarity to it, which is kind of intriguing that everybody is trying to solve the same problem but just at different levels of scale. And so this idea applies to things like phone trees rather than trying to get the message out to everybody in my class, it might be that I call two people who call two friends who call two friends until everybody gets covered. Nobody does the whole job. Everybody just does a little part of it.

Sometimes you'll see these fractal drawings where there is a large leaf which when you look closer actually consists of littler leaves, which themselves are littler leaves, so that at every level of scale the same image is being reproduced, and the result kind of on the outside is something that in itself if you look deeper has the same problem but smaller embedded in it. So it's a pretty neat sort of way of solving things. I am gonna tell you a little bit about how the code looks, and then I really am not gonna be able to show you much code today. I think it's actually even better to show you this code on Monday when we can come back fresh, but that it involves taking – So we're looking at functional recursion first, and functional recursion is taking some sort of functions, a function that takes an input and returns an answers, returns non-void thing that comes back. And we're gonna be looking at problems where you're trying to solve this big problem, and that if you had the answer to making a call to yourself on a smaller version of the problem – maybe one call, maybe two calls, maybe several calls – that you could add those, or multiply those, or combine those in some way to answer the bigger problem. So if I were trying to solve this campus survey, having the answers to these smaller campus surveys gives me that total result. And so the – I really should not try to do this in two minutes. What I should do is try to tell you on Monday. We'll come back and we'll talk about recursion, and it will be an impressive week. Meanwhile, work on your ADT homework.

[End of Audio]

Duration: 50 minutes

ProgrammingAbstractions-Lecture08

Instructor (Julie Zelenski):Good afternoon. You guys are talkative. That sounds like a good weekend. I heard like some low noise on Assignment 2, and I thought we could elevate that to a little bit broader level, so if there's somebody who's already gotten fairly far on one or both of the problems on there, and has stumbled across some insight that perhaps was a little bit painful to get through, but you'd like to provide the benefit to your fellow classmates, now would be a great time to offer any bit of advice you think that might make the world a happier place.

Anyone? No one? [Inaudible]. You're sitting in the wrong seat. You don't sit there.

Student: When you make an iterator of like a map or [inaudible] like that, capitalize the word iterator.

Instructor (Julie Zelenski):Okay. So this insight is that the iterator class that is nested within the map and nested within the set, its name really is capital I iterator, so it's capital M map of something you're putting in there, int, whatever, and then colon colon capital I iterator.

It's pretty easy to look at the code and be a little bit – the lower case I and capital I often in small fonts look about the same, too.

Student:[Inaudible].

Instructor (Julie Zelenski):I've no doubt, and that actually is a great bit of insight is that you think the error would be like "Spell it the right way, doofus," and C++ never gives you the error and says here's what you'd like to do to fix it. It says according to my internal specs of which I have referenced point Article 72 of the C++ standard and all this stuff that is just total gobbledygook.

So one of the things you get very good at is letting the compiler direct you to where to look for the problem, but then ignoring what it told you was the problem because its analysis of it was often not very helpful. You just get good at kind of looking at the line yourself to figure it out. Okay?

Student:I keep forgetting to clear the classes or renew like objects and things like that.

Instructor (Julie Zelenski):Yeah. So all those pound includes, right? So C++, very fussy about that. You start using set. You start using map. You need to make sure you've got that pound include of the set and the map and whatnot, and that none of the code you write that will use it will make any sense to the compiler until it's been fully informed about what map and set look like. So if you don't do that – And same thing for like iostream and all these other things you're using the random library, when you start making those calls, C++ wants to know what the interfaces are and how they're set up.

And the way to tell it about it is to get the right pound includes in there. Without them, it won't get you very far at all.

Student:I have a question. Why isn't it possible to just give access to every [inaudible]?

Instructor (Julie Zelenski):Well, you could certainly do that. We could make one sort of big master include, which is like include the whole world, and just grab everything and bring it on in. And what that will do is it will just slow down all your compiles because it will be looking through that thing again and again every single time. And even though you're not using it, it will have to have kind of read past it and understood it. And then it turned out to be a wasted effort. In the case of our programs, the amount of headers we're talking about is small enough that you could imagine doing that without really slowing your development down. But for the large-scale development, you will typically include exactly those headers you need to avoid slowing down all your compiles by rereading a lot of things that aren't necessary for this unit.

Student: Careful when you concatenate characters to strings.

Instructor (Julie Zelenski):Yes.

Student: Because it will give you – because if you do it a certain way, it will give you a bunch of junk.

Instructor (Julie Zelenski):That is a great point. And we talked about that at strings, but it's a fine time to reiterate it – is that when you're using [cuts out] plus equal to take a string and extend it with some new characters or another string, one of those operands needs to be a C++ string, which means a variable, a parameter, something that was declared and created as a string.

And the things in double quotes remember are old style strings, and until they have been converted they are still old style. So if you have an expression that looks like this, and you try to add a character there, this compiles but it does not do what you want. It takes the old style string and uses this character kind of as an offset and causes the string basically to turn into garbage. So if you were to say something equals this, you will not get what you want. You're expecting to get my string extended with that new character. So remember that one or both the arguments needs to be a string. If you ever need to force, you can introduce that typecast around it, and does the promotion. In most situations, the promotion will happen automatically and you won't have to be involved, but this is one case where the legacy of C++ being derived from C means that the old language did have a meaning for this, and they couldn't break that old meaning, so they left it in place for you to stumble across when you least expect. So that's a great thing to always be keeping in mind when you're doing that concatenation, looking at your arguments, making sure one of them at the very least [cuts out] C++ screen. Way over there.

Student: Why does the set pair operator need to have ordering?

Instructor (Julie Zelenski):Because it actually is more than just a quality. So he's asking [cuts out] just say yes, no, are they the same? It's really using sorting to [cuts out] are you exactly like this one, but should I put you to the things that are smaller or the things that are larger because it's largely using that to kind of quickly throw away the parts of the set that it doesn't need to explore to find a match. If you actually made yours just equalities that just returned zero and one all the time –

Student: Yeah, and it doesn't work.

Instructor (Julie Zelenski):It definitely doesn't work. It will end up just losing things. You'll put them in the set, and it won't be able to find them again. That's because you told it they would be one place, and in fact, they didn't get put there because in effect your comparison looks a little random. So it really does want full ordering. We'll keep going. So Assignment 2 is coming in this Wednesday, right? So hopefully you're making good progress on that, and then we will get out your third assignment then, which will be your recursion problem set, which allows you to practice on recursion. We're gonna be talking about recursion all week. The reader chapters that go along with this pretty much lecture for lecture 4, 5, and 6, and this is the place where I just encourage you again. I think that's some of the best material in the reader, so I encourage you to make some time to do the reading, especially in advance of lecture will pay off the best. So we talked just a little bit about the vocabulary of recursion at the end of Friday. It was very rushed for time, so I'm just gonna repeat some of those words to get us thinking about what the context for solving problems recursively looks like. And then we're gonna go along and do a lot of examples [cuts out]. So the idea is that a recursive function is one that calls itself. That's really all it means at the most trivial level. It says that in the context of writing a function binky, it's gonna make a call or one or more calls to binky itself past an argument as part of solving or doing some task. The idea is that we're using that because the problem itself has some self-similarity where the answer I was seeking – so the idea of surveying the whole campus can actually be thought of as well if I could get somebody to survey this part of campus and this part of campus, somebody to survey all the freshmen, somebody to [cuts out] and whatnot, that those in a sense are – those surveys are just smaller instances of the same kind of problem I was originally trying to solve. And if I could recursively delegate those things out, and then they themselves may in turn delegate even further to smaller but same structured problems to where we could eventually get to something so simple – in that case of asking ten people for their input that don't require any further decomposition – we will have worked our way to this base case that then we can gather back up all the results and solve the whole thing. This is gonna feel very mysterious at first, and some of the examples I give you'll say I have really easy alternatives other than recursion, so it's not gonna seem worth the pain of trying to get your head around it. But eventually, we're gonna work our way up to problems where recursion really is the right solution, and there are no other alternatives that are obvious or simple to do. So the idea throughout this week is actually just a lot of practice. Telling you what the terms mean I think is not actually gonna help you understand it. I think what you need to see is examples. So I'm gonna be doing four or five examples today, four or five examples on Wednesday, and four or five examples on Friday that each kind of build on each other, kind of take the ideas and get a little more

sophisticated. But by the end of the week, I'm hoping that you're gonna start to see these patterns, and realize that in some sense the recursive solutions tend to be more alike than different. Once you have your head around how to solve one type of problem, you may very well be able to take the exact same technique and solve several other problems that may sound different at first glance, but in the end, the recursive structure looks the same. So I'd say just hold the discomfort a little bit, and wait to see as we keep working which example may be the one that kind of sticks out for you to help you get it through. So we're gonna start with things that fit in the category of functional recursion. The functional in this case just says that you're writing functions that return some non-void thing, an integer, a string, some vector of results, or whatever that is, and that's all it means to be a functional recursion. It's a recursive function that has a result to it. And because of the recursive nature, it's gonna say that the outer problem's result, the answer to the larger problem is gonna be based on making calls, one or more calls to the function itself to get the answer to a smaller problem, and then adding them, multiplying them, comparing them to decide how to formulate the larger answer. All recursive code follows the same decomposition into two cases. Sometimes there's some subdivisions within there, but two general cases. The first thing – this base case. That's something that the recursion eventually has to stop. We keep saying take the task and break it down, make it a little smaller, but at some point we really have to stop doing that. We can't go on infinitely. There has to be some base case, the simplest possible version of the problem that you can directly solve. You don't need to make any further recursive calls. So the idea is that it kinda bottoms out there, and then allows the recursion to kind of unwind. The recursive cases, and there may be one or more of these, are the cases where it's not that simple, that the answer isn't directly solvable, but if you had the answer to a smaller, simpler version, you would be able to assemble that answer you're looking for using that information from the recursive call. So that's the kind of structure that they all look like. If I'm at the base case, then do the base case and return it. Otherwise, make some recursive calls, and use that to return a result from this iteration. So let's first look at something that – the first couple examples that I'm gonna show you actually are gonna be so easy that in some sense they're almost gonna be a little bit counterproductive because they're gonna teach you that recursion lets you do things that you already know how to do. And then I'm gonna work my way up to ones that actually get beyond that. But let's look at first the raise to power. C++ has no built-in exponentiation operator. There's nothing that raises a base to a particular exponent in the operator set. So if you want it, you need to write it, or you can use – there's a math library called pow for raise to power. We're gonna run our own version of it because it's gonna give us some practice thing about this. The first one I'm gonna show you is one that should feel very familiar and very intuitive, which is using an iterative formulation. If I'm trying to raise the base to the exponent, then that's really simply multiplying base by itself exponent times. So this one uses a for loop and does so. It starts the result at one, and for iterations through the number of exponents keeps multiplying to get there. So that one's fine, and it will perfectly work. I'm gonna show you this alternative way that starts you thinking about what it means to divide a problem up in a recursive strategy. Base to the exponent – I wanna raise five to the tenth power. If I had around some delegate, some clone of myself that I could dispatch to solve the slightly smaller problem of computing five to the ninth power, then all I would need to do is take that answer and multiply it by one more five,

and I'd get five to the tenth. Okay. If I write code that's based on that, then I end up with something here – and I'm gonna let these two things go through to show us that to compute the answer to five to the tenth power, what I really need is the answer to five to the ninth power, which I do by making a recursive call to the same function I'm in the middle of writing. So this is raise that I'm defining, and in the body of it, it makes a call to raise. That's what is the mark of a recursive function here. I pass slightly different arguments. In this case, one smaller exponent which is getting a little bit closer to that simplest possible case that we will eventually terminate at where we can say I don't need to further dispatch any delegates and any clones out there to do the work, but if the exponent I'm raising it to is zero, by definition anything raised to the exponent of zero is one, I could just stop there. So when computing five to the tenth, we're gonna see some recursion at work. Let me take this code into the compiler so that we can see a little bit about how this actually works in terms of that. So that's exactly the code I have there, but I can say something that I know the answer to. How about that? First, we'll take a look at it doing its work, so five times five times five should be 125. So we can test a couple little values while we're at it. Two the sixth power, that should be 64, just to see a couple of the cases, just to feel good about what's going on. And then raising say 23 to the zero power should be one as anything raised to the zero power should be. So a little bit of spot testing to feel good about what's going on. Now I'm gonna go back to this idea like two to the sixth. And I'm gonna set a breakpoint here. Get my breakpoint out. And I'm gonna run this guy in the debugger. Takes a little bit longer to get the debugger up and running, so I'll have to make a little padder up while we're going here. And then it tells me right now it's breaking on raise, and I can look around in the debugger. This is a – did not pick up my compilation? I think it did not. I must not have saved it right before I did it because it's actually got the base is 23 and the exponent is zero. It turns out I don't wanna see that case, so I'm gonna go back and try again. I wanna see it – no, I did not. And I'm just interested in knowing a little bit about the mechanics of what's gonna happen in a recursive situation. If I look at the first time that I hit my breakpoint, then I'll see that there's a little bit of the beginnings of the student main, some stuff behind it. There's a little bit of magic underneath your stack that you don't really need to know about, but starting from main it went into raise, and the arguments it has there is the base is two, the exponent is six. If I continue from here, then you'll notice the stack frame got one deeper. There's actually another indication of raise, and in fact, they're both active at the same time. The previous raise that was trying to compute two to the sixth is kind of in stasis back there waiting for the answer to come back from this version, which is looking to raise two to the fifth power. I continue again, I get two to the fourth. I keep doing this. I'm gonna see these guys kinda stack up, each one of those kind of waiting for the delegate or the clone to come back with that answer, so that then it can do its further work incorporating that result to compute the thing it needed to do. I get down here to raising two to the first power, and then finally I get to two to the zero, so now I've got these eight or so stacked frames, six up there. This one, if I step from here, it's gonna hit the base case of returning one, and then we will end up kind of working our way back out. So now, we are at the end of the two to the one case that's using the answer it got from the other one, multiplying it by two. Now I'm at the two to the two case, and so each of them's kind of unfolding in the stack is what's called unwinding. It's popping back off the stack frames that are there and kind of revisiting them, each passing up the

information it got back, and eventually telling us that the answer was 64, so I will let that go. So the idea that all of those stack frames kind of exist at the same time – they're all being maintained independently, so the idea that the compiler isn't confused by the idea that raise is invoking raise which is invoking raise, that each of the raise stack frames is distinct from the other ones, so the indications are actually kept separate. So one had two to the sixth, the next one had two to the fifth, and so on. And then eventually we need to make some progress toward that base case, so that we can then stop that recursion and unwind. Let me actually show you something while I'm here, which is one thing that's a pretty common mistake to make early on in a recursion is to somehow fail to make progress toward that base case or to – not all cases make it to the base case. For example, if I did something where I forgot to subtract one [cuts out], and I said oh yeah, I need to [cuts out]. In this case, I'm passing it exactly the same arguments I got. If I do this and I run this guy, then what's gonna happen is it's gonna go two to the sixth, two to the sixth, two to the sixth, and I'm gonna let go of this breakpoint here because I don't really wanna watch it all happening. And there it goes. Loading 6,493 stack frames, zero percent completed, so that's gonna take a while as you can imaging. And usually, once you see this error message, it's your clue to say I can cancel, I know what happened. The only reason I should've gotten 6,500 stack frames loaded up is because I made a mistake that caused the stack to totally overflow. So the behavior in C++ or C is that when you have so many of those stack frames, eventually the space that's been allocated or set aside for the function stack will be exhausted. It will use all the space it has, and run up against a boundary, and typically report it in some way that suggests – sometimes you'll see stack overflow, stack out of memory error. In this case, it's showing you the 7,000 stack frames that are behind you, and then if you were to examine them you would see they all have exactly the same argument, so they weren't getting anywhere. I'm not gonna actually let it do that because I'm too impatient. Let me fix this code while I'm here. But other things like even this code that actually looks correct for some situations also has a subtle bug in it. Even if I fixed this, which is that, right now it assumed that the exponent is positive, that it's some number that I can subtract my way down to zero. If I actually miscalled raise, and I gave it a negative exponent, it would go into infinite recursion as well. If you started it at ten to the negative first power, it would go negative first, negative second, negative third. 6,500 stack frames later, we'd run out of space. In this case, since we're only intending to handle those positive powers, we could just put an error that was like if the exponent is less than zero then raise an error and don't even try to do anything with it. Okay. So let me show you a slightly different way of doing this that's also recursive, but that actually gets the answer a little bit more efficiently. This is a different way of dividing it up, but still using a recursive strategy which is that if I'm trying to compute five to the tenth power, but if I have the answer not of five to ninth power, but instead I have the answer of five to the fifth power, and then I multiply that by itself, I would get that five to the tenth power that I seek. And then there's a little bit of a case though of what if the power I was trying to get was odd, if I was trying to raise it to the eleventh power, I could compute the half power which get me to the fifth – multiplied by the fifth, but then I need one more base multiplied in there to make up for that half. Okay. And so I can write another recursive formulation here. Same sort of base case about detecting when we've gotten down, but then in this case the recursive call we make is to base of the exponent divided by two, and then we use it with an if else whether the

exponent was even or odd, it decided whether to just square that number or whether to square it and tack in another power of the base while we're at it. So this one is gonna be much more quick about getting down to it, whereas the one we saw was gonna put one stack frame down for every successive exponent power. So if you wanted to raise something to the 10th, or 20th, or 30th power, then you were giving yourself 10, 20, 30 stack frames. Something like 30 stack frames is not really something to be worried about, but if you were really trying to make this work on much larger numbers, which would require some other work because exponent is a very rapidly growing operator and would overflow your integers quickly, but this way very quickly divides in half. So it goes from a power of 100, to a power of 50, to 25, to 12, to 6, to 3, to 2, to 1, so that dividing in half is a much quicker way to work our way down to that base case and get our answer back, and we're doing a lot fewer calculations than all those multiplies, one per base. So just a little diversion. So let me tell you something, just a little bit about kind of style as it applies to recursion. Recursion is really best when you can express it in the most kind of direct, clear, and simple code. It's hard enough to get your head around a recursive formulation without complicating it by having a bunch of extraneous parts where you're doing more work than necessary, or redundantly handling certain things. And one thing that's actually very easy to fall in the trap of is thinking about there's lots of other base cases you might be able to easily handle, so why not just go ahead and call out for them, test for – you're at the base case. You're close to the base case. Checking before you might make a recursive call, if you're gonna hit the base case when you make that call, then why make the call. I'll just anticipate and get the answer it would've returned anyway. It can lead to code that looks a little bit like this before you're done. If the exponent's zero, that's one. If the exponent's one, then I can just return the base. If it's two, then I can just multiply the base by itself. If it's three, I can start doing this. Certainly, you can follow this to the point of absurdity, and even for some of the simple cases, it might seem like you're saving yourself a little bit of extra time. You're like why go back around and let it make another recursive call. I could stop it right here. It's easy to know that answer. But as you do this, it complicates the code. It's a little harder to read. It's a little harder to debug. Really, the expense of making that one extra call, two extra calls is not the thing to be worried about. What we really want is the cleanest code that expresses what we do, has the simplest cases to test, and to examine, and to follow, and not muck it up with things that in effect don't change the computational power of this but just introduce the opportunity for error. Instead of a multiply here, I accidentally put a plus. It might be easy to overlook it and not realize what I've done, and then end up computing the wrong answer when it gets to the base case of two. In fact, if you do it this way, most things would stop at three, but these would suddenly become kind of their own special cases that were only hit if you directly called them with the two. The recursive cases will all stop earlier. It just complicates your testing pass now because not everything is going through the same code. So we call this arm's length recursion. I put a big X on that looking ahead, testing before you get there. Just let the code fall through. So Dan, he's not here today, but he talked to me at the end of Friday's class, and it made me wanna actually just give you a little bit of insight into recursion as it relates to efficiency. Recursion by itself, just the idea of applying a recursive technique to solving a problem, does not give you any guarantee that it's gonna be the best solution, the most efficient solution, or the fact that it's gonna give you a very inefficient solution. Sometimes people

have kind of a bad rap on recursion. It's like recursion will definitely be inefficient. That actually is not guaranteed. Recursion often requires exactly the same resources as the iterative approach. It takes the same amount of effort. Surveying the campus – if you're gonna survey the 10,000 people on campus and get everybody's information back, whether you're doing it divide and conquer, or whether you're sitting out there in White Plaza asking each person one by one, in the end 10,000 people got survey. The recursion is not part of what made that longer or shorter. It might actually depending on how you have resources work out better. Like if you could get a bunch of people in parallel surveying, you might end up completing the whole thing in less time, but it required more people, and more clipboards, and more paper while the process is ongoing than me standing there with one piece of paper and a clipboard. But then again, it took lots of my time to do it. So in many situations, it's really no better or no worse than the alternative. It makes a little bit of some tradeoffs of where the time is spent. There are situations though where recursion can actually make something that was efficient inefficient. There are situations where it can take something that was inefficient and make it efficient. So it's more of a case-by-case basis to decide whether recursion is the right tool if efficiency is one of your primary concerns. I will say that for problems with simple iterative solutions that operate relatively efficiently, iteration is probably the best way to solve it. So like the raise to power, yeah you could certainly do that iteratively. There's not some huge advantage that recursion is offering. I'm using them here because they're simple enough to get our head around. We're gonna work our way toward problems where we're gonna find things that require recursion, which is kind of one of the third points I put in. Why do we learn recursion? What's recursion gonna be good for? First off, recursion is awesome. There are some problems that you just can't solve using anything but recursion, so that the alternatives like I'll just write it iteratively won't work. You'll try, but you'll fail. They inherently have a recursive structure to them where recursion is the right tool for the job. Often, it produces the most beautiful, direct, and clear elegant code. The next assignment that will go out has these problems that you do in recursion, and they're each about ten lines long. Some of them are like five lines long. They do incredible things in five lines because of the descriptive power of describing it as here's a base case, and here's a recursive case, and everything else just follows from applying this, and reducing the problem step by step. So you will see things where the recursive code is just beautiful, clean, elegant, easy to understand, easy to follow, easy to test, solves the recursive problem. And in those cases, it really is a much better answer than trying to hack up some other iterative form that may in the end be no more efficient. It may be even less efficient. So don't let efficiency be kind of a big fear of what recursion is for you. So I'm gonna do more examples. I've got three more examples, or four I think today, so I will just keep showing you different things, and hopefully the patterns will start to kind of come out of the mist for you. A palindrome string is one that reads the same forwards and backwards when reversed. So was it a car or a cat I saw, if you read that backwards, it turns out it says the same thing. Go hang a salami, I'm a lasagna hog. Also handy when you need to have a bar bet over what the longest palindrome is to have these things around. There are certainly ways to do this iteratively. If you were given a string and you were interested to know is it a palindrome, you could kind of do this marching – you're looking outside and kind of marching your way into the middle. But we're gonna go ahead and let recursion kinda help us deal with the subproblem of this,

and imagine that at the simplest possible form – you could say that a palindrome consists of an interior palindrome, and then the same letter tacked on to the front and the back. So if you look at was it a car or a cat I saw, there are two Ws there. It starts and it ends with a W, so all palindromes must start and end with the same letter. Okay. Let's check that, and if that matches, then extract that middle and see if it's a palindrome. So it feels like I didn't really do anything. It's like all I did was match two letters, and then I said by the way delegate this problem back to myself, making a call to a function I haven't even finished writing to examine the rest of the letters. And then I need a base case. I've got a recursive case, right? Take off the outer two letters. Make sure they match. Recur on the inside. What is the simplest possible palindrome?

Student: One letter?

Instructor (Julie Zelenski):One letter. One letter makes a good palindrome. One letter is by definition first and last letter are the same letter, so it matches. That's a great base case. Is it the only base case?

Student:[Inaudible].

Instructor (Julie Zelenski):Two letters is also kind of important, but there's actually an even simpler form, right? It's the empty string. So both the empty string and the single character string are by definition the world's simplest palindromes. They meet the requirements that they read the same forward and backwards. Empty string forwards and backwards is trivially the same, so that makes it even easier than doing the two-letter case. So if I write this code to look like that where if the length of the strength is one or fewer, so that handles both the zero and the one case, then I return true. Those are trivial palindromes of the easiest immediate detection. Otherwise, I've got a return here that says if the first character is the same as the last character, and the middle – so the substring starting at Position 1 that goes for length minus two characters that picks up the interior discarding the first and last characters – if that's also a palindrome, then we've got a palindrome. So given the short circuiting nature of the and, if it looks at the outer two characters and they don't match, it immediately just stops right there and says false. If they do match, then it goes on looking at the next interior pair which will stack up a recursive call looking at its two things, and eventually we will either catch a pair that doesn't match, and then this false will immediately return its way out, or it will keep going all the way down to that base case, hit a true, and know that we do have a full palindrome there. So you could certainly write this with a for loop. Actually, writing it with a for loop is almost a little bit trickier because you have to keep track of which part of the string are you on, and what happens when you get to the middle and things like that. In some sense, the recursive form really kinda sidesteps that by really thinking about it in a more holistic way of like the outer letters plus an inner palindrome gives me the answer I'm looking for. So this idea of taking a function you're in the middle of writing and making a call to it as though it worked is something that – they require this idea of the leap of faith. You haven't even finished describing how is palindrome operates, and there you are making a call to it depending on it working in order to get your function working. It's a very kind of wacky thing to get your head around. It feels a little bit

mystical at first. That feeling you feel a little bit discombobulated about this is probably pretty normal, but we're gonna keep seeing examples, and hope that it starts to feel a little less unsettling. Anybody wanna ask a question about this so far? Yeah?

Student: So I guess create your base case first, then test it? [Inaudible].

Instructor (Julie Zelenski):That's a great question. So I would say typically that's a great strategy. Get a base case and test against the base case, so the one character and the two character strings. And then imagine one layer out, things that will make one recursive call only. So in this case for the palindrome, it's like what's a two-character string? One has AB. One has AA. So one that is a palindrome, one that isn't, and watch it go through. Then from there you have to almost take that leap and say it worked for the base case. It worked for one out. And then you have to start imagining if it worked for a string of length N, it'll work for one of N plus one, and that in some sense testing it exhaustively across all strings is of course impossible, so you have to kind of move to a sort of larger case where you can't just sit there and trace the whole thing. You'll have to in some sense take a faith thing that says if it could have computed whether the interior's a palindrome, then adding two characters on the outside and massaging that with that answer should produce the right thing. So some bigger tests that verify that the recursive case when exercised does the right thing, then the pair together – All your code is going through these same lines. The outer case going down to the recursive case, down to that base case, and that's one of the beauty of writing it recursively is that in some sense once this piece of code works for some simple cases, the idea that setting it to larger cases is almost – I don't wanna say guaranteed. That maybe makes it sound too easy, but in fact, if it works for cases of N and then N plus one, then it'll work for 100, and 101, and 6,000, and 6,001, and all the way through all the numbers by induction. Question?

Student: You have to remove all the [inaudible], all the spaces?

Instructor (Julie Zelenski):Yeah. So definitely like the was it a car to match I should definitely be taking my spaces out of there to make it right. You are totally correct on that. So let me show you one where recursion is definitely gonna buy us some real efficiency and some real clarity in solving a search problem. I've got a vector. Let's say it's a vector of strings. It's a vector of numbers. A vector of anything, it doesn't really matter. And I wanna see if I can find a particular entry in it. So unlike the set which can do a fast contains for you, in vector if I haven't done anything special with it, then there's no guarantee about where to find something. So if I wanna say did somebody score 75 on the exam, then I'm gonna have to just walk through the vector starting at Slot 0, walk my way to the end, and compare to see if I find a 75. If I get to the end and I haven't found one, then I can say no. So that's what's called linear search. Linear kind of implies this left to right sequential processing. And linear search has the property that as the size of the input grows, the amount of time taken to search it grows in proportion. So if you have a 1000 number array, and you doubled its size, you would expect that doing a linear search on it should take twice as long for an array that's twice as large. The strategy we're gonna look at today is binary search which is gonna try to avoid this looking in every one of those boxes to find something. It's gonna take a divide and conquer strategy,

and it's gonna require a sorted vector. So in order to do an efficient lookup, it helps if we've done some pre-rearrangement of the data. In this case, putting it into sorted order is gonna make it much easier for us to be able to find something in it because we have better information about where to look, so much faster. So we'll see that surely there was some cost to that, but typically binary search is gonna be used when you have an array where you don't do a lot of changes to it so that putting it in a sorted order can be done once, and then from that point forward you can search it many times, getting the benefit of the work you did to put it in sorted order. If you plan to sort it just to search it, then in some sense all the time you spent sorting it would kind of count against you and unlikely to come out ahead. So the insight we're gonna use is that if we have this in sorted order, and we're trying to search the whole thing – we're looking for let's say the No. 75 – is that if we just look at the middlemost element, so we have this idea that we're looking at the whole vector right now from the start to the end, and I look at the middle element, and I say it's a 54. I can say if 75 is in this vector because it's in sorted order, it can't be anywhere over here. If 54's right there, everything to the left of 54 must be less than that, and 75 wouldn't be over there. So that means I can just discard that half of the vector from further consideration. So now I have this half to continue looking at which is the things that are the right of 54 all the way to the end. I use the same strategy again. I say if I'm searching a vector – so last time I was searching a vector that had 25 elements. Now I've got one that's got just 12. Again, I use the same strategy. Look at the one in the middle. I say oh, it's an 80, and then I say well the number I'm looking for is 75. It can't be to the right of the 80. It must be to the left of it. And then that lets me get rid of another quarter of the vector. If I keep doing this, I get rid of half, and then a quarter, and then an eighth, and then a 16th. Very quickly, I will narrow in on the position where if 75 is in this vector, it has to be. Or I'll be able to conclude it wasn't there at all. Then I kind of work on my in, and I found a 74 and a 76 over there, then I'm done. That base case comes when I have such a small little vector there where my bounds have crossed in such a way that I can say I never found what I was looking for. So let's walk through this bit of code that kind of puts into C++ the thing I just described here. I've got a vector. In this case, I'm using a vector that's containing strings. It could be ints. It could be anything. It doesn't really matter. I've got a start and a stop, which I identify the sub-portion of the vector that we're interested in. So the start is the first index to consider. The stop is the last index to consider. So the very first call to this will have start set to zero and stop set to the vector's size minus one. I compute the midpoint index which is just the sum of the start and stop divided by two, and then I compare the key that I'm looking for to the value at that index. I'm looking right in the middle. If it happens to match, then I return that. The goal of binary search in this case is to return the index of a matching element within the vector, or to return this not found negative one constant if it was unable to find any match anywhere. So when we do find it at whatever the level the recursion is, we can just immediately return that. We're done. We found it. It's good. Otherwise, we're gonna make this recursive call that looks at either the left half or the right half based on if key is less than the value we found at the midpoint, then the place we're searching is – still has the same start position, but is now capped by the element exactly to the left of the midpoint, and then the right one, the inversion of that, one to the right of the midpoint and the stop unchanged. So taking off half of the elements under consideration at each stage, eventually I will get down to the simplest possible case. And the simplest possible

case isn't that I have a one-element vector and I found it or not. The really simple case is actually that I have zero elements in my vector that I just kept moving in the upper and lower bound point until they crossed, which meant that I ran out of elements to check. And that happens when the start index is greater than the stop index. So the start and the stop if they're equal to each other mean that you have a one element vector left to search, but if you – For example, if you got to that case where you have that one element vector left to search, you'll look at that one, and if it matches, you'll be done. Otherwise, you'll end up either decrementing the stop to move past the start, or incrementing the start to move past the stop. And then that next iteration will hit this base case that said I looked at the element in a one-element vector. It didn't work out. I can tell you for sure it's not found. If it had been here, I would've seen it. And this is looking at just one element each recursive call, and the recursive calls in this case stack up to a depth based on the logarithm of the size to the power of two, so that if you have 1000 elements, you look at one, and now you have a 500-element collection to look at again. You look at one, you have a 250 element, 125, 60, 30, 15. So at each stage half of them remain for the further call, and the number of times you can do that for 1000 is the number of times you can divide 1000 by two, which is the log based two of that, which is roughly ten. So if you were looking at 1000 number array, if it's in sorted order, it takes you ten comparisons to conclusively determine where an element is if it's in there, or that it doesn't exist in the array at all. If you take that 1000 element array, and you make it twice as big, so now I have a 2000 number array, how much longer does it take?

Student: One more step.

Instructor (Julie Zelenski):One more step. Just one, right? You look at one, and you have a 1000 number array, so however long it took you to do that 1000 number array, it takes one additional comparison, kinda one stack frame on top of that to get its way down. So this means actually this is super efficient. That you can search so for example a million is roughly two the 20th power. So you have a million entry collection to search, it will take you 20 comparisons to say for sure it's here or not, and here's where I found it, just 20. You go up to two million, it takes 21. The very slow growing function, that logarithm function, so that tells you that this is gonna be a very efficient way of searching a sorted array. A category thing called the divide and conquer that take a problem, divide it typically in half, but sometimes in thirds or some other way, and then kind of – in this case it's particularly handy that we can throw away some part of the problem, so we divide and focus on just one part to solve the problem. All right. So this is the first one that's gonna start to really inspire you for how recursion can help you solve problems that you might have no idea how to approach any other way than using a recursive formulation. So this is an exercise that comes out of the reader in Chapter 4, and the context of it is you have N things, so maybe it's N people in a dorm, 60 people in a dorm, and you would like to choose K of them. Let's K a real number, four – four people to go together to Flicks. So of your 60 dorm mates, how many different ways could you pick a subset of size four that doesn't repeat any of the others? So you can pick the two people from the first floor, one person from the middle floor, one person from the top floor, but then you can kind of shuffle it up. What if you took all the people from the first floor, or these people from that room, and whatnot? You can imagine there's a lot of kind of

machinations that could be present here, and counting them, it's not quite obvious unless you kinda go back to start working on your real math skills how it is that you can write a formula for this. So what I'm gonna give you is a recursive way of thinking about this problem. So I drew a set of the things we're looking at? So there are N things that we're trying to choose K out of. So right now, I've got 12 or so people, or items, whatever it is. What I'm gonna do is I'm gonna imagine just designating one totally at random. So pick Bob. Bob is one of the people in the dorm. I'm gonna kind of separate him from everybody else mentally in my mind, and draw this line, and kinda mark him with a red flag that says that's Bob. So Bob might go to Flicks or might not go to Flicks. Some of the subsets for going to Flicks will include Bob, and some will not. Okay. So what I'd like to think about is how many different subsets can I make that will include Bob, and how many different subsets can I make that don't include Bob. And if I added those together, then that should be the total number of subsets I can make from this collection. Okay, so the subsets that include Bob – so once I've committed to Bob being in the set, and let's say I'm trying to pick four members out of here, then I have Bob and I have to figure out how many ways can I pick three people to accompany Bob to the Flicks. So I'm picking from a slightly smaller population. The population went down by one, and the number I'm seeking went down by one, and that tells me all the ways I can pick three people to go with Bob. The ones that don't include Bob, and Bob's just out of the running, I look at the remaining population which is still one smaller, everybody but Bob, and I look for the ways I can still pick four people from there. So what I have here is trying to compute C of NK is trying to compute C of N minus one K minus one and add it to C of N minus one K. So this is picking friends to accompany Bob. This is picking people without Bob. Add those together, and I will have the total number of ways I can pick K things out of N. So we're very much relying on this recursive idea of if I had the answer – I don't feel smart enough to know the answer directly, but if I could defer it to someone who was actually willing to do more scrutiny into this thing, if you could tell me how many groups of three you can join with Bob, and how many groups of four you can pick without Bob, I can tell you what the total answer is. The simplest possible base case we're gonna work our way down to are when there are just no choices remaining at all. So if you look back at my things that are here, in both cases the population is getting smaller, and in one of the recursive calls, the number of things I'm trying to pick is also getting smaller. So they're both making progress toward kind of shrinking those down to where there's kind of more and more constraints on what I have to choose. For example, on this one as I keep shrinking the population by one and trying to get the same number of people, eventually I'll be trying to pick three people out of three, where I'm trying to pick of K of K remaining. Well, there's only one way to do that which is to take everyone. On this one, I'll eventually keep picking people, so the K is always less than the N in this case. The K will eventually kinda bottom out, or I'll say I've already picked all the people. I've already picked four people. I need to pick zero more. And at that point, that's also very easy, right? Picking zero out of a population, there's only one way to do that. So what we end up with is a very simple base case of if K equals zero – so I'm not trying to choose anymore. I've already committed all the slots. Or if K is equal to N where I've discarded a whole bunch of people, and now I'm down to where I'm facing I've gotta get four, and I've got four left. Well, there's only one way to do those things, and that's to take everybody or to take nobody. And then otherwise, I make those two

recursive calls with Bob, without Bob, add them together to get my whole result. That's wacky. I'm gonna read you something, and then we'll call it a day. I brought a book with me. I stole this from my children.

[End of Audio]

Duration: 48 minutes

ProgrammingAbstractions-Lecture09

Instructor (Julie Zelenski):Hey. Good afternoon. Assignment two coming in today so hopefully you brought some paper copies and did some E-submits and are ready to immediately take on your next assignment. There's no rest for the weary in this class. My understanding is that Miron calls the version where we just chat about some of the assignment the pain pole. I think that's really just much too negative and I'm gonna call it the joy pole instead, and I'm interested in getting a little bit of feedback on how much joy you had in doing your last assignment. First, let's just do the quick counts of time spent.

How many people think they managed to get the whole thing done in less than 10 hours? All right. That's a few of you. That's very good. Very good. Ten to 15? Kind of more of my target range, right, and so that looks like a big healthy chunk in there. Fifteen to 20? Okay. A little bit smaller and then more than 20? Anybody willing to – maybe not. Here's another question, which is how many of you think – so there's two halves, right, the random writer, the maze generating and solving, how many of you think you spent more time or found it more difficult or more time consuming to get the first one done, the random writer than the maze? How many people thought the maze is where you spent your time? Yeah. And of the maze part, how many thought it was harder to generate the maze than to solve the maze? A little bit. How many thought it was harder to solve the maze than generate the maze? Okay. So it looks like solving the maze may be where it comes down to taking the biggest chunk of joy was spent. Okay. Good to know. Hopefully one of the experiences or one of the take home points right after the end of that was certainly that templates provide a lot of leverage. Having these container classes, you can get a lot of things done; write these sophisticated programs with a small amount of code.

I saw a lot of people write who had a comparison function that just did not reliably compare things, all right, for example did not order them. The set really needs to know ordering and who's first, who's last, who's in between because that's how it stores them to later look them up. If your comparison function is not reliable, if you pass A and B and it'll say oh, A is less than B and if you pass B and A, it'll say B is less than A, right, you have a total inconsistency in your system, which the set will then make a hash of, right, it'll throw things in places that it doesn't look later because it took your word at it when it said it should go to this side or that side and it won't find it when it goes looking for it this next time so you do really have to have an ordering function that's reliable that every single time you pass it, it's the same things.

No matter which order they are, they're gonna tell you the same truth about their relative positions. That's one of the lesson learned from that. Assignment 3 is going out today and assignment three is basically a problem set. There are six problems on it. Each of them requires one recursive function and in a couple cases, a little tiny bit of support code around it. Those recursive functions are short. Typically, 10 lines at most. Some of them, six or seven in some of the easier ones. But that code is hard one. Right. You won't find those lines easy to knock out the way you could write out four loop and be done with it, right; there really is some thinking and careful planning and doing the recursive thing and

getting your head around it. So it is a small amount of code, but a very dense and complicated bit of code to generate so hopefully you will start that one early and give it time to gel because sometimes I think you'll see the problem the first time and not see the recursive insight. A day or two later, it will start to make more sense to you, but if you try to compress that all into the night before it's due you may not have enough time to let everything come together well. It is really important to get recursion started now. This is gonna be the recursion problem set.

The next problem we will do is the big venerable boggle that has recursion at the heart of it, plus a lot of other things going on so if you don't spend the time this week getting your recursive footing right, it's gonna start to snowball because recursion shows up from here all the way out through the quarter so this is the time. When you're working on this homework, be sure to get the help you need to make sure the concepts start to come together for you because this is kind of a key to doing well in the rest of the quarter.

So today I'm going, I'm talking about [inaudible] recursion. This is kind of corresponding to reader chapter five. I will do a bunch of examples today that kind of get at this idea with different domains. And then we will go on to recursive backtracking which is the topic from chapter six for Friday. We won't cover the later sections of chapter six. We're mostly gonna focus on the first two sections. There is a later section about game playing and strategy and stuff. It's very interesting and it's worth reading, but we won't formally cover it and use that material in this course right now. Okay. So kind of remind ourselves about thinking recursively and where we're trying to work out way toward.

Recursive [inaudible] is really the hard part. Somebody gives you a problem and they say solve this recursively, where are you gonna find you spend your time is saying, well, given this problem how can I find that self similarity in it, how can I break it apart in such a way that a smaller form of the problem within, right, if I made that recursive call and then used it was gonna solve the whole problem for me. Maybe there's more than one solve problem, maybe it's a divide in half, something like that, but that part is usually where you'll spend a lot of your time is trying to figure out how it is you can structure it to divide it in a self similar way. Once you have that plan about how you're getting a little bit smaller, a little bit simpler, it's usually not so hard to follow that to the logical conclusion and say well, if I keep doing that enough times, what do I get to that will really allow me to terminate this, the simplest possible case that all of those, you know, minimizing of the problem might eventually lead to something so simple we can directly solve it that we're working toward. Sometimes there's more than one base case, but ideally, there's exactly one that everything comes to. It kind of cleans up your code if you can make it come out that way.

There's a couple common patterns and we've seen some of these already and we're gonna see them again and again, but partly what you're trying to develop is a set of examples you've seen before that help you to kind of use those patterns to explore further problems in the future and so a very common way of dividing stuff is you have some collection, whether it's a vector or a string or a set of paths or something like that that

part of the recursion often is that you somehow separate one from them so when we were trying to choose the people to go to the flicks, right, we picked some student at random and said okay, this student is either in or out and so we separated one from the mass, leaving us an N minus 1 and then that N minus 1, right, was that recursive piece about that substructure saying, well, if I had the answer for the remaining N minus 1, right, I could use the information about this 1 I've separated out to join it together. So sometimes, that's the first person, sometimes that's the last person. Sometimes it's almost a random person that you're picking, but the idea of separating one or some small number leaving the N minus that number to work on. Sometimes it's a bigger chunk, it's a division in half or in thirds or something where I'm dividing and conquering the binary search doing that saying. We've got some information from the middle that tells us about what these two halves are in the case of binary search, we only do recurring one.

We'll see some cases we're actually able to divide it in half and recur on both, but where we're still just dividing the problem in easier, in this case, half as big versions of the same problem, but then we can attack recursively. And sometimes there's more like a choice. This is maybe the thematic thing for some of the later things when [inaudible] is that there's some choice of the options that remain and the recursions that attack it to make it one of those choices, which then kind of leads you to a state that has fewer choices to be made, fewer decisions and allows you try decisions from there to work your way to some base case. One of the things I'm also gonna be talking about today is some of the implications of the relationship between when you process this part you've separated out and when you make the recursive call. But it's actually totally not irrelevant whether I make the recursive and then try to process the one, or whether I process it and then do the recursion. In some situations, they'll produce different results and different answers that are interesting to know about how those things play out. So today, the theme is procedure recursion and procedural is differentiated from functional in a very minute almost a bit semantic about the definition, but it's worth telling you.

A functional recursion problem is one where you have a function returning an answer like is [inaudible] true/false. The factorial is a raised power telling you what the answer is. In a procedural recursion, we have functions that don't return anything so there's not an answer being computed in result of that, but there is recursive in [inaudible] so there's a procedure here that calls itself eventually bottoming out at some base case and as a result of all that recursion being done, something happens. The one I'm gonna do at the beginning here is some drawing pictures, some graphics that draw fractal graphics where there's not an answer being computed, but there's something being illustrated on the graphics window. So I'm gonna look at two examples of this because I think sometimes for some students the numbers and the strings aren't the best way to get it; sometimes visually really helps to open up the enlightenment for what recursion means in terms of the structure. We're gonna look at some pictures that have a self similar structure that repeat within itself that you if you look at it at one level of detail you'll see certain elements that, actually, if you look a little closer, you'll see that same element, but just repeated on a smaller scale within that and then within that again kind of moving down to that base case.

The way these things that we've written in code will give us some outer fractal that you draw that is part of its handling then makes recursive calls to draw these inner, smaller versions of the fractal within itself. Okay. So here is a little bit of code. So I'm not gonna tell you what it goes, but we're gonna sort of just imagine thinking it through and looking at this to see what kind of things it could produce. Draw fractal, let's give it an X, Y in width and height, which I'll assume is the bounding rectangle of what's being drawn and then it makes a call to some helper called draw triangle. If I assume that does what it seems like it does it says oh, it draws a triangle, right, that's inscribed within that rectangle. Next up I'm seeing this base case that's, like, well, if that triangle was particularly small, in this case, two tenths of an inch or smaller in either dimension, then we don't do anything further with it. We leave it alone. Otherwise, we do a little bit of computation here, a little bit of math to compute some coordinates and they're labeled the left, top and right, which is basically it's making a call on the left sort of sub region of this rectangle that's at the lower left corner, but half as tall, half as wide. Here's one that starts in the middle and is also half as tall, half as wide that represents a chunk out of the top, and then one that's to the right.

This produced something you've seen before, but you generated it a very different way last time. Right. In assignment one, we had you do the chaos demonstration, it produced this drawing, which is due to Sirpinski from a different mechanism, but in fact, it is a very recursive drawing when you look at it. Right. There's this outer triangle and then within it, there is a left most, a top most and a right most reproduction kind of the outer triangle at a smaller level of scale. If I look in each of those triangles, would I see that same self-similarity? If I just keep going down at each level of scale, I see the same thing, that there's a triangle which itself has these three left, right, top regions that exhibit the same structure the outer one did, but just a little bit smaller. It eventually gets so small that it stops drawing and that's what terminates our recursion. So that is one of the aspects we'll be looking at. Okay. I did the work first. What happens if I turn it around so I have the same base case right and the same order of left, top, right here, but I wait to draw the big triangle until after I've drawn all the other interior fractals so doing the processing for this one after I've done those. I'm gonna get the same picture, but it's gonna grow in a different way that only I've done the whole thing is the outer triangle; in this case, drawn on top which isn't actually very noticeable because the inner piece is kind of already made up the outer boundaries anyway.

Student:

Can you go slower?

Instructor (Julie Zelenski):Yeah, I slowed this one down a little bit. It isn't any slower in real life. I was kind of screwing with the pausing because I wanted to get it just right and then one of the pauses I think the pause is slightly faster than the other. But growing right from that lower left corner up then working all the way through the top and then all the way through the right and then the outer most something only being built is an after effect of once the inners have been constructed, so the very first one appearing way down here. Now, let me change my three recursive calls. It currently goes left, top, right. I'm

gonna change the code to go top, left, right. I'll get the same picture, but where is the very first triangle drawn? Is it small or big? It's small and where does it live within the triangle? The very, very top, right? So the top most point…the top small little triangle, it goes there. What is the second triangle that's drawn? The one that's just down to the left of that one so I should see the top most one, it's left most neighbor and it's right most neighbor and then we'll start working our way down from there. One thing that is sometimes helpful to think about in terms of visualizing what a recursive procedure is doing is to think of it as a kind of an upside down tree where here's my draw fractal to the outer one and then it makes these calls, right to top to left to right.

Realize that the way that the recursion proceeds is that it makes the call to the draw fractal of top, which then makes a call to the draw fractal of its top, which makes the call to the draw fractal of this top so it goes deep, right, until it hits that base case and so this was where I went top, top, top, top, top, top, top, right to the very top most one. Well, after it bottoms out here, it comes back to this one that said, okay, well, I drew the little tiny top, now, I need to draw the left and the right and so the ones that are fleshed out at the bottom here are the left and right of the top most one. And after those, it kind of comes back.

So sometimes thinking about it in terms of a tree and realizing that it is not sort of level by level or [inaudible], it's not like it draws all the outer ones and then the inner side of that, it really goes all the way down to that base case before it start unwinding and backing up and revisiting the pieces that kind of deferred as part of the recursion to come back to. So the very first call to draw fractal is sitting on the stack frame while all these other ones are active and when they finish their work and unwind, it picks up where it left off in drawing the other two thirds of the fractal. So this one should grow down from the top from there and then it should start to go to the left also from the top working its way down and then once it's worked out the whole of the left, do the right also from the top working its way down. So same picture in all these cases, right, am I doing the big triangle first, the inner triangle first, the left, the top, the right, but they show you that kind of the processing and how it would evolve is different. In some cases, this is gonna really matter to us. It's gonna actually change the outcome of the recursion how we're doing this so it's good to have a little bit of an idea how to visualize the different tracing through the calls. Any questions?

Student: Is it possible to draw by joining in three different places at the same time?
Instructor

Well, I mean, [inaudible] control, right, and so at any given point we're doing one thing at a time, so really without more fancy things, the answer is basically no. Let me show you more little recursive fractal. And this is based on a famous Dutch painter Mondrian, right, who was alive in the first half of the 20th Century there, one of the Cubists, the group he belongs to, and you've probably seen some of these paintings. They might look familiar to you with these very bright primary colors, a lot of horizontal and vertical lines bisecting the canvas and then some of those squares being filled in with bright colors. This is sort of one of the more famous of the pieces that he did.

And his little philosophy I quoted from there, right, was this idea that this is about harmony and beauty and you can just through lines and colors produce things that really have an aesthetic component to them. I think it's a pretty neat thing. But I'm not an artist, by any means, so I think the solution to that is if I want to create beautiful things I need to write computer programs that do them for me because if it was up to me, I won't do it with my own hands. So what I'm gonna show you is a technique for thinking of this as a recursive problem. If you look at what a Mondrian is and if you're sitting down to start this, you have a big canvas in front of you. It starts blank and you say, well, where should I begin? Well, let me just pick a random direction, horizontal or vertical, and then I'll pick a random placement for it so I decide I'm gonna bisect vertically and I pick a spot somewhere in the middle of the thing and then I do a bisection of that with my big black line and now I look at the thing that I have left and I've got well two smaller canvases; one to the left of the line, one to the right, which themselves I could imagine using that same strategy for. Well, look at it and decide whether to make a horizontal or a vertical line and then, occasionally, along the way decide to throw in a little color so when I have a canvas maybe wash it in red or blue before I go through my division.

And sometimes, I decide not to divide it at all. So this canvas that I've sectioned off here, I might just decide to leave it there and say, okay, that's good enough. I don't want to go any further on that one. That's not what I wanted to do. So it has these three options, divide the canvas horizontally, divide it vertically, do nothing and then when I have those two smaller canvases, I recursively can apply the same kind of Mondrian style affect to it and when I get to something that's too small, I can stop trying to divide it further. So I'll show you what the code looks like here and then I'll show it running to you. Most of it actually is a little bit gunky just because there's a little math of adding and moving and stuff like that around, but it has this idea of oh, here's the rectangle you're trying to draw a Mondrian in. If it's too small, in this case, anything under an inch I decided was too small. Otherwise, I fill the rectangle in the random color, which often, is white, sometimes yellow, sometimes blue, and then based on three cases of a random choice, I either decide to just leave it as is filled with whatever color I chose or I make a line, vertically in this case or horizontally in that case, and then recursively section off the two portions I've made to recursively apply a drawn Mondrian to draw some more Mondrian in those pieces.

Does it work? Any by work, the code works, but does it actually produce art? You can be the judge. Now, the good thing is see it's easy to make new ones if you don't like that one. I don't like that one either. A little too much blue. That's getting somewhere. I might put that on my wall and then at this point, I just started making them just as many as you want and just keep going. Some of them it stops early, it has nothing to do. My mother-in-law is an artist and I feel like I have to apologize to her whenever I do this because I'm really not trying to make fun of art as though actually a computer can do anything. The truth is, when you do this actually what you realize is that it's not art, right, what's truly aesthetic is something that actually is not generated by randomness and recursion, but it's still interesting how [inaudible] have that pattern to them that are sort of Mondrianesque just by simply taking this recursive strategy and saying that's a little Mondrian, this is a little Mondrian and if you assemble them all together, you've built a big Mondrian.

Millions of dollars right here I'm generating. I tell you, when I retire from teaching, that's gonna be my next career, turning out fake Mondrian's. All right. So any questions about little pictures and graphics and things?

Student: On the previous one, could you go over what's passed in each of recursive or draw a Mondrian?

Instructor (Julie Zelenski):Yeah. So the only thing it takes is it's just a rectangle. It says, this rectangle you should fill with Mondrian like things, so the outer rectangle is the whole canvas and then once I've done a bisection, right, I've picked some point in the middle, I've divided the rectangle into two smaller rectangles, the left and the right. It's not necessarily exactly, but I have one that starts in the origin and goes to the midpoint, one that starts in the midpoint and goes to the right side and so it's just basically taking the outer rectangle and dividing it. If I drew a picture here, it might help. You've got this thing.

You choose to bisect this way and so what you're passing, is this rectangle and that rectangle, and when this one decides to bisect this way, then it'll pass this rectangle and that rectangle and so at each stage you'll have these smaller rectangles that were pieces of the outer rectangle that you're continuing to draw Mondrian into and eventually they get so small that you say I won't further divide that up anymore. I'll just stop right here. And so the line actually is drawn on the way down it turns out. If you'll notice, the draw black line happens before it draws the thing, so draw the line and then draw a background on top around it and stuff. Either way it doesn't actually really matter. You could draw it on the way out and you'd end up with the same picture. If just would appear differently.

Student: Explain why you haven't returned something, like, a break or some other leads?

Instructor (Julie Zelenski):Well, in general, I just need something that says don't go through the rest of this code. I'm not trying to get out of a loop or a switch statement, right, so if I really want to leave the function, return is the quickest way to do that. The alternative is I could've inverted the sense of this task and said, well, if it's at least an inch wide and an inch tall, then get into this code and effectively it would say if, and it didn't pass if you didn't put a drop in the bottom and falling off, but either way, right. So I think I probably tend to prefer to do it this way just because I like to get my base case up from the front of my eyes seeing what it does and, typically, the right thing for the base case to do is identify that you're at the base case, do whatever simple processing is required in that case and then return the answer or just return right away saying, I've handled the base case and now let the rest of the code be and now in the recursive case going on.

This is very much the style I tend to use. If we're at the base case, do the base case stuff and return. Now I'm gonna hit you with the most classic recursion example known to all cs students worldwide. I wouldn't be doing my duty if I didn't teach you a little bit about the legend of Brahmin and its towers and how relevant it is in today's world that you know how to move graduated discs around. All right. So I'll give you the set up.

Apparently, there are these monks and there's a legend of the tower Brahmin that these monks, apparently, big troublemakers, monks, you know how they are, you gotta keep them out of trouble so the dyad here had this idea. Well, I'm gonna give them this task that's gonna keep them busy for a long time and that way they won't be causing problems. So what he did was he stacked up these three spindles, A, B, and C, big poles and he had these big heavy discs, they were made of solid gold apparently in the Brahmin religion and they are graduated so the big disc at the bottom and a slightly smaller one on top and then so on, and they're all stacked up let's say on tower A. I've labeled them, A, B and C just to keep track of which is which, and the dyad says yeah, I got all these discs all stacked up on this in A, but it turns out I'd really much rather have them on tower B.

Sorry. What can you do? I accidently put them there and I'm so strong and big and heavy I could move them, you on the other hand, right, puny little human beings that you are, are restricted to being able to only move one disc at a time because they are so inordinately heavy that it takes everybody's strength to move just even the smallest of the discs from one place to another. And he further in states these rules that the discs all have to live on a spindle so you can't actually just pick them up and start throwing them around in the desert. You're trying to move this whole tower from A to B. You can move one at a time. And in addition, because they're so heavy, if you were ever to put a small disc on a spindle and put a larger disc on top it would immediately crush the one underneath it. So that's disallowed as well. So you always have to keep an ordering of large ones on the bottom to smaller ones on the top. All right. That's the deal. That's the set up. Those trouble-making monks. Apparently, there's 64 of these things. All right. In computer science, I don't have any solid gold heavy discs, but I do have some very brightly colored plastic discs. So this is not so much the towers of Brahmin or the towers of [inaudible], I changed locations somewhere along the point of history; this is the towers of Fischer Price.

I did not actually steal these from my children, but they have one at home, which I keep waiting to see them show evidence of their recursive nature and I must say very, very disappointing. Okay. So I've got a tower, in this case, of six. This is A, this is B, this is C and I want to get this from here to here and I don't want to violate all those rules. I can't just start throwing the discs around and making a mess. I have to move it from here to here. I don't have a lot of choices here, but once it's there, for example, this yellow guy can't go on top of it because it would squash it, right? So I could kind of move yellow in there, so there's something a little bit about this idea that somehow there's gonna have to be some back and forth that is a part of what's going on. So let's go back and look at this for a second. So as I said, many of your classic recursive patterns involve sort of looking at the problem you have at hand and trying to decide is there some way you could separate it a little bit and one of the more likely ways to do it is say is there some way I can move one disc out of the way leaving N minus 1 that could be worked on recursively? Okay.

So it's probably not likely that I'm gonna be able to get, let's say the blue disc out of the middle very easily, so it seems like probably the two most likely candidates for that are either the top most disc somehow gets moved out of the way to uncover the bottom part

or the bottomless one, somehow I get this other tower out of the way. So let's start by thinking about the top one because that's certainly the easiest. I can get this top guy, as I said, and stick it somewhere. Well, let's say I put it over here in the temporary one. Okay. I get it out of the way and now I have the tower of N minus 1 left. Okay. That seems like it's kind of getting me somewhere and now I need to move that tower, N minus 1, over here, but the situation I made is I've actually made the problem, not easier in the situation, it's a little bit smaller but it's actually further complicated by the fact that given this small disc is over here occupying the small spindle means that, suddenly, this spindle is out of commission. So the purpose of moving this spindle might as not exact, right, because this guy is blocking anybody from getting there. So it seems like I have a problem that looks what I started with, but it's not actually any easier, in fact, it actually got a little bit harder. So let me back up from that.

Let me just let that be a dead end for now. What about this purple disc that is down here on the bottom? I'd like to get to the purple disc. Well, I can't just pick up the whole tower and move it, or can I? Well, let's think. I've got this tower of N minus 1 on top of the purple tower. If I were able to produce a delegate, a clone, a co-worker, right, who I could say, Hey, by the way, you know how to move towers, could you please move the N minus 1 discs that are on top of the purple one over to the temporary spindle using this other one as the spare? Well, that's kind of interesting. So if I could move that N minus 1 out of my way and over to that temporary spindle, then it would uncover this bottomless one which then is one step away from where it wants to be; it wants to move to the middle. So this is really where recursions are kind of buying us a lot of the heavy lifting. I want to get this tower from here to there. There's all these rules. It's very complicated. I can't think about it, it makes my head hurt, but if I just believed that I was in the midst of solving this problem that I haven't yet solved and if I had that solution and I believed and I had faith, right, and I believe that it will work, I move these guys out of the way, I move this over here, I move it back.

So let's see how much I can do without blowing it. All done. Don't ask to see it again. I'll show you a little code. I'll show you the code. Tiny, tiny little piece of code that solves the whole problem. I've got a [inaudible] coming in here, which is the height of the tower to move and I'm identifying the source destination attempt, the three spindles that are currently in play here with these letters, characters; if there's something to move – so I'm gonna talk about the base case in a second, but in the general case where I've got five things to move it says move the four discs that are on the source. So the source, let's say is A in this case, and now instead of moving them to the destination, move them aside to the temporary spindle using the destination as the temporary. Once you've gotten those out of the way, you move that single disc, the bottom of the disc from the source of the destination, and then you go pick up N minus 1 tower you left on temporary and stack it on top of you in the destination now using the source as your temporary. At any given stage, you've got three to go and then you have to move the tower of height off of you.

When you've got two to go, you have to move the tower of height of 1 off you. Base case you could imagine being pretty easy while single height tower, just one disc, could be exactly moved, this actually prefers going to even a simpler case, right, which is if you're

asking to move a zero height tower there's nothing to do. In fact, the base case for the zero is if N equals zero, do nothing for any positive number then it goes through the process of shuffling the ones off of you to uncover your bottomless disc and then moving them back on. So five little lines and all the power in the world. Question?

Student: Why use the destination as a temporary?

Instructor (Julie Zelenski):The idea is that if we're moving here from A to B, right, I need to leave this one open, right, so that I'll be able to move the purple one when the time comes. So I'm trying to move this out of the way, right, so here's where it's trying to go and in the process, right, you have these two places you can kind of shuffle things back and forth, where they came from, where they're not going and where they are going, so there's also the source of destination and it's pretty obvious; and then what happens is that whatever is left over is the one that gets used as a temp.

Student: Why can't you put those on the second one?

Instructor (Julie Zelenski):Well, if I put them on the second one, how am I gonna get the purple one underneath them?

Student: Well, you wanted to put them on the last one.

Instructor (Julie Zelenski):Well, my plan was to move from A to B actually. It doesn't really matter.

Student: Oh, okay.

Instructor (Julie Zelenski):I'm trying to move from A to B so in fact, C is the place where I'm just leaving stuff temporarily.

Student: Okay.

Instructor (Julie Zelenski):So you're gonna see that the stack frames here show how deep the recursion ever gets at any given point, right, at this case, having four discs it gets about four deep, actually five because it goes to the zero case and that you'll see it kind of go down and back as it shows it kind of moving the tower away and then getting back to moving the bottomless disc and then doing another deep recursion to move that tower back on top. And so at any given stage, right, they're all stacked up and the start/finish attempts start switching positions depending on what level of the recursion we're at. Each time we get to zero we hit our base case and unwind. Let me make it go a little bit faster because it's a little bit slow.

Student:[Inaudible]

Instructor (Julie Zelenski):[Inaudible]. All right. So I just moved the tower of height three all the way away and now let's move the bottomless one back and now it's gonna

start moving the tower that was shuffled away back on to it and then it will eventually stack it back up on the B where it wants to go. I can make him move the tower.

Student: Sixty-four.

Instructor (Julie Zelenski):Yeah. Sixty-four. We'll talk a little bit about why I wouldn't want to put in the number 64. You can see I'm kind of it around. Dancing, dancing, much better than I would do. I actually maintained the rules at all times. No discs getting smashed. I go up to six and so never getting lost, always keeping track of what it's up to, right, it's a very tricky thing for a human to do, but a very easy thing for a computer to be able to identify what part of the recursion we're at and what we're trying to do right now without getting confused about the other things that are also kind of ongoing.

They're all very neatly kept apart. So when people will ask you about recursion, everyone will want to know have you been exposed to the towers problem, and now, you have. So very powerful little piece of code, right, and it feels very much like a trick. I can remember sitting in a room, not unlike yourselves, a gazillion years ago and having someone explain the towers of Hanoi to me and just not believing. Just saying that can't work. You didn't do anything. All you did was call yourself before you were even done and that's just nonsense and that's just not right. And then after some amount of thinking about it and working it through, I did conclude that it actually did work, but at first, it really did seem like a trick. So if you're feeling that way, this is par for the course. It is a little wacky to get your head around, but it is sort of relying on this idea of the mathematical induction; if you can do it for the smaller version of the problem and then build on that to solve the bigger problem, as long as you make that logical progression from this problem to the smaller one down to that simple case, you will eventually solve the whole thing.

Student:[Inaudible]

Instructor (Julie Zelenski):Source destination in this case. Okay. So then there are two problems I want to do. I'll probably only do one today and I'll do one on Friday that are what I think of as the mother problems of all recursions that many, many problems in the end just boil down to an instance of either the permutations problem or the subset problem. So I'm gonna show you permutations and subsets sort of in gory detail and then I'll try to build with you about how many things actually are just a permutations problem or subset problem when looked at the right way and so once you kind of have these two in your arsenal, you're very much prepared to attack a lot of different recursion things using those same patterns. So the one I'm looking at here is permutations. You have an input string, let's say it's the alphabet A through Z, 26 letters, what you'd like to do is enumerate or print or list all of the ways you could permute the alphabet. So there's 26 letters in the alphabet, if you know anything about combinations and permutations, you realize there's 26 ways you can choose the first letters, A, B, C, all the way to Z and then that leaves you with 25 letters from which to pick the next one, right, so there's 25 ways to do that and then 24 ways to pick after that and so at each step you have one fewer

choices remaining to conclude the permutation; and then the number of permutations is then factorials in the length of the input, so 26 factorial, which is an enormous number.

It tells you about all the different ways you could rearrange the alphabet. For a smaller string, you know, A, B, C, right, there are three factorials for that, six different ways you could permute that. The same principle applies no matter how large the string is. So our goal is to write a function that, given an input string, will print all of the permutations of that string. That's the goal. Okay. So I've got A, B, C, D coming in, I want to be able to print D, C, B, A and C, A B, D and all these other variations. I'm gonna use the strategy that actually I described just the way I did kind of intuitively about how permutations are constructed. That at any given point, right, I have a choice to make, and that choice in this case, will be what's the next letter to attach to the permutations. So if I measure my goal by trying to build the permutation up one letter at a time. I start with an empty string so I'm gonna have the input and the output. The input is the letters I haven't yet used, so in the case of the string, A, B, C, it might be the whole string, A, B, C. I've got this output, which is what letters I've chosen so far. It starts empty. I look at my A, B, C and I say, Well, each of those letters could be the next one that's here, why don't I go ahead and pick one, pick A for that matter, put it on the output and then recursively call myself to say well given A is in the front, because you also print all the things that you can permute the remaining letters, B, C, behind it.

After the magic recursion has done what it's gonna do I can come back to the call I was at and say okay, well, now it's not just A in the front I need to try; I also need to try B in the front. So I tried B in the front and now permute the A, C, that I leave behind. I try C in the front and permute the A, B, that's left behind. So at any given stage of this recursion, right, I have these options, which are of all the letters remaining in the input, each of them needs to be tried as the next one to go and then I need to recursively permute on this slightly smaller form of the same problem where the output is one shorter of. I've removed one letter from consideration because I've picked it and the input is one longer. So at each level of the recursion, one letter is shuffled from the output to the input after I've done that N times, right, I will have created a permutation and then I can just print it. So I've talked about this. Let me just show you the code because I think that is where we can spend the most time illuminating what's trying to go on. So here's recursive permute. The form that I'm gonna use here is gonna take two arguments. I'll explain the need for it in a second, but largely what I'm gonna be tracking is that input and output.

What I have assembled so far is in that first parameter, so the things I have committed in the permutation I'm building right now, the rest is those letters that haven't been chosen that have been passed over so far that are remaining to be permuted and attached on to so far. If the rest is empty, that means everything has been attached in so far, I have a permutation and then I just print it. If it's not empty, then for every letter that remains in the rest – so imagine I'm starting with the whole alphabet and so far was empty, this four loop is going to iterate 26 times, it's going to take out the [inaudible] character and it's gonna append it to the permutation and then it's gonna remove it from the rest to show that it's been used so we won't accidently repeat it later in the permutation.

And then we make this recursive call saying okay, now having picked the letter N and put it in the front, here's the whole alphabet without N. So at the beginning we have everything in the input, nothing in the output. Well, once we pick something that shuffles on character from here to there. A subsequent call down here moves another character and then eventually we've emptied out the rest. There are other alternatives though, for example, up here it could be that we picked B and left behind A, C, it could be that we picked C and left behind A, B. Before it's all done, we're gonna have to try all of those things and so the very first level of the recursion is gonna make a number of calls equal to the length of the input. Each subsequent level, right, in here makes one fewer call at each branch. So if I had a 26 at the top, each of those 26 then makes a 25 underneath so there's 26 25s branching down there and then each of the 25 makes a 24. So this is an enormously wide tree. The depth is bounded by the total number of characters, but very, very wide. I'm just gonna show you a little part of the tree to get an idea. If I'm permuting A, B, C, D in the so far for the input, [inaudible] starts empty and there are four calls that are being made, first with A, then with B, then with C, then with D and in each case, removing that letter from what remains and then exploring.

So if I kind of further expand this call, it makes three calls, right, one having pulling B to attach on, one having attached C on, and one having attached D on leaving the remaining two characters. And then underneath that, picking that third character and then eventually pocking the fourth character which there's no choices there. So this is only a partial part of the tree, but it gives you an idea of what the shape of that recursion looks like. This is a tricky thing to get your head around. But it is a critical pattern to get in your arsenals to you know how to apply it. So what this is sort of that choice pattern, right; of the choices I have, I need to go through the process of making those choices, updating my state to show that choice has been made and then recursing from there – making that recursive call to further explore whatever choices remain having made this one, which caused there to be one fewer choices to be made. And in a permutation, you have to make 26 choices, right, for the alphabet. Well, you make one and then you have 25 to go, you make one, you have 24 to go. Each of those calls, right, is working its way down.

There's one little detail I'll mention and then we'll finish on Friday, which is that probably the way this function would be presented to a client or a user who wanted to get the permutations is it makes more sense for them to say here's the string I have, could you please just list the permutations. This notion that somehow I need to track what I've generated so far is very much an internal housekeeping part and so it effects permutations to look like this and then just immediately turn around and make a call to the real recursive function that set up the state, the housekeeping that was being tracked through it, so we'll call this thing a wrapper function. All it is one line of code that just sets up the right state to get the recursion going and kind of state managed. We will see that quite a lot in a lot of codes. They just know about it. Come Friday, we'll talk about subsets and then we'll start looking at recursive backtracking.

[End of Audio]

Duration: 52 minutes

ProgrammingAbstractions-Lecture10

Instructor (Julie Zelenski):Hey there. Welcome. Welcome. Welcome. My computer's not quite talking to anything yet here. You guys got an answer for that? Unplug and replug that? Okay, I don't know what's – whether somebody can help make my computer talk to the projector? Yeah. It's plugged in like ordinary. Yeah, I – we do this every day. There we go. Thank you very much. All right, back to your regularly scheduled lecture.

Today I'm gonna keep talking a little bit more about another procedural recursion example and then go on to talk about, kind of, the approach for recursive backtracking, which is taking those procedural recursion examples and kind of twisting them around and doing new things with them. This corresponds with to the work that's in chapter six of the reader.

And then from here we're actually gonna do a little detour to pick up an explanation of recursive data in the form of linked lists, and that's gonna give us a chance to start talking about C++ pointers, which I know everyone has been eagerly awaiting a chance to get a little bit of mystery of the pointers revealed to them. And so the coverage for that is in chapter 2 in the early sections, 2-2 and 2-3. I [inaudible] covered in the text in sort of 9-5. There's actually a handout I'll give out on Monday's lecture that helps to go along with what we're doing there so there's a little bit of supplemental material to kind of work through that. And so it'll be our first chance to see points [inaudible] we can really use them to do something, and so I kind of differed talking about them until now because we don't have a good use until we get there, and that we'll be exploring a recursive data type called the link list. I'll be hanging out at the Turbine Café after class, so if, actually, you're available this afternoon and want to come and hang out, we'll be there for an hour or two, so if you want to walk over with us or come later when you're free, that would be great. If not this Friday, hopefully some other one. How many people started the recursion problem set? Got right on it. How many people have already solved a problem or two? Oh, yeah. How many people solved all of them? Okay. That's good though. And how is it going so far? Thumbs up? Lots of joy? Joy in Mudville. All right. So as I say, the handout, you know, there's six problems, each of them is actually very, very short, and that may lull you into believing that, well, you can start at it right before because it's not gonna be typing speed that's gonna keep you there all night, but it is actually some really dense, complex code to kind of get your head around and think about. And I think, actually, it helps a lot to have started at it, and thought about it, and let it gel for a little time before you have to kind of really work it all the way out. So I do encourage you to get a jump on that as soon as you've got some time. Okay. This is the last problem I did at the end of Wednesday's lecture. It's a very, very important problem, so I don't actually want to move away from it until I feel that I'm getting some signs from you guys that we're getting some understanding of this because it turns out it forms the basis of a lot of other solutions end up just becoming permutations, you know, at the core of it. So it's a pretty useful pattern to have [inaudible] list the permutations of a string. So you've got the string, you know, A B C D, and it's gonna print all the A C D A B C D A variations that you can get by rearranging all the letters. It's a deceptively short piece of code that does what's being done here – is that it breaks it down in terms of there being the

permutation that's assembled so far, and then the remaining characters that haven't yet been looked at, and so at each recursive call it's gonna shuffle one character from the remainder onto the thing that's been chosen so far. So chose the next character in the permutation, and eventually, when the rest is empty, then all of the characters have been moved over, and have been laid down in permutation, then what I have is a full permutation of the length end. In the case where rest is not empty, I've still got some choices to make. The number of choices is exactly equal to the number of characters I have left in rest, and for each of the remaining characters at this point, maybe there's just C and D left, then I'm gonna try each of them as the next character in the permutation, and then permute what remains after having made that choice. And so this is [inaudible] operations here of attaching that [inaudible] into the existing so far, and then subtracting it out of the rest to set up the arguments for that recursive call. A little bit at the end we talked about this idea of a wrapper function where the interface to list permutations from a client point of view probably just wants to be, here's a string to permute the fact that [inaudible] keep this housekeeping of what I've built so far is really any internal management issue, and isn't what I want to expose as part of the interface, so we in turn have just a really simple one line translation that the outer call just sets up and makes the call to the real recursive function setting up the right state for the housekeeping in this case, which is the permutation we've assembled at this point. So I'd like to ask you a few questions about this to see if we're kind of seeing the same things. Can someone tell me what is the first permutation that's printed by this if I put in the string A B C D? Anyone want to help me?

Student:

A B C D.

Instructor (Julie Zelenski):A B C D. So the exact string I gave it, right, is the one that it picks, right, the very first time, and that's because, if you look at the way the for loop is structured, the idea is that it's gonna make a choice and it's gonna move through the recursion. Well, the choice it always makes is to take the next character of rest and attach it to the permutation as its first time through the for loop. So the first time through the very first for loop it's got A B C D to chose, it chooses A, and now it recurs on A in the permutation and B C D to go.

Well, the next call does exactly the same thing, chooses the first of what remains, which is B, attaches it to the permutation and keeps going. So the kind of deep arm of the recursion making that first choice and working its way down to the base case, right, right remember the recursion always goes deep. It goes down to the end of the – and bottoms on the recursion before it comes back and revisits any previous decision, will go A B C D, and then eventually come back and start trying other things, but that very first one, right, is all just the sequence there.

The next one that's after it, right, is A B D C, which involved unmaking those last couple decisions. If I look at the tree that I have here that may help to sort of identify it. That permute of emptying A B C, the first choice it makes is go with A, and then the first

choice it makes here is go with B, and so on, so that the first thing that we can see printed our here will be A B C D, we get to the base case.

Once it's tried this, it'll come back to this one, and on this one it'll say, well, try the other characters. Well, there's no other characters. In fact, this one also finishes. It'll get back to here as it unwinds the stack the way the stack gets back to where it was previously in these recursive calls. And now it says, okay, and now try the ones that have, on the second iteration of that for loop, D in the front, and so it ends up putting A B D together, leaving C, and then getting A B D C.

And then, so on kind of over here, the other variations, it will print all of the ones that have A in the front. There are eight of them. I believe three, two – no, six. Six of them that have A in the front, and we'll do all of those. And only after having done all of working out that whole part, which involves kind of the [inaudible] of this whole arm, will eventually unwind back to the initial permute call, and then say, okay, now try again. Go with B in the front and work your way down from reading the A C D behind it. So we'll see all the As then the bunch that lead with B, then the bunch that leads with C, and then finally the bunch that leads with D.

It doesn't turn out – matter, actually, what order it visits them in because, in fact, all we cared about was seeing all of them. But part of what I'm trying to get with you is the sense of being able to look at that recursive code, and use your mind to kind of trace its activity and think about the progress that is made through those recursive calls. Just let me look at that code just for a second more and see if there's any else that I want to highlight for you.

I think that's about what it's gonna do. So if you don't understand this cost, if there's something about it that confuses you, now would be an excellent time for you to ask a question that I could help kind of get your understanding made more clear. So when you look at this code, you feel like you believe it works? You understand it?

Student:[Inaudible]. I have a question. Is there a simple change you can make to this code so that it does combinations [inaudible]?

Instructor (Julie Zelenski):Does combinations – you mean, like, will skip letters?

Student: Right.

Instructor (Julie Zelenski):Yes. It turns out we're gonna make that change in not five minutes. In effect, what you would do – and there's a pretty simple change with this form. I'm gonna show you a slightly different way of doing it, but one way of doing it would be to say, well, give me the letter, don't attach it to next right? So right now, the choices are pick one of the letters that remain and then attach it to the permutation. Another option you could do was pick a letter and just discard it, right? And so, in which case, I wouldn't add it into so far. I would still subtract it from here, and I'd make another recursive call saying, now how about the permutations that don't involve A at all? And I

would just drop it from the thing and I would recurs on just the B C D part. So a pretty simple change right here. I'm gonna show you there's a slightly different way to formulate the same thing in a little bit, but – anybody want to own up to having something that confuses them?

Student:

[Inaudible] ask how you, like, what you would set up to test it?

Instructor (Julie Zelenski):So [inaudible] one of the, you know, the easiest thing to do here, and I have the code kind of actually sitting over here just in case, right, hoping you would ask because right now I just have the most barebones sort of testing. It's like, yeah, what if I just, you know, throw some strings at it and see what happens, right? And so the easiest strings to throw at it would be things like, well what happens if I give it the empty string, right? You know, so it takes some really simple cases to start because you want to see, well what happens when, you know, you give it an empty input. Is it gonna blow up?

And it's, like, the empty input, right, immediately realizes there's no choices and it says, well, look, there's nothing to print other than that empty string. What if I give it a single character string? Right? I get that string back. I'm like, okay, gives me a little bit of inspiration that it made one recursive call, right, and then hit the base case on the subsequent one. Now I start doing better ones.

A and B, and I'm seeing A B B A. Okay, so I feel like, you know, I got this, so if I put a C in the front, and I put the B A behind, then hopefully what I believe is that it's gonna try, you know, C in the front, and then it's gonna permute A and B behind it, and then similarly on down. So some simple cases that I can see verifying that it does produce, kind of, the input string as itself, and then it does the back end rearrangement, leaving this in the front, and then it makes the next choice for what can go in the front, and then the back end rearrangement. And kind of seeing the pattern that matches my belief about how the code works, right, helps me to feel good. What happens if I put in something that has double letters in it? So I put A P P L E in there. And all the way back at the beginning, right, I can plot more of them, right, that grows quite quickly right as we add more and more letters, right? Seeing the apple, appel, and stuff like that. There's a point where it starts picking the P to go in the front, and the code as it's written, right, doesn't actually make it a combination for this P and this P really being different. So it goes through a whole sequence of pull the second character to the front, and then permute the remaining four. And then it says, and now pull the third character to the front and permute the remaining four, which turns out to be exactly the same thing. So there should be this whole sequence in the middle, right, of the same Ps repeated twice because we haven't gone over a way to do anything about that, right? So – but it is reassuring to know that it did somehow didn't get, you know, confused by the idea of there being two double letters. [Inaudible] if I do this – if I just say A B A, right? Something a little bit smaller to look at, right? A in the front. A in the front goes permuted, B in the front, and then it ends up permuting these two ways that ends up being exactly the same, and then I get a duplicate of those in the front. So right now, it doesn't know that there's anything to

worry about in terms of duplicates? What's a really easy way to get rid of duplicates? Don't think recursively, think – use a set. Right? I could just stuff them in a set, right? And then if it comes across the same one again it's like, yeah, whatever. So it would still do the extra work of finding those down, but would actually, like, I could print the set at the end having coalesced any of the duplicates. To actually change it in the code, it's actually not that hard either. The idea here is that for all of the characters that remain, right, and sometimes what I want to choose from is of the remaining characters uniqued, right? Pick one to move to the front. So if there's three more Ps that remain in rest, I don't need to try each of those Ps individually in the front, right? They'll produce the same result – pulling one and leaving the other two behind is completely irrelevant. So an easy way to actually stop it from even generating them, is when looking at the character here is to make sure that it is – that it doesn't duplicate anything else in the rest. And so probably the easiest way to do that is to either pick the first of the Ps or the last of the Ps, right, and to recur on and ignore the other ones. So, for example, if right here I did a find on the rest after I – do you see any more of these? And if you do, then skip it and just let those go. Or, you know, do the first. Don't do the remaining ones. You can either look to the left of look to the right and see if you see any more, and if so, skip the, you know, early or later ones depending on what your strategy is. Any questions about permute here?

Student:

[Inaudible].

Instructor (Julie Zelenski):So the [inaudible] can be just – look down here, right, it just looks like that, right? This is a very common thing. You'll see this again, and again, and it tends to be that the outer call, right, it tends to have this sort of this, you know, solve this problem, and then it turns out during the recursive calls you need to maintain some state about where you are in the problem. You know, like, how many letters you moved, or what permutation you've built so far, or where you are in the maze, or whatever it is you're trying to solve.

And so typically that wrapper call is just gonna be setting up the initial call in a way that has the initial form of that state present that the client to the function didn't need to know about. It's just our own housekeeping that we're setting up. Seems almost kind of silly to write down the function that has just line that just turns it into, basically, it's just exchanging – setting up the other parameters.

I am going to show you the other kind of master patter, and then we're gonna go on to kind of use them to solve other problems. This is the one that was already alluded to, this idea of a combinations. Instead of actually producing all of the four character strings that involve rearrangements of A B C D, what if I were to [inaudible] in kind of the subgroups, or the way I could chose certain characters out of that initial input, and potentially exclude some, potentially include some?

So if I have A B C, then the subsets of the subgroups of that would be the single characters A B and C. The empty string A B and C itself the full one, and then the combinations of two, A B, A C, and B C. Now, in this case we're gonna say that order doesn't matter. We're not – whereas permutations was all about order, I'm gonna use – I'm gonna structure this one where I don't care. If it's A B or B A I'm gonna consider it the same subset. So I'm just interested in inclusion. Is A in or out? Is B in or out? Is C in or out?

And so the recursive strategy we're gonna take is exactly what I have just kind of alluded to in my English description there, is that I've got an input, A B C. Each of those elements has either the opportunity of being in the subset or not. And I need to make that decision for everyone – every single element, and then I need to kind of explore all the possible combinations of that, right? When A is in, what if B is out? When A is in, what if B is in? So the recursion that I'm gonna use here is that at each step of the way, I'm gonna separate one element from the input, and probably the easiest way to do that is just to kind of take the front most element off the input and sort of separate it into this character by itself and then the remainder.

Similarly, the way I did with permute, and then given that element I have earmarked here, I can try putting it in the current subset or not. I need to try both, so I'm gonna make two recursive calls here, one recursive call where I've added it in, one recursive call where I haven't added it in. In both cases, right, I will have removed it from the rest, so what's being chosen from to complete that subset always is a little bit smaller, a little bit easier. So this is very, very much like the problem I described as the chose problem. Trying to choose four people from a dorm to go to flicks was picking Bob and then – [inaudible] to Bob and not Bob, and then saying, well, Bob could go, in which I need to pick three people to go with him, or Bob could not go and I need to pick four people to go without Bob.

This is very much that same pattern. Two recursive calls. Identify one in or out. The base case becomes when there's nothing left to check out. So I start with the input A B C – A B C D let's say. I look at that first element – I just need to pick one. Might as well pick the front one, it's the easy one to get to. I add it to the subset, remove it from the input, and make a recursive call. When that recursion completely terminates, I get back to this call, and I do the same thing again.

Remaining input is B C D again but now the subset that I've been building doesn't include. So inclusion exclusion are the two things that I'm trying. So the subset problem, right, very similar in structure to the way that I set up permutations, right, as I'm gonna keep track of two strings as I'm working my way down the remainder in the rest, right, things that I haven't yet explored so far is what characters I've chosen to place into the subset that I'm building. If I get to the end where there's nothing left in the rest, so there's no more choices to make, then what I have in the subset is what I have in the subset, and I go ahead and print it.

In the case that there's still something to look at I make these two calls, one where I've appended it in, where I haven't, and then both cases, right, where I have subtracted it off of the rest by using the subster to truncate that front character off. So the way that permute was making calls, right, was in a loop, and so sometimes it's a little bit misleading. You look at it and you think there's only one recursive call, but in fact it's in inside a loop, and so it's making, potentially, end recursive calls where end is the length of the input. It gets a little bit shorter each time through but there's always, you know, however many characters are in rest is how many recursive calls it makes. The subsets code actually makes exactly two recursive calls at any given stage, in or out, and then recurs on that and what is one remaining. It also needs a wrapper for the same exact reason that permutations did. It's just that we are trying to list the subsets of a particular string, in fact, there's some housekeeping that's going along with it. We're just trying the subset so far is something we don't necessarily want to put into the public interface in the way that somebody would have to know what to pass to that to get the right thing. Anybody have a question about this? So given the way this code is written, what is the first subset that is printed if I give it the input A B C D? A B C D. Just like it was with permute, right? Everything went in. What is the second one printed after that? A B C, right? So I'm gonna look at my little diagram with you to help kind of trace this process of the recursive call, right, is that the leftmost arm is the inclusion arm, the right arm is the exclusion arm. At every level of the tree, right, we're looking at the next character of the rest and deciding whether to go in or out, the first call it makes is always in, so at the beginning it says, I'm choosing about A. Is A in? Sure. And then it gets back to the level of recursion. It says, okay, look at the first thing of rest, that's B. Is B in? Sure. Goes down the right arm, right? Is C in? Sure. Is D in? Sure. It gets to here and now we have nothing left, so we have [inaudible] A B C D. With the rest being empty, we go ahead and print it [inaudible] there's no more choices to make. We've looked at every letter and decided whether it was in or out. It will come back to here, and I'll say, okay, I've finished all the things I can make with D in, how about we try it with D out. And so then D out gives us just the A B C. After it's done everything it can do with A B C in, it comes back to here and says, okay, well now do this arm, and this will drop C off, and then try D in, D out. And so it will go from the biggest sets to the smaller sets, not quite monotonically though. The very last set printed will be the empty set, and that will be the one where it was excluded all the way down, which after it kind of tried all these combinations of in out, in out, in out, it eventually got to the out, out, out, out, out, out, out case which will give me the empty set on that arm. Again, if I reverse the calls, right, I'd still see all the same subsets in the end. They just come out in a different order. But it is worthwhile to model the recursion in your own head to have this idea of understanding about how it goes deep, right? That it always makes that recursive call and has to work its way all the way to the base case and terminate that recursion before it will unfold and revisit the second call that was made, and then fully explore where it goes before it completes that whole sequence. Anybody want to ask me a question about these guys? These are just really, really important pieces of code, and so I'm trying to make sure that I don't move past something that still feels a little bit mystical or confusing to you because everything I want to do for the rest of today builds on this. So now if there's something about either of these pieces of code that would help you – yeah.

Student:

What would be the simplest way to do that?

Instructor (Julie Zelenski):So probably the simplest way, like, if you said, oh, I just really want it to be in alphabetical order, would be to put them in a set, right, and then have the set be the order for you.

Student: So you said order doesn't matter?

Instructor (Julie Zelenski):Oh, you did care about how they got ordered.

Student: So, like, let's say you didn't want B C D to equal C D B.

Instructor (Julie Zelenski):So in that case, right, you would be more likely to kind of add a permutation step into it, right? So if B C D was not the same thing as A B C, right, it would be, like, well I'm gonna chose – so in this case, right, it always – well the subsets will always be printed as kind of a subsequence. So – and let's say if the input was the alphabet, just easy way to describe it, all of the subsets I'm choosing will always be in alphabetical order because I'm always choosing A in or not, B in or not.

If I really wanted B Z to be distinct from Z B, then really what I want to be doing at each step is picking the next character to go, and not always assuming the next one had to be the next one in sequence, so I would do more like a permute kind of loop that's like pick the next one that goes, remove it from what remains and recur, and that I need that separate step we talked about of – and in addition to kind of picking, we also have to leave open the opportunity that we didn't pick anything and we just kind of left the subject as is, right, so we could [inaudible] or not.

And so permute always assumes we have to have picked everything. The subset code would also allow for some of them just being entirely skipped. So we pick the next one, right, and then eventually stopped picking.

Student:[Inaudible] just in your wrapper function then [inaudible] put a for loop or something like that, right? When you're through changing your string?

Instructor (Julie Zelenski):Yeah, you can certainly do that, like in a permute from the outside too, right. So there's often very, you know, multiple different ways you can get it the same thing whether you want to put it in the recursion or outside the recursion, have the set help you do it or not, that can get you the same result.

So let me try to identify what's the same about these to try to kind of back away from it and kind of move just to see how these are more similar than they are different, right, even though the code ends up kind of being – having some details in it that – if you kind of look at it from far away these are both problems that are about choice. That the recursive calls that we see have kind of a branching structure and a depth factor that

actually relates to the problem if you think about it in terms of decisions, that in making a permutation your decision is what character goes next, right? In the subset it's like, well, whether this character goes in or not that the recursive tree that I was drawing that it shows all those calls, the depths of it represents the number of choices you make.

Each recursive call makes a choice, right? A yes, no, or a next letter to go, and then recurs from there. So I make, you know, one of those calls, and then there's N minus 1 beneath it that represent the further decisions I make that builds on that. And so, in the case of permutation, once I've picked the, you know, N minus 1 things to go, there's only one thing left, right? And so in some sense the tree is N because I have to pick N things that go in the permutation and then I'm done.

In the subsets, it's also N, and that's because for each of the characters I'm deciding whether it's in or out. So I looked at A and decided in or out. I looked at B and decided in or out. And I did that all the way down. That was the life of the input. The branching represents how many recursive calls were made at each level of the recursion. In the case of subsets, it's always exactly two. In, out, all the way down, restarts at 1, branches to 2, goes to 4, goes to 8, 16, and so on.

In the permute case, right, there are N calls at the beginning. I have N different letters to choose from, so it's a very wide spread there, and that at that next level, it's N minus 1. Still very wide. And N minus 2. And so the overall tree has kind of an N times N minus 1 times N minus 2 all the way down to the bottom, which the factorial function – which grows very, very quickly.

Even for small inputs, right, the number of permutations is enormous. The number of subsets is to the end in or out, right, all the way across. Also, a very, you know, resource intensive problem to solve, not nearly as bad as permutation, but both of them, even for small sizes of N, start to become pretty quickly intractable. This is not the fault of recursion, right, these problems are not hard to solve because we're solving them in the wrong way.

It's because there are N factorial permutations. There are [inaudible] different subsets, right? Anything that's going to print to the N things, or N factorial things is going to do N factorial work to do so. You can't avoid it. There's a big amount of work to be done in there. But what we're trying to look at here is this idea that those trees, right, represent decisions. There's some decisions that are made, you know, a decision is made at each level of recursion, which then is kind of a little bit closer to having no more decisions to make. You have so many decisions to make, which is the depth of the recursion. Once you've made all those decisions, you hit your base case and you're done.

The tree being very wide and very deep makes for expensive exploration. What we're gonna look at is a way that we can take the same structure of the problem, one that fundamentally could be exhaustive, exhaustive meaning tried every possible combination, every possible rearrangement and option, and only explore some part of the

tree. So only look around in some region, and as soon as we find something that's good enough.

So in the case, for example, of a search problem, it might be that we're searching this space that could potentially cause us to look at every option, but we're also willing to say if we make some decisions that turn out good enough, that get us to our goal, or whatever it is we're looking for, we won't have to keep working any farther. So I'm gonna show you how we can take an exhaustive recursion problem and turn it into what's called a recursive backtracker.

So there's a lot of text on this slide but let me just tell you in English what we're trying to get at. That the idea behind permutations or subsets is that at every level there's all these choices and we're gonna try every single one of them. Now imagine we were gonna make a little bit less a guarantee about that. Let's design the function to return some sort of state that's like I succeeded or I failed. Did I find what I was looking for? At each call, I still have the possibility of multiple calls of in out, or a choice from what's there. I'm gonna go ahead and make the choice, make the recursive call, and then catch the result from that recursive call, and see whether it succeeded.

Was that a good choice? Did that choice get me to where I wanted to be? If it did, then I'm done. I won't try anything else. So I'll stop early, quite going around the for loop, quit making other recursive calls, and just immediately [inaudible] say I'm done. If it didn't – it came back with a failure, some sort of code that said it didn't get where I want to do, then I'll try a different choice. And, again, I'll be optimistic. It's a very optimistic way of doing stuff. It says make a choice, assume it's a good one, and go with it. Only when you learn it didn't work out do you revisit that decision and back up and try again. So let me show you the code. I think it's gonna make more sense, actually, if I do it in terms of looking at what the code looks like. This is the pseudo code at which all recursive backtrackers at their heart come down to this pattern.

So what I – I tried to be abstract [inaudible] so what does it mean to solve a problem, and what's the configuration? That depends on the domain and the problem we're trying to solve. But the structure of them all looks the same in that sense that if there are choices to be made – so the idea is that we cast the problem as a decision problem. There are a bunch of choices to be made. Each [inaudible] will make one choice and then attempt to recursively solve it.

So there's some available choices, in or out, or one of next, or where to place a tile on a board, or whether to take a left or right turn and go straight in a maze. Any of these things could be the choices here. We make a choice, we feel good about it, we commit to it, and we say, well, if we can solve from here – so we kind of update our statement so we've made that term, or, you know, chosen that letter, whatever it is we're doing.

If that recursive call returned true then we return true, so we don't do any unwinding. We don't try all the other choices. We stop that for loop early. We say that worked. That was good enough. If the solve came back with a negative result, that causes us to unmake that

choice, and then we come back around here and we try another one. Again, we're optimistic. Okay, left didn't work, go straight. If straight doesn't work, okay, go right. If right didn't work and we don't have any more choices, then we return false, and this false is the one that then causes some earlier decision to get unmade which allows us to revisit some earlier optimistic choice, undo it, and try again.

The base case is hit when we run out of choices where we've – whatever configuration we're at is, like, okay, we're at a dead end, or we're at the goal, or we've run out of letters to try. Whatever it is, right, that tells us, okay, we didn't – there's nothing more to decide. Is this where we wanted to be? Did it solve the problem? And I'm being kind of very deliberate today about what does it mean [inaudible] update the configuration, or what does it mean for it to be the goal state because for different problems it definitely means different things. But the code for them all looks the same kind of in its skeletal form.

So let me take a piece of code and turn it into a recursive backtracker with you. So I've got recursive permute up here. So as it is, recursive permute tries every possible permutation, prints them all. What I'm interested in is is this sequence a letters an anagram. Meaning, it is – can be rearranged into something that's an English word.

Okay. So what I'm gonna do is I'm gonna go in and edit it. The first thing I'm gonna do is I'm gonna change it to where it's gonna return some information. That information's gonna be yes it works, no it didn't. Okay? Now I'm gonna do this. I'm gonna add a parameter to this because I – in order to tell that it's a word I have to have someplace to look it up. I'm gonna use the lexicon that actually we're using on this assignment.

And so when I get to the bottom and I have no more choices, I've got some permutation I've assembled here in – so far. And I'm going to check and see if it's in the dictionary. If the dictionary says that that's a word then I'm gonna say this was good. That permutation was good. You don't need to look at any more permutations. Otherwise I'll return false which will say, well, this thing isn't a word. Why don't you try again?

I'm gonna change the name of the function while I'm at it to be a little more descriptive, and we'll call it is anagram. And then when I make the call here [inaudible] third argument. I'm not just gonna make the call and let it go away. I really want to know did that work, so I'm gonna say, well, if it's an anagram then return true. So if given the choice I made – I've got these letters X J Q P A B C, right? It'll pick a letter and it'll say, well if, you know, put that letter in the front and then go for it.

If you can make a word out of having made that choice, out of what remains, then you're done. You don't need to try anything else. Otherwise we'll come back around and make some of the further calls in that loop to see if it worked out. At the bottom, I also need another failure case here, and that comes when the earlier choices, right – so I got, let's say somebody has given me X J, and then it says, given X J, can you permute A and B to make a word?

Well, it turns out that you can – you know this ahead of time. It doesn't have the same vision you do. But it says X J? A B? There's just nothing you can do but it'll try them all dutifully. Tried A in the front and then B behind, and then tried B in the front and A behind it, and after it says that, it says, you know what, okay, that just isn't working, right? It must be some earlier decision that was really at fault. This returned false is going to cause the, you know, sort of stacked up previous anagram calls to say, oh yeah, that choice of X for the first letter was really questionable. So imagine I've had, like, E X, you know, T I. I'm trying to spell the word exit – is a possibility of it. That once I have that X in the front it turns out nothing is going to work out, but it's gonna go through and try X E I T, X E T I, and so on. Well, after all those things have been tried it says, you know what, X in the front wasn't it, right? Let's try again with I in the front. Well after it does that it won't get anywhere either. Eventually it'll try E in the front and then it won't have to try anything else after that because that will eventually work out. So if I put this guy in like that, and I build myself a lexicon, and then I change this to anagram word. I can't spell. I'd better pass my lexicon because I'm gonna need that to do my word lookups. [Inaudible]. And down here. Whoops. Okay. I think that looks like it's okay. Well, no – finish this thing off here. And so if I type in, you know, a word that I know is a word to begin with, like boat, I happen to know the way the permutations work [inaudible] try that right away and find that. What if I get it toab, you know, which is a rearrangement of that, it eventually did find them. What if I give it something like this, which there's just no way you can get that in there. So it seems to [inaudible] it's not telling us where the word is. I can actually go and change it. Maybe that'd be a nice thing to say. Why don't I print the word when it finds it? If lex dot contains – words so far – then print it. That way I can find out what word it thinks it made out of it. So if I type toab – now look at that, bota. Who would know? That dictionary's full of some really ridiculous words. Now I'll get back with exit. Let's type some other words. Query. So it's finding some words, and then if I give it some word that I know is just nonsense it won't print anything [inaudible] false. And so in this case, what it's doing is its come to a partial exploration, right, of the permutation tree based on this notion of being able to stop early on success. So in the case of this one, right, even six nonsense characters, it really did do the full permutation, in this case, the six factorial permutations, and discover that none of them worked. But in the case of exit or the boat that, you know, early in the process it may have kind of made a decision, okay so [inaudible] in this case it will try all the permutations with Q in the front, right? Which means, okay, we'll go with it, and then it'll do them in order to start with, but it'll start kind of rearranging and jumbling them up, and eventually, right, it will find something that did work with putting in the front, and it will never unmake that decision about Q. It will just sort of have – made that decision early, committed to it, and worked out, and then it covers a whole space of the tree that never got explored of R in front, and Y in the front, and E in the front because it had a notion of what is a satisfactory outcome. So the base case that it finally got to in this case met the standard for being a word was all it wanted to find, so it just had to work through the base cases until it found one, and potentially that could be very few of them relative the huge space. All right. I have this code actually on this slide. So it's permute, and that is turning into is anagram. And so, in blue, trying to highlight the things that changed in the process, right, that the structure of kind of how it's exploring, and making the recursive calls is exactly the same. But what we're using now is some return

information from the function to tell us how the progress went, and then having our base case return some sense of did we get where we wanted to be, and then when we make the recursive call, if it did succeed, right, we immediately return true and unwind out of the recursion, doing no further exploration, and in the case where all of our choices gave us no success, right, we will return the call that says, well that was unworkable how we got to where we were. So this is the transformation that you want to feel like you could actually sort of apply again and again, taking something that was exhaustive, and looked at a whole space, and then had – change it into a form where it's like, okay, well I wanted to stop early when I get to something that's good enough. A lot of problems, right, that are recursive backtrackers just end up being procedural code that got turned into this based on a goal that you wanted to get to being one of the possibilities of the exploration. Anybody have any questions of what we got there? Okay. I'm gonna show you some more just because they are – there are a million problems in this space, and the more of them you see, I think, the more the patterns will start to emerge. Each of these, right, we're gonna think of as decision problems, right, that we have some number of decisions to make, and we're gonna try to make a decision in each recursive call knowing that that gives us fewer decisions that we have to make in the smaller form of the sub problem that we've built that way, and then the decisions that we have open to us, the options there represent the different recursive calls we can make. Maybe it's a for loop, or maybe a list of the explicit alternatives that we have that will be open to us in any particular call. This is a CS kind of classic problem. It's one that, you know, it doesn't seem like it has a lot of utility but it's still interesting to think about, which is if you have an eight by eight chessboard, which is the standard chessboard size, and you had eight queen pieces, could you place those eight queens on the board in such a way that no queen is threatened by any other? The queen is the most powerful player on the board, right, can move any number of spaces horizontally, vertically, or diagonally on any straight line basically, and so it does seem like, you know, that there's a lot of contention on the board to get them all in there in such a way that they can't go after each other. And so if we think of this as a decision problem, each call's gonna make one decision and recur on the rest. The decisions we have to make are we need to place, you know, eight queens let's say, if the board is an eight by eight board. So at each call we could place one queen, leaving us with M minus 1 to go. The choices for that queen might be that one way to kind of keep our problem – just to manage the logistics of it is to say, well, we know that there's going to be a queen in each column, right, it certainly can't be that there's two in one column. So we can just do the problem column by column and say, well, the first thing we'll do is place a queen in the leftmost column. The next call will make a queen – place a queen in the column to the right of that, and then so on. So we'll work our way across the board from left to right, and then the choices for that queen will be any of the [inaudible] and some of those actually are – we may be able to easily eliminate as possibilities. So, for example, once this queen is down here in the bottommost row, and we move on to this next column, there's no reason to even try placing the queen right next to it because we can see that that immediately threatens. So what we'll try is, is there a spot in this column that works given the previous decisions I've made, and if so, make that decision and move on. And only if we learned that that decision, right, that we just made optimistically isn't successful will we back up and try again. So let me do a little demo with you. Kind of shows this doing its job. Okay. So [inaudible] I'm gonna do it as I said, kind of column

by column. [Inaudible] is that I'm placing the queen in the leftmost column to begin, and the question mark here says this is a spot under consideration. I look at the configuration I'm in, and I say, is this a plausible place to put the queen? And there's no reason not to, so I go ahead and let the queen sit there.

Okay, so now I'm going to make my second recursive call. I say I've placed one queen, now there's three more queens to go. Why don't we go ahead and place the queens that remain to the right of this. And so the next recursive call comes in and says, if you can place the queens given the decision I've already made, then tell me yes, and then I'll know all is good.

So it looks at the bottommost row, and it says, oh, no, that won't work, right? There's a queen right there. It looks at the next one and then sees the diagonal attack. Okay. Moves up to here and says, okay, that's good. That'll work, right? Looks at all of them and it's okay. So now it says, okay, well I've made two decision. There's just two to go. I'm feeling good about it. Go ahead and make the recursive call.

The third call comes in, looks at that row, not good, looks at that row, not good, looks at that row, not good, looks at that row, not good. Now this one – there weren't any options at all that were viable. We got here, and given the earlier decisions, and so the idea is that, given our optimism, right, we sort of made the calls and just sort of moved on. And now we've got in the situation where we have tried all the possibilities in this third recursive call and there was no way to make progress. It's gonna hit the return false at the bottom of the backtracking that says, you know what, there was some earlier problem.

There was no way I could have solved it given the two choices – or, you know, whatever – we don't even know what choices were made, but there were some previous choices made, and given the state that was passed to this call, there's no way to solve it from here. And so this is gonna trigger the backtracking. So that backtracking is coming back to an earlier decision that you made and unmaking it. It's a return false coming out of that third call that then causes the second call to try again.

And it goes up and it says okay, well where did I leave off? I tried the first couple of ones. Okay, let's try moving it up a notch and see how that goes. Then, again, optimistic, makes the call and goes for it. Can't do this one. That looks good. And now we're on our way to placing the last queen, feeling really comfortable and confidant, but discovering quickly, right, that there was no possible.

So it turns out this configuration with these three queens, not solvable. Something must be wrong. Back up to the most immediate decision. She knows it doesn't unmake, you know, earlier decisions until it really has been proven that that can't work, so at this point it says, okay, well let's try finding something else in this column. No go. That says, okay, well that one failed so it must be that I made the wrong decision in the second column.

Well, it turns out the second column – that was the last choice it had. So in fact it really was my first decision that got us off to the wrong foot. And now, having tried everything

that there was possible, given the queen in the lower left in realizing none of them worked out, it comes back and says, okay, let's try again, and at this point it actually will go fairly quickly. Making that initial first decision was the magic that got it solved.

And then we have a complete solution to the queens. We put it onto eight, and let it go. You can see it kind of checking the board, backing up, and you notice that it made that lower left decision kind of in – it's sticking to it, and so the idea is that it always backs up to the most recent decision point when it fails, and only after kind of that one has kind of tried all its options will it actually back up and consider a previous decision as being unworthy and revisiting it.

In this case that first decision did work out, the queen being in the lower left. It turns out there were – you know, you saw the second one had to kind of slowly get inched up in the second row. Right? It wasn't gonna work with the third row. It tried that for a while. Tried the fourth row for a while. All the possibilities after that, but eventually it was that fifth row that then kind of gave it the breathing room to get those other queens out there.

But it did not end up trying, for example, all the other positions for the queen in the first row, so it actually – it really looked at a much more constrained part of the entire search tree than an exhaustive recursion of the whole thing would have done. The code for that – whoops – looks something like this. And one of the things that I'll strongly encourage you to do when you're writing a recursive backtracking routine, something I learned from Stuart Regis, who long ago was my mentor, was the idea that when – trying to make this code look as simple as possible, that one of the things you want to do is try to move away the details of the problem.

For example, like, is safe – given the current placement of queens, and the row you're trying, and the column you're at, trying to decide that there's not some existing conflict on the board with the queen already being threatened by an existing queen just involves us kind of traipsing across the grid and looking at different things. But putting in its own helper function makes this code much easier to read, right?

Similarly, placing the queen in the board, removing the queen from the board, there are things they need to do. Go up to state and draw some things on the graphical display. Put all that code somewhere else so you don't have to look at it, and then this algorithm can be very easy to read. It's like for all of the row. So given the column we're trying to place a queen in, we've got this grid of boolean that shows where the queens are so far, that for all of the rows across the board, if, right, it's safe to place a queen in that row and this column, then place the queen and see if you can solve starting from the column to the right, given this new update to the board.

If it worked out, great, nothing more we need to do. Otherwise we need to take back that queen, unmake that decision, and try again. Try a higher row. Try a higher row, right. Again, assume it's gonna work out. If it does, great. If it doesn't, unmake it, try it again. If we tried all the rows that were open to us, and we never got to this case where this returned true, then we return false, which causes some previous one – we're backing up

to a column behind it. So if we were currently trying to put a queen in column two, and we end up returning false, it's gonna cause column one to unmake a decision and move the queen up a little bit higher.

If all of the options for column one fail, it'll back up to column zero. The base case here at the end, is if we ever get to where the column is past the number of columns on the board, then that means we placed a queen all the way across the board and we're in success land. So all this code kind of looks the same kind of standing back from it, right, it's like, for each choice, if you can make that choice, make it. If you solved it from here, great, otherwise, unmake that choice.

Here's my return false when I ran out of options. There's my base case – it says if I have gotten to where there's no more decisions to make, I've placed all the queens, I've chosen all the letters, whatever, did I – am I where I wanted to be? There's no some sort of true or false analysis that comes out there about being in the right state. How do you feel about that? You guys look tired today, and I didn't even give you an assignment due today, so this can't be my fault, right?

I got a couple more examples, and I'm probably actually just gonna go ahead and try to spend some time on them on Monday because I really don't want to – I want to give you a little bit more practice though. So we'll see. We'll see. I'll do at least one or two more of them on Monday before we start talking about pointers and linked lists, and so I will see you then. But having a good weekend. Come and hang out in Turbine with me.

[End of Audio]

Duration: 50 minutes

ProgrammingAbstractions-Lecture11

Instructor (Julie Zelenski):Hi there. Good afternoon. Welcome to Monday. Today's a big day. A big day, right? We've got a couple of recursive backtracking examples that we didn't get to on Friday that I'm gonna talk you through today, and then we're gonna talk a little bit about pointers, the long anticipated introduction to one of the scarier parts of the C++ language as kind of a step toward building linked lists and the recursive data idea that we will study today and continue into Wednesday.

And so the material at this point jumps around a little bit, right? We go back and pick up some of the pointers in array information that was in earlier chapters. Linked lists are covered a little bit later in kind of a different context that is – you can do but it's not the best match to how we're covering it here. And then handout 21, which I gave out today, is more similar to the way I'm going to be showing you linked lists and its concepts.

Once we get the linked list up, we go back to the reader of chapter seven looking at algorithms and big O. We'll spend actually several days on that on sorting, and analysis of algorithms, and things like that. [Inaudible] you guys should be working on that, right, coming in on Wednesday, right, some good practice getting your recursive decomposition skills down and figuring out how to work your way toward the base case and things like that.

And then what goes out at [inaudible] will be actually kind of your first really big complete program, right, it is the venerable bottle that you may have heard of because actually it's such a legend in the 106 program that kind of brings together a lot of the stuff, ADTs, and recursion, and all sorts of things we study all term kind of build one big complete program now that we've kind of got a bunch of skills to put together on that, which will go out on Wednesday when assignment three comes in.

Note that tomorrow's Super Tuesday, so if you are resident of a state who is one of the 24 or so who are participating in tomorrow's big primary, be sure to get out and vote. Anything administratively you want to ask about? Questions? How many people have done, you know, at least one of the problems on the recursion problem set now? Oh, yeah, yeah. How many of you have done all of them?

Not quite. Okay. Anybody who's gotten along the way have any insights that they want to offer up to their – those who are a little further behind you? Any way of lending a hand to your fellow student?

Student: Draw a diagram.

Instructor (Julie Zelenski):Draw a diagram. What kind of diagrams have you drawn?

Student: Like, each step [inaudible] trying to figure out what it's doing if I go all the way down and it's a little difficult.

Instructor (Julie Zelenski):So he's suggesting here, right, start with, you know, one of your bigger cases. Maybe that's gonna take four or five, you know, calls before it hits that base case, and watch it do its work, right, think about, okay, what the first call makes, what the second call makes, what the third call makes, make sure you're working toward that base case, and see how it both goes down into the calls and then unwinds its way back out, right, can definitely help a lot.

For the ones that have a pretty high branching factor, that gets a little bit tricky, right, to sort of – [inaudible] has a five way branch with a five way branch under it, it would go a little crazy, so you'd have to pick some pretty small examples for the more complex problems. But certainly for the simple cases, right, being able to do that. Anything else? Yeah.

Student:[Inaudible].

Instructor (Julie Zelenski):Yes. So the [inaudible] we gave you, right, you really do need to match our prototypes, but they are very likely in many cases to not be enough, right, they'll get you started but there's gonna be more housekeeping. You're gonna be keeping them along the way, so probably a lot of them are just gonna be those one line wrappers that make a call into your real recursive function that then picks up the outer state plus some other kind of housekeeping to work its way down the recursive call.

So yeah, it's definitely true. A lot of little one-line wrappers in our prototype going into your recursive call. Over here.

Student: This doesn't really have to do with recursion, but go back to [inaudible] I guess C++ or header file, you need to, like, physically move it to the right folder.

Instructor (Julie Zelenski):Yes. Yeah, sure.

Student: Add it in Visual Studio doesn't do it. It never compiles.

Instructor (Julie Zelenski):Yes, so you – when we give you a .cpp file, right, with some code included in the project, you really have to get it into the right place and get your project to include it, otherwise it'll turn up saying I've never heard of this lexicon, you know, it will fail to compile or link one or the other, depending on which step it got hung up on. So if we give you some new code, make sure you incorporate it into your project, right, so that it actually is kind of built into it, and you can use that code in solving your problems. Anything else? Okay. Oh, wait.

Student:[Inaudible].

Instructor (Julie Zelenski):The what?

Student: Failure cases?

Instructor (Julie Zelenski):Yes, failure cases, right? Like if – often you get focused on what the truth will be, what the right answer – get to the success case, and then kind of completely ignore these other things about what about the dead ends, right, the things that are going nowhere. For example, on the phone T9 Text one, right, there definitely are some cases where you have to kind of stop things going down dead ends, and if you don't, right, you can get into this sort of nasty exhaustive, you know, infinite recursion that can really make quite a mess of things.

So making sure you're thinking both about how you know when you got to where you want to be, and where you get to something that you don't want to be but that you can back out of. So the two samples I want to do are both based on the same backtracking pseudocode that we were using on Friday. I just want to go through them. I'm gonna do a little bit less attention to the code and a little bit more attention to the problem solving because at some point I think the problem solving is really where the tricky stuff comes on.

And then the kind of – turning it into code, there's some details there but they're not actually as important, so I'm gonna de-emphasize that just a little bit here and think more about solving it. So this is the pseudocode for any form of a backtracker, right, that has some sort of, you know, failure and success cases, like when we run out of choices, we've hit a dead end, or we've hit a goal state, we've – you know, there's no more decisions to make, right, is this want to be, yes or no, and then otherwise there are some decisions to make.

And for all the possible decisions we could make, we're gonna try one, be optimistic about it working out, make that recursive call that says that choice was where I wanted to be. If it returns true, or whatever the success return value is, then we return true, right? No need to look any further. That choice was good enough. If it didn't work then we gotta unmake – try some other choices. If we try all the things available to us and no case, right, did solve ever return true, then we can only conclude that the configuration as given to this call was unsolvable, which is where that return false causes it to back up to some earlier call and start unmaking that next layer of decisions, and then recur further down, eventually either unwinding the entire recursive [inaudible] all the way back to the beginning and saying it was totally unsolvable no matter what, or eventually finding some sequence of decisions that will lead to one that will get us to a success case. So the one I'm gonna show you here is the sudoku, which is those little puzzles that show up in newspapers everywhere. They're actually not attributed to – apparently, to the Japanese, but it apparently got a lot more popular under its Japanese name than it did under the original English name for it. And the goal of a sudoku, if you haven't ever done one, right, is it's a nine by nine grid in which you're trying to place numbers, and the requirement for the numbers is such that within any particular row, or any particular column, or any particular block, which is a three by three subsection of that, the numbers one through nine each appear once. So you can never have more than two ones in a row, two twos in a column, three threes in a block, or anything like that. And so there has to be some rearrangement, or, in fact, permutation of the numbers one through nine in each row, in each column, and each block, such that the whole puzzle kind of works out

logically. So when it's given to you, you know, usually some fraction of the slots are already filled in, and the goal for you is to fill in those remaining slots without violating any of these rules. Now the sort of pure sudoku solvers don't really use guessing. It's considered, actually, poor form, you know, that you're supposed to actually logically work it out by constraints about what has to be true here versus what has to be true there to kind of realize that – what choices you have. We're actually not gonna be a pure, artistic, you know, sudoku solver. What we're gonna do is we're actually gonna use a brute force recursive algorithm that's just gonna try recursive backtracking, which is to say, make an assignment, be optimistic about it, and if it works out, great, and if not, we'll eventually come back to that decision and revisit it. So what we have here basically is a big decision problem. You know, of the 81 squares on here, you know, about 50 of them, right, need to be chosen. For each of those 50 squares, right, we're gonna do them one at a time and recur on the remaining ones. So, you know, choose one then we'll just go left to right from the top. Choose that one at the top, make an assignment that works, and so that's what we'll use the context we have in problem here. So, for example, if you look at this first row, there's a one in this column so we can't use one. There's a two in that block so we can't use two, but there's not a three in either that row, or that column, or that block, so we'll say, well, three looks good, you know, just trying the numbers in order. We'll be optimistic, say that works, and say, well if we planted a three here, could we recursively solve the remaining 49 holes and work it out? And so we get to that next one – we look at this one and say, okay, well we could put a one here, right, because there's not a one in this column, not a one in that row, not a one in that block, so we'll kind of move on. I'm gonna do a little demo of that for you. And maybe it's to kind of keep moving our way across, and only when we get to a dead end based on some of our earlier decisions will we unmake and come back. So let me – okay. So that's the same set of numbers there. So I put the three up in the corner. And so it puts the one here thinking, okay, that looks good. So it gets to the next square over here and then the one can't be used because it's actually already in use both in that column and in the row we're building so far, but two can be used. Two doesn't conflict with anything we have so far. And so it just keeps going optimistically, right? At that stage over there it turns out almost all the numbers are in use. Most of the numbers all the way up through seven and nine are there, and seven's in its column. So, in fact, the only number that that could possibly be is nine, so the only choice we have here to try is nine, and then we'll place the seven next to it. And so now we have a whole row that doesn't conflict with any of its blocks this way, and then we just keep moving on, right, so that is to keep kind of going from top to bottom, left to right. We'll place that four. We'll place another one. Place a three. So it's actually choosing them – it's actually going in order from one to nine just picking the first one that fits, right, so when we get to the next square here, right, it can't use one, it can't use two, it can't use three, it can't use four, it can't use five because they're all in that row. It can't use six because it's in the column. It can't use seven because it's in that row, but it should be able to use eight, and so it'll place an eight there. So it just kind of examines them in sequence until it finds the first one that doesn't violate any things already decided, and then kind of moves optimistically forward. So about this point, right, we're doing pretty well, but we're starting to run into some troubles if you look at the next one over here. It'll place the six there, but then it will – once the six is placed there, then it looks to the right and it says, oh, I need to put a nine there. That's the

only number left. It tries all the numbers one, two, three, four, and it says that isn't gonna work. And so it actually fails on the rightmost column, which causes it to back up to the one right before and it says, well, why don't you try something else here? Well, it looks at seven, eight, and nine, none of those can work either, so it's actually gonna back up even further and then say, well what if we try to put a nine here? That also doesn't work. So now it's gonna start seeing that it's kind of unwinding as the constraints we have made have kind of got us down a dead end and it's slowly working its way back out to where the original bad decision was made. So it tries again on moving nine in here, and moving across, right, but again, kind of, you know, working its way forward but then kind of backing its way up. And let me go ahead and just run it faster so you can kind of see it. But, you know, it's working on that row for a while, but essentially to note that the top row stays relatively constant. It kind of believes that, well, that first three must have been good because, you know, we're getting somewhere on that, and it keeps kind of going. You can see that the activity is kind of always at the kind of tail end of that decision making, which eventually, right, worked its way out. And so it turns out, like, those three ones that we did put in the first spots were fine. That is, choices, right, it did work out. We didn't know that when we started, right, but it was optimistic, right, it put it down and then kept moving, and then eventually, right, worked out how the other things that had to get placed to make the whole puzzle be solvable. And so this thing can solve, actually, any solvable sudoku, right, and if it's not animating, instantaneously, even though it is really doing it in a fairly crude way, right, it's basically just trying everything it can, moving forward, and only when it kind of reaches a contradiction based on some of those choices that they will – that can't possibly be because at this point I'm forced into a situation where there's nothing that works in this square, so it must be that some earlier decision was wrong. And you notice that when it backs up, it backs up to the most immediate decision. So if you think of it in terms of recursive call, here's your first decision, your second decision, your third decision, your fourth decision, your fifth decision. If you get down here and you're, like, trying to make your eighth decision, and there's nothing that works, right, the decision that you come back to will be your seventh one. The one right before it. You don't go throw everything away and start over. You don't go all the way back to the beginning and say, oh, well that didn't work, let's try again. It's that you actually use the kind of context to say, well, the last decision I made was probably the one that needs a little fixing. Let me just back right up to that one. That's the way the calls unwind, and it says we'll pick up trying some other options there. Which ones have we not tried on that one? And then go forward again, and again, if you get down to that eighth decision and you're still stuck, you come back to the seventh decision again, and only after kind of the seventh decision has gone back and forth with the eighth unsuccessfully through all its options would we eventually return to that sixth decision, and potentially back to the fifth, and fourth, and so on. The code for this guy, a little abstracted that should very much fit the pattern of what you think recursive backtracking looks like, and then the kind of sort of goofier parts that are about, well, what does it mean to test a sudoku for being a sign of having conflicts is actually then passed out into these helper functions that manage the more domain specific parts of the problem. So at the very beginning it's like, find an unassigned location on the grid, and it returns the row and column by reference. It turns out in this case those are reference parameters. So [inaudible] searches from top to bottom to find the first slot that actually

does not have a value currently in it. If it never finds one, so exhaustively searched the grid and didn't find one, then it must be that we have a working sudoku because we never put a number in unless it worked for us, and so if we've managed to assign them all, we're done. If this didn't return true, that meant it found one, and it assigned them a row and column, and then what we're gonna go through the process of is assigning that row and column. So we look at the numbers one through nine. If there are no conflicts for that number, so it doesn't conflict with the row, column, or block, that number isn't already in use in one of those places, then we go ahead and make the assignment, and then we see if we can solve it from here. So having updated the grid to show that new number's in play, you know, if we move on, the next [inaudible] of sudoku will then do a search for find unassigned location. This time the one that we previously found, right, has been assigned, so it actually won't get triggered on that one. It'll look past it further down into the puzzle, and eventually either find the next one to make a call on, and kind of work its way through, or to – you have to solve the whole thing. If it didn't work, so we made that assignment to the number nine, and we went to solve, and eventually this tried all its possibilities from here and nothing came up good, then this unassigned constant is used to unmake that decision, and come back around, and try assigning it a different number. If we try all of our examples, so for example, if we never find one that doesn't already conflict, or if we try each one and it comes back false, right, that this return false here is what's triggering the backtracking up to the previous recursive call to reconsider some earlier decision – the most recent early decision for this one, and say that was really our mistake, right, we've got to unmake that one. So it should look like kind of all the recursive backtracking all through looks the same, right? You know, if we're at the end, here's our base cases for all our options. If we can make this choice, make it, try to solve it, be optimistic, if it works out, return. Otherwise, unmake it, allow the loop to iterate a few more times trying again. If all of those things fail to find a solution then that return false here will cause the backtracking out of this decision. I just let it become constant. You know, I made it up. I used negative one, in fact, just to know that it has no contents. Just specific to sudoku in this case. Now would be a great time to ask a question. You guys kind of have that sort of half-okay look on your face. It could be I'm totally bored. It could be I'm totally lost.

Student:

Where do row and column get assigned?

Instructor (Julie Zelenski):So they – finding assigned location takes [inaudible] reference. So if you look at the full code for this – this is actually in the handout I gave out last time, and so there's a pass by reference in that function, and it returns true if it assigned them something, and then they have the coordinates of that unassigned location. Question over here.

Student: How'd you know to write sol sudoku as a bool rather than as, like, a void function?

Instructor (Julie Zelenski):Well, in this case – that's a great question, right, in most cases in recursive backtracking I am trying to discover the success or failure of an operation, and so that's a good way to tell because otherwise I need to know did it work. And so if I made it void then there had to be some other way I figured it out. Maybe, you know, that you have to check the grid when you're done and see that it has – no longer has any unassigned locations, but the bool is actually just the easiest way to get that information out.

So typically your recursive backtracking machine will probably return something. Often it's a true or false. It could be, you know, in some other case, just some other good know value, and some other sentinel that says, you know, bad value. So, for example, in the finding an anagram of the words it might be that it returned the word if it found one, or returned an empty string if it didn't. So using some sort of state that says here's how – if you made the call, you'll know whether it worked because we really do need to know. Make the call and find out whether it worked. Well, how are we gonna find out? One of the best ways to get that information is from a return value. Way in the back?

Student: Exactly does the return calls trigger in the backtracking [inaudible]?

Instructor (Julie Zelenski):So think about this as that, you know, it's hard to think about because you have to kind of have kind of both sides of the recursion in your head, but when – let's say we're on the fifth decision, right? We make the call to solve the sudoku that's gonna look at the sixth decision, right? If the sixth decision fails, it says, I looked at all my choices and none of them worked, right? It's that that causes the fifth decision to say, well I – here go solve for the sixth decision down.

Well the sixth decision came back with a false. It got to this case after trying all its options, then it's the fifth decision that then says, okay, well then I'd better unmake the decision I made and try again. And so at any given stage you can think about the caller and the callee, it's saying, I'm making a decision. You tell me if you can make it work from here. If the delegate that you passed on the smaller form of the recursive problem comes back with a return false, and then I've tried everything I could possibly do, but there was no way.

The situation you gave me was unworkable. And that says, oh, okay, well you know what, it must have been my fault, right? I put the wrong number in my box. Let me try a different number and now you try again. And so they're constantly kind of these – you have to keep active in your mind the idea that there's this whole chain of these things being stacked up, each of them being optimistic and then delegating down.

And if your delegate comes back with the – there was nothing I could do, then you have to revisit your earlier optimistic decision, unmake it, and try again. And so that – this is the return false to the outer call, the one that made the solved call that unwinds from. Way over here?

Student:[Inaudible]?

Instructor (Julie Zelenski):Pretty much any recursive piece of thing can be reformed into a backtracking form, right? You need to – to do a little bit like the [inaudible] we tried last time, we'll take a void returning thing and make it return some true or false, and there's kind of, like, make a decision and be optimistic, see if it worked out. But that kind of rearrangement of the code should work with pretty much all your standard recursion stuff.

So any kind of puzzle that actually, like, all these kind of jumble puzzles, and crossword puzzles, and things that involve kind of choosing, right, can be solved with recursive backtracking. It's not necessarily the fastest, right, way to get to a solution, but it will exhaustively try all the options until a sequence of decisions, right, leads you to a goal if there is a goal to be found.

And so that if you can think of it as – if you can think of your problem as a decision problem – I need to make some decisions, and then from there make some more decisions and so on, and eventually, right, I will bottom out either at a goal or dead end, then you can write a recursive problem solving – backtracker to solve it. Way in the back?

Student: This doesn't have anything to do with recursion, but how did you slow down the simulation?

Instructor (Julie Zelenski):How did I slow it down? I'm just using pause. There's a pause function in our stenographics library, and you just give it a time. You say one second, half a second, and just – I use that a lot in our demos, for example, for the maze, so that you can actually see what's going on, and that way it just animates a little more. Stenographics, CSTgraph.h. [Inaudible] me show you one more kind of just in the same theme. The idea of taking sort of some, you know, puzzle that somebody might give you, trying to cast it in terms of a decision problem, right, where you make decisions and then you move on, can I solve something like these little cryptographic puzzles? So here's send plus more equals money. I've got eight digits in there – eight letters in there. You know, D E M N O R S Y across all of them, and the goal of this puzzle is to assign these letters – eight letters to the digits zero through nine without replacement so that if D's assigned to three, it's assigned to three in all places. For example, O shows up two places in the puzzle. Both of those are either a two or both are three. There's not one that's one and one that's another. And each digit is used once. So, like, if I assigned two to O, then I will not assign two to D. So what we've got here is eight letters. We've got 10 digits. We want to make an assignment. What we've got here is some choices. The choices are, for each letter, what digit are we gonna map it to? Another way to think about it is if you took these eight letters, and then you attach two dashes on the end, then you considered that the letter's position in the string was – the index of it was its digit. It's really just like trying the permutations, right, rearranging those letters in any of those sequences. So right now, for example, maybe D is assigned zero, and one, and two, and so on, and these guys are eight, nine. Well, if you rearrange the letters into some other permutation then you've made a different assignment. So in effect, right, what you're looking at is a problem that has a lot in common with permutations, right? I'm gonna make an assignment, take a letter off, decide what index it's gonna be placed at, so it's kind of like

deciding where it would go in the permutation, and then that leaves us with a smaller form of the problem which has one fewer letters to be assigned, and then recursively explore the options from there to see if we can make something that makes the addition add up correctly that D plus E equals Y means that if D was assigned five, and E was three, then Y better be eight for this to work out. So the first form I'm gonna show you is actually just the dumb, exhaustive recursive backtracking that works very much like the sudoku problem where it just – it finds the next unassigned letter, it assigns it a digit of the next auto sign digits, right, and then just optimistically figures that'll work out. After it makes an assignment for all the letters it says, take the puzzle, convert all the letters into their digit equivalents, and see if it adds together correctly. If so, we're done. If not, then let's backtrack. So let me – I mean, actually, I'll show you the code first because it's actually quite easy to take a look at. Again, it has helper routines that kind of try to abstract the pieces of the puzzle that actually aren't interesting from kind of looking at just the core of the recursive algorithm, so it looks a lot like sudoku in that if letters do assign, so it actually keeps the string of the letters that haven't yet been assigned. If there are no more letters in that string, we take one off at each time we assign it, then we check and see if the puzzle's solved. So that kind of does the substitution, and does the math, and comes back saying yes or no. If it worked out, we'll get a true. If it didn't work out we get a false. If we still have letters to assign then it goes through the process of making a choice, and that choice is looking at the digits zero through nine if we can assign it. So looking at that first letter in that digit, and then that's making sure that we don't already have an assignment for that letter, that we don't have an assignment for that digit. Sort of make sure that just the constraints of the problem are being met. If we're able to make that assignment then we go ahead and make a recursive call, having one fewer letters to make a decision for, and if that worked out true, otherwise we do an unassignment and come back around that loop and then eventually the same return false at the bottom, which said, well given the previous assignments of the letters before we got to this call, there was nothing I could do from here to make this work out. So let me show you this guy working because you're gonna see that it is actually a crazy way to try to solve this problem in terms of what you know about stuff. So I say C S plus U equals fun, a well-known equation. So I did this little animation to try to get you visualized what's going on at any given stage. So it has the letters down across the bottom, S U N C O Y F. That's the string of letters to assign. It's gonna assign it the next available digit when it makes that recursive call, so the first recursive call is gonna be about assigning S, and then the next call will be assign U, and the next call – and so on. So it always gets up to seven deep on any particular thing. And so it says, okay, well first digit available is zero, go ahead and assign that to S. So it's gonna make a call there. And then it says, okay, well look at U. Well we can't use zero because zero's already in use. What don't we assign U to be one? Okay. Sounds good. Keep going. And then it'll say, okay, let's get to N. Let's make an assignment for N. It says I'll look around. Okay, well zero's in use, one's in use, how about two? Okay. Good. And then it assigns this to three, this to four, this to five, and this to six. Okay. It gets to here and it says, hey, does that work, 30 plus 541 equals 612? No? Okay. Well, you know what the problem was? It was F, right? I assigned F to six. How stupid of me. It can't be six. Let's make it seven. And then it says, oh, oh no, oh I'm sorry. I'm sorry, 30 plus 541 that's not 712. How silly. I need to assign F to be eight. And it's gonna do this for a little while. A long while [inaudible]. And then it says, oh,

okay. Well, you know what, given the letters you'd assigned to the first six things, when you got to F, I tried everything I could and there was nothing I could do to fix this. You know what the problem was? It's not me. It's not me. I'm not the problem. You're the problem. So it returns false from this call at the bottom, having tried all its options, which causes Y to say, oh, yeah, yeah, I know. I know I said five. Did I said five? I didn't meant to say five. I meant to say six. And so it moves up to the six option. Again, optimistically saying that's good. Go for it. See what you can do. So it picks five. That won't work, right? It picks seven. It's gonna go for a long time, right, because it turns out, right, this is one of those cases where that very first decision, that was one of your problems, right? If you assign S to be zero, there's nothing you can assign U and N to be that are gonna work out. So what it's gonna be going through is this process though of having, you know, committed to this early decision and kind of moving on it's gonna try every other variation over here before it gives up on that. So let me set it to going. Even though C S plus U does equal fun. I guarantee it. We'll let it do some work for a while. So those bars is they grow up are desperation. You can think of that as, like, it's running out of options. It's running out of time, you know, and it's like oh, oh, wait, earlier, back up, back up. And so, okay, you can kind of see how far its backed up but sort of how tall some of the deeper recursive calls, right, the earlier ones in the sequence. And so it doesn't, you know, revisit any of these decisions because it's really busy over here, but you can see that C is now up to seven. It'll get up to eight. It'll get up to nine. And that was when it will cause itself to say, you know, I tried everything that was possible from C on down, and there was no way to make this thing work out. It must be that the bad decision was made earlier. Let's back up and look at that. And so it'll come back to the decision N, bump it up by one, and then go again. It's gonna do this a long time, right? Go through all the options for N and its neighbors before it comes back and revisits the U. It's gonna have to get U all the way up, right, through having tried one, two, three, four. So adding zero to one, and two, and three, and four, and discovering it can never make a match over there before it will eventually decide that the S is really where we got ourselves in trouble. So this is kind of in its most naïve form, right? The recursive backtracking is doing basically permutations, right? You can see this is actually just permuting the digits as they're assigned to the numbers, and there are, in this case, you know, seven factorial different ways that we can make this assignment, and there are a lot of them that are wrong, right? It's not being at all clever about how to pick and choose among which to explore. So in its most naïve form, right, recursive backtracking can be very expensive because often you're looking at things that have very high branching, and very long depth to them, which can add up to a lot of different things tried. Just using some simple – in this case, heuristics, sort of information you know about the domain can help you to guide your choices instead of just making the choices in order, trying the numbers zero through nine as though they're equally likely, or kind of waiting to make all the assignments before you look at anything to see if it's actually good. There's actually some ways we can kind of shape our decision making to look at the most likely options before we look at these more first. Finding the problem instead of niggling around this dead end. So I'm gonna let that guy work for a little bit while I show you another slide over here. So what the smarter solver does, is that it doesn't really consider all permutations equally plausible. We're gonna use a little grade school addition knowledge. If I look at the rightmost column, the least significant digit, I'm gonna assign

D first. I assign D to zero. I assign E to one, and then I look at Y – I don't try Y is five, or seven, or anything like that. I say there's exactly one thing Y has to be, right, if D is zero and E is one, then Y has to be one, and I can say, well that won't work because I already used one for E. So it'll say, well that's impossible. Something must be wrong early. Rather than keep going on this kind of dumb dead end, it's to realize right away that one of the decisions I've already made, D and E, has gotta be wrong. So it'll back up to E. It'll try two, three, four, very quickly realizing that of all the things you could assign to E, once you have assigned D to zero, you're in trouble. And so it will quickly unmake that decision about D and work its way down. So using the kind of structure of the problem. So it makes it a little bit more housekeeping about where I'm at, and what I'm doing, and what's going on, but it is using some real smarts about what part of the tree to explore, rather than letting it kind of just go willy nilly across the whole thing. Let me get that out of the way. And I'll run this one. I say C S plus U equals fun. Okay. So it goes zero here, and tries the one, and it says no, that won't work. How about the two? No, that won't work, right, because there's nothing you can assign the N that'll make this work. And so it immediately is kind of failing on this, and even after it tries kind of all nine of these, it says none of these are looking good, then it comes back to this decision and says, no, no, actually, S's a zero. That wasn't so hot. How about we try S as one? And then kind of works its way further down. Hello? Okay. So let me try this again. Let me get it to just go. Okay. So it took 30 assignments – 30 different things it tried before it was able to come up with 41 plus 582 equals 623, which does add correctly. It didn't have to unmake once it decided that S was one. It turns out that was a workable solution so it didn't have to go very far once it made that commitment, and then you can see it kind of working its way up. So 30 assignments, right, across this tree that has, you know, hundreds of thousands in the factorial, very large number, but very quickly kind of pruning down those things that aren't worth looking at, and so focusing its attention on those things that are more likely to work out using information about the problem. It doesn't really change what the recursion does. It's kind of interesting if you think that the same backtracking and recursive strategy's the same in all these problems, but what you're trying to do is pick your options, like, looking at the choices you have and trying to decide what are the more likely ones, which ones are not worth trying, right, so sort of directing your decision making from the standpoint of trying to make sure you don't violate constraints that later will come back to bite you, and things like that. This one's back here. It's at 13,000 and working hard. Getting desperate. Doing its thing. Still has not unmade the decision about S is zero, so you can get an idea of how much longer it's gonna take. We'll just let it keep going. I'm in no hurry – and let it do its thing. So let me – before I move away from talking about recursion, just try to get you thinking just a little bit about how the patterns we're seeing, right, are more alike than they are different. That solving a sudoku, solving the eight queens, you know, solving the find an anagram in a sequence of letters, that they all have this general idea of there being these decisions that you're making, and you're working your way down to where there's, you know – fewer and fewer of those decisions until eventually you end up, sort of, okay, I'm done. I've made all the decisions I can. What letter goes next, or whether something goes in or out. And the two problems that I call the kind of master or mother problems, right, of permutations and subsets are kind of fundamental to adapting them to these other domains. And so I'm gonna give you a couple examples of some things that come up that actually are just kind of

permutations, or subsets, or some variation that you can help me think about a little bit. One that the CS people are fond of is this idea of knapsack feeling. It's often cast as someone breaking into your house. All criminals at heart apparently in computer science. And you've got this sack, and you can put 50 pounds of stuff in it, right, and you're looking around, you know, at all the stuff that's up for grabs in this house that you've broken into, and you want to try to pack your sack, right, so that you've got 50 pounds of the high value stuff. So let's say there's, like, 100 things, right, you could pick up, right, and they weigh different amounts, and they're worth different amounts. What's the strategy for going about finding the optimal combination of things to stick into your sack, right, so that you got the maximum value from your heist? What problem does that look like that you've seen? It's a subset. It's a subset. [Inaudible]. Right? So if you look at the, you know, the Wii, and you say, oh, should I take the Wii? You know, it weighs this much, and it's this much value, well let's try it in and see what happens, right, and see what else I can stuff in the sack with that, right, and then see how well that works out. I should also try it with it out. So while you're standing there trying to decide what to steal, you have to type in all the values of things in every computer program. Go through the kind of machinations of well, try this with that because some things are big, but have a lot of value, but they only leave a little bit of odd space left over you might not be able to use well, or something. But what we're looking for is the optimal combination, the optimal subset. So trying the different subsets tells you how much value and weight you can get in a combination, and then you're looking for that best value you can get. You're the traveling salesman. You've got 10 cities to visit. Boston, New York, Phoenix, Minneapolis, right? You want to cover them in such a way that you spend the minimal amount of time, you know, in painful air travel across the U.S. Well you certainly don't want to be going Boston, New York, right, like, Boston, to L.A., to New York, to Seattle, to D.C., to Los Angeles back and forth, right? There's gotta be a pattern where you kind of visit the close cities and work your way to the far cities to kind of minimize your total distance overall. What problem was that really in disguise? We've got 10 cities. What are you trying to do? Help me out a little. I hear it almost. Louder. Permutations. You've got a permutation problem there, all right? You got 10 cities. They all have to show up, right? And it's a permutation. Where do you start, where do you end, where do you go in the middle, right? What sequencing, right, do you need to take? So what you're looking at is try the permutations, tell me which ones come up short. There are things, right, about heuristics that can help this, right? So the idea that certainly, like, the ones that are closer to you are likely to make a better choice than the longer one. So kind of using some information about the problem can help you decide which ones are more promising avenues to explore. But in the end it's a permutation problem. I'm trying to divide you guys into fair teams. I want to, you know, divide you up into 10 teams to have a kind of head-to-head programming competition. I happen to know all your Iqs. I don't know. Some other, you know, useless fact that perhaps we could use as some judge of your worth. And I want to make sure that each time has kind of a fair amount of IQ power in it relative to the others that I didn't put, you know, all the superstars on one team. What am I doing? How do I divide you guys up? It's a subset problem, right? So if – in this case it's a subset that's not just in or out. It's like which team am I gonna put you in? So I've got 10 subsets to build. But in the end it's an in out problem that looks like, well, I have the next student. Which of the 10 teams can I try them in? I can try them in each of them,

right? So in some sense it's trying in the first team, the second team, the third team, right, and then pull the next student, try him in those teams, and then see whether I can come up with something that appears to get a balance ratio when I'm done. It turns out you can take the letters from Richard Millhouse Dickson and you can extract and rearrange them to spell the word criminal. I am making no conclusion about anything, having done that, just an observation of fact. But that sort of process, right, I'm trying to find, well what is the longest word that you can extract from a sequence of letters? What kind of things is it using? Anything you've seen before? Oh, somebody come on. I hear the whispering but no one wants to just stand up and be committed. How about a little of both, right? The what?

Student:

Like the sudoku puzzle?

Instructor (Julie Zelenski):It's a little bit like the sudoku, right? It's got a permutation sort of backing, and it's got a little bit of a subset backing, right? That is that I'm not choosing all the letters from this. And in fact there's a subset process, which is deciding whether it's in or out, and then there's a permutation problem, which is actually rearranging them, right? That picking the C, and the R, and the I, and the M, and then kind of rearranging them.

So in fact it has kind of a little bit of both, right? Which is what sequence of letters can I extract and then rearrange to make a word – would end up using kind of elements of both of those kinds of recursions sort of mixed together so that when you look at a lot of recursion problems, they end up actually just mapping down to one or the other, or maybe a little bit of both.

And so feeling very comfortable with those problems, how you turn them into a recursive backtracker, and how you can recognize, right, their roots inside these other problems, it's kind of a really good first step to becoming kind of a journeyman, in a way, or recursion. So just whenever you see a new problem you start thinking, okay, is it permutations? Is it subset? Or is it something totally new?

It's probably much more likely to be the first two then the third, and so trying to use the ones that you feel really comfortable with to kind of branch out to solve similar problems, right, where the domain is a little different – the structure of the code is still very much the same. All right. I've got 10 minutes to teach you about pointers. This should be no problem. Let's go see what that guy back there's doing.

Oh, look at that, 38,000 assignments, still has not given up on that S is zero. It's a trooper. Okay. So this is definitely gonna be your first introduction on kind of the first day of talking about this. This is not the only day. So don't get to worried about me trying to do it in 10 minutes. What I'm gonna do is give you the kind of – a little bit of the basics today, and we're gonna follow up with some more stuff tomorrow where we start kind of making use of them and doing something kind of interesting with them.

So there is this notion in C++, which is inherited from the C language in fact, of using a variable type called a pointer as part of the things that you operate on. Okay. People tend to have a lot of trepidation. Just the word pointer causes a little bit of fear, kind of to immediately rise in a new programmer. But hopefully we can demystify a little bit, and then also get a little bit of understanding of why there are reasons to be a little bit wary of working with pointers.

A pointer is actually an address. So here's a couple things we have to kind of think about a little bit how the machine works to have a vision of what's going on here, that when you declare variables, you know, here I am in main. And I declare, you know, int Num, and string S, that there is space set aside as part of the working memory of this program that is gonna record whatever I assign to Num or S.

So if I say Num equals 45, what is that? S equals hello, that there has to be memory that's being used to hold onto that. And as you declare variables, right, more of the space is set aside, and, you know, when you initialize it, when you read to it, when you write to it, right, that space is being accessed and updated as you go through. When you open a new scope, right, new variables come in a scope.

When you close that scope, really that function, that space gets de-allocated. All this happens on your behalf without you actually taking a really active role in it, but in fact you do have to have a little bit of understanding of that to kind of idea of what a pointer's about, is that there is this just big sequence of data, starting from an address zero.

You can think of these things as actually having a little number on them. So maybe this is number 1,000, and this is number 1,004. In this case, assuming that we're numbering by the byte, which is the smallest chunk of memory we can talk about, and that an integer requires four of those bytes to store all the different patterns for different kinds of numbers so that the bytes from 1,000 to 1,003 are reserved for holding onto the information about 45.

And then from 1,000 forward – to however big the string is, which we don't even really know how big it is, so we'll kind of leave it a little bit vague here – is gonna be set aside for storing information about that string. Okay. So usually it's of interest to the compiler, right, to know where things are being stored and what's going on. But it also turns out to be somewhat useful for us to be able to talk about something by address.

Instead of saying the variable whose name is Num, I can talk about the variable who lives at location 1,000. And say there's an integer, and I can point to it, or refer to it by saying go look at memory address 1,000, and read or write the contents there that are of integer type. Okay. It seems a little bit odd at first. You're like, well I already have other ways to get to Num. Why is it I want to go out of my way to find another different mechanism that can get me access to that same variable?

And we're gonna see that actually there's a lot of flexibility that is being bought by us adding this to our feature set. We can say, well, I could actually have one copy of

something. So imagine that you have a system, like, an access system where you have student records, and they're enrolled in a bunch of different courses, that your enrolled in four, five different courses, that when it comes time for a class to know who's in their list, it might be handy for them, instead of actually having a copy of that student's information to have a pointer to one student record, and have a lot of different placers where there are additional pointers taken out to that same location and says, go look at the student who's at address 1,000.

I have the students at address 1,000, at 1,026, at, you know, 1,035, whatever those places are, and that that's one way I can talk about what students I have, and those same student addresses might show up in someone else's class list in math 51, or physics, you know, 43 or whatever, and that we are all referring to one copy of the student data without duplicating it.

So this idea of being able to refer to stuff not just by name, but by where it lives, is gonna give us some flexibility in terms of how we build things out of that. So let me kind of show you a little bit of the basic operations, and then – and I'll talk a little bit along the way about this thing about why they're scary because using, you know, memory access as your way to get to something is a little bit more error prone and a little bit harder to deal with than some of the more – other operations we have.

So what I'm gonna show you here is a little piece of code. It shows some simple use of pointers. All right. So I'm gonna draw some of the variables that are going on here. This is main and it declared an integer whose name is Num, so I draw a box for that, and it declares two pointer variables. So the addition of the asterisk by the name there says that P and Q are pointers to integers.

So P and Q themselves are variables that live in the stack. So all the local variables we say live in the stack. They are automatically allocated and de-allocated when you enter a routine. The space for them comes into being. When you leave it, it goes away, and that P and Q are designed not to hold integers themselves. They don't hold numbers, but they hold the address of a number somewhere else.

And so the first thing I did with P there was to assign it the address of a new integer variable that came out of the heap. So the new operator is like the new operator in Java. It takes the thing you want to create one of, it makes one of those in the heap, and it returns to you its address.

In that way it works exactly like in Java. So, in fact, Java actually has pointers despite what anybody ever told you, that the way you create objects and you use new to access them and stuff like that is exactly the same as it is in C++ as it is pointers behind the scene. So I say P gets the value of a new integer. This memory over here is called the heap. So this is not to confuse you with the idea of the stack ADT. We've been using the [inaudible] but it does kind of – it helps you to remember that the stack actually kind of, like, by the virtue of the way function calls get made, main calls A, which calls B, which calls C, that that memory actually kind of is laid out like a stack.

The heap is just this unordered crazy pile of stuff. I ask for new integer, so this might be address 1,000. This might be address 1,004. This might be address 1,008. Typically the stack variables are laid out right next to each other. This could be, like, address 32,016 or something over here. Some other larger address. So I've assigned P to be that thing. So what I actually gotten written into P's box is behind the scenes there really is the number, you know, the address, 32,016.

What I'm gonna draw is an arrow to remind myself that what it is is it's a location of an integer stored elsewhere. The de-reference operator, which is the first thing we see on this next line, says to follow P and assign it a 10. So this is taking that address that's in P, using it as a location to go look up something, and it says, and go write to that location in address 32,016 the number 10. And I said Q equals no int.

So Q gets an [inaudible] maybe this is at 23,496. Some other place out there. And so that's actually kind of what's being stored here, 23,496. And then I did this thing where I assigned from D referencing P to get an integer, and assign that onto what Q is that has the effect of kind of following P, reading its 10, and then writing it over here. So copying the integers at the end of those two pointers to make them point to the same value.

So they point to two different locations, but those locations hold the same integer value that is different than the next line. Without the stars, where I said Q equals P, without the stars on it, it's saying take the address that's in P, and assign it to Q, causing Q and P now to be aliases for the same location. So now I have two different ways of getting to that same piece of memory, either by reaching through P and de-referencing, or reaching through Q and de-referencing it – both of them are reading and writing to that same location in the heap, where the 10 is, and then this one down here's no longer accessible. I sort of lost track of it when I overwrote Q with the copy of P.

When I am done in C++ it is my job to delete things that are coming out of the heap. Yes, it should. I'll take that one at 32,016. Whereas in Java, when you say new, and then you stop using something, it figures it out and it does what's called garbage collection to kind of clean up behind you. In C++, things that you new out of the heap are your responsibility to delete.

So you delete something when you're done with it. If I'm no longer using this new integer I created in the heap, I say to delete P to allow that memory to be reclaimed. And so that causes this piece of memory to get marked as freed, or reusable, so that a subsequent new call can have that space again and use it for things. I will note that right now, the code as written has a little bit of an error in it because it says delete P.

And delete P says we'll follow out to 32 [inaudible] and mark it as available. The next thing I did was said delete Q. Well Q points to the same place P did. So in fact I'm saying take that piece of freed memory and mark it freed again. There is no real guarantee about whether that's gonna do something sensible, or whether it's just gonna ignore me.

One thing it could do is just say, well look, that memory's already freed, so you saying to free it twice is kind of stupid. On the other hand, it could still cause some more problems depending on how the heap allocater works, and there's no guarantee. So it becomes very [inaudible] on the programmer to be very careful about this matching, that if you make a new call, you make a delete call.

And if you already made a delete call, you don't make another one accidentally. So I really should have that line out of there. The piece of memory that Q originally pointed to, right, I no longer have any way to get to, and so I have no way of making a delete call to it and freeing it. And so this little piece of memory we call an orphan.

He's stranded out there in the heap, no way to get back to it, and C++ will not automatically reclaim it for us. So we have created a little bit of a mess for ourselves. If we did that a lot, right, we could end up clogging our heap filled with these orphans, and have to keep getting new memory because the old memory, right, was not being properly reclaimed. We're gonna talk more about this, so this is not the all – end all of pointers, but just a little bit to think about today. We'll come back on Wednesday, talk more about it, and then talk about how we can use this to build linked lists, and that will be fun times for all.

[End of audio]

Duration: 52 minutes

ProgrammingAbstractions-Lecture12

Instructor (Julie Zelenski):Hey there. Okay, we'll start with talking about just where we're at in terms of material, right, as we're gonna keep talking about pointers, which we just got started with on Monday, and go on to use them to do some stuff with recursive data in the form of a link list.

The reading is a little bit scattered at this point, mostly. Handout 21 is probably the best resource for the material I'm talking about today, and then once we work through the link list material we'll go back to the text and start looking at chapter 7 and fall by chapter 8 in order.

We'll spend a couple days on chapter 7 though, because there's a lot to talk about there in terms of algorithms and analysis and sorting and things like that.

[Inaudible] today, and some [inaudible] going out, so first off let's ask a little bit of joy pull, how much joy there was in the recursion assignment. Let me first see if anybody wants to own up to having spent less than 10 hours total getting all those problems working.

All right, that's good. That's a good sign, that you must really kind of have your recursion head on. Ten to 15 hours? Another big, healthy chunk there. Fifteen to 20? Oh, a little bit of that. And then more than 20?

And so in terms of sort of hours spent per line written, this probably had sort of one of the highest ratios you've seen, right? You know, you work on it 10 hours and you write four lines. The output field's a little bit sparse relative to the effort that went into writing it.

That's a very sort of symptomatic of kind of what recursion is like, right? It is an elegant and direct way of solving the problem, but a lot of the work gets spent in just figuring out what that formulation is, and once you have the formulation, getting it down in code is not the time-consuming part. There's some details to handle and things like that.

But your bigger time spent certainly should be conceptual, kind of figuring out how to break it down and get recursion to work for you.

The next assignment going out is Boggle. Boggle is a legacy of the CS106 program. I assigned Boggle for the very first time in 1994, so we have actually had Boggle kind of ongoing, on and off, for almost 15 years, and every time we've tried to replace it with another kind of big recursion program, we're always dissatisfied and we come back to Boggle, because we think it's just one of the best programs to have at this point in the quarter. So you're seeing kind of a long tradition here.

I'll tell you what I think is really neat about Boggle to kind of get you psyched. We give you a little bit more time to work on it, because actually this is your first big, complete

program, right? Not a bunch of little parts, not a little problem to solve here, one big thing to do, which is to write a program that plays Boggle against a human.

Lets the human find words, lets the computer find words, kind of head-to-head. It uses recursion in two ways, both in doing searches for the human as well as doing searches for the computer, so it actually has a chance to kind of flex those recursion muscles that you've been working on this week.

It uses some of the ADTs we've seen so far, the grid and the lexicon come into play; potentially some of the others as well. A lot of different pieces to kind of bring together, you'll do some interacting with the user. There's a little bit of drawing. Although we did most of the drawing code for you, you have to kind of interact with our library to get those things up on the screen.

And a lot of pieces to kind of bring together. So at this point we're at a really good milestone for, you know, you're feeling good, you've got a handle, it kind of touches on everything we have talked about so far, right, shows up in this program, and it's your first kind of big, complete kind of put the whole thing together, and that means it's a really good opportunity to also be very conscious of style.

On the recursion problems, certain things that are smaller, the opportunities to distinguish good stuff from bad stuff aren't that broad. But in the context of a whole program, there's actually a lot of choices you can make that can turn out well or poorly, depending on how much time and care you're putting into those early decisions, right, can really make a big difference in the long run.

So this is one where I think up front thinking and planning and kind of doing a good job with the style really pays off in terms of getting an easier ride all the way to a good solution.

It's also really cool when you're done because you write this program that, in fact, right, will kick your butt every time you play it, and I think that's actually a real important milestone for you, is to write a program – at some point you have to write programs that do things you can't do yourself, right, that you can't do by hand, to realize that you're capable of harnessing this power to write things, right, that extend beyond the capabilities of you yourself I think is a real significant accomplishment, and one that you can take a lot of pride in.

And so when you get to where you have a working Boggle and you're playing it and it's beating you every time, you don't have to feel bad about your vocabulary, what you get to feel good about is your programming skills in having done that.

And so every year I always get a couple of people who give me screenshots of them actually beating the computer. Because it's so rare, it's worthy of recording to prove.

We used to keep a machine, actually, in my office a while ago that was just running a game of Boggle and my officemate and I would occasionally go over and look at it and type in words, right? And then we'd just let it sit there. Like, you can take as long as you want, that's the way – the game is stacked for you. As patient as you are, you can win if you're just willing to sit and work on it for a very long time.

So we'd leave it there for days. Students would come in and help us, we'd keep finding one word at a time, one word at a time, you know. After we'd had it running for a week or two we'd finally say okay, we found all the words, there are no words left. And then we would let the computer's turn go and it would find all the words we had not found, and then we would feel dumb. But we would keep trying, and every now and then we would win.

A little note about paper copies, right, is that we are asking you to turn in both an electronic copy and a paper copy. The paper copy serves as the place where we're writing a lot of the comments and working with you on the IGs. It is important that you turn in both, and the reports back from the section leaders is that there's been a little bit of delinquency on that. That pretty much everybody's getting an e-submit in, but they have been a little bit sloppy about following through on the paper copy.

We have not been making a big deal out of it, but we do plan on being a little bit tighter about that in the future. So if you have been taking advantage of us and turning them in kind of when and where you feel like it or not at all, do be aware that we will start enforcing a late-day policy, that if the paper copy comes in late, right, then we will be calling it late, even if the electronic submit was made on time.

So there really is no reason to – you know, once you've got the e-submit, just print it right out, bring it on in, and all is well.

All right, anything else kind of on the administration angle that I could – okay.

Now I'm going to show you a movie that's going to make everything about pointers clear, okay? All right, speaking of that officemate and I who I used to play Boggle with, this is Nick Parlante, who actually is now my next-door neighbor in the Gates building. And he in 1999, with a little too much time on his hands, did a little experiment in Claymation that will reveal all the mysteries of pointers in three minutes. Here we go. [Video]

Hey, Binky, wake up, it's time for pointer fun. [Video]

What's that? Learn about pointers? Oh, goody. [Video]

Well, to get started, I guess we're gonna need a couple pointers. [Video]

Okay. This code allocates two pointers, which can point to integers. [Video]

Okay, well, I see the two pointers, but they don't seem to be pointing to anything. [Video]

That's right. Initially, pointers don't point to anything. The things they point to are called pointees, and setting them up is a separate step. [Video]

Oh, right, right, I knew that. The pointees are separate. Er, so how do you allocate a pointee? [Video]

Okay, well, this code allocates a new integer pointee and this part sets X to point to it. [Video]

Hey, that looks better. So, make it do something. [Video]

Okay. I'll dereference the pointer X to store the number 42 into its pointee. For this trick, I'll need my magic wand of dereferencing. [Video]

Your magic wand of dereferencing? That's great. [Video]

This is what the code looks like. I'll just set up the number, and – [Video]

Hey, look, there it goes. So doing a dereference on X follows the arrow to access its pointee – in this case, to store 42 in there. Hey, try using it to store the number 13 to the other pointer, Y. [Video]

Okay. I'll just go over here to Y and get the number 13 set up, and then take the round of dereferencing and just – ooh. [Video]

Oh, hey, that didn't work. Say, Binky, I don't think dereferencing Y is a good idea, because, you know, setting up the pointee is a separate step, and I don't think we ever did it. [Video]

Hm, good point. [Video]

Yeah, I mean, we allocated the pointer Y but we never set it to point to a pointee. [Video]

Hm, very observant. [Video]

Hey, you're looking good there, Binky. Can you fix it so that Y points to the same pointee as X? [Video]

Sure, I'll use my magic wand of pointer assignment. [Video]

Is that gonna be a problem like before? [Video]

No, this doesn't touch the pointees, it just changes one pointer to point to the same thing as another. [Video]

Oh, I see. Now Y points to the same place as X. So wait, now Y is fixed, it has a pointee. So you can try the wand of dereferencing again to send the 13 over. [Video]

Uh, okay. Here it goes. [Video]

Hey, look at that. Now dereferencing works on Y. And because the pointers are sharing that one pointee, they both see the 13. [Video]

Yeah, sharing – uh, whatever. So are we gonna switch places now? [Video]

Oh, look, we're out of time. Just remember the three pointer rules. Number 1, the basic structure is that you have a pointer, and it points over to a pointee. But the pointer and pointee are separate, and the common error is to set up a pointer but to forget to give it a pointee.

Number 2, pointer dereferencing starts at the pointer and follows its arrow over to access its pointee. As we all know, this only works if there is a pointee, which kind of gets back to rule number 1.

Number 3, pointer assignment takes one pointer and changes it to point to the same pointee as another pointer. So after the assignment, the two pointers will point to the same pointee. Sometimes that's called sharing.

And that's all there is to it, really. Bye-bye, now.

Instructor (Julie Zelenski):There you go. Whoo. So if you ever wanna meet Nick Parlante, who lives next door to me in the Gates building, you can tell him that you enjoyed his clay movie and that all your mystery of pointers has been cleared up.

All right, let me – let me show you some code. I'm going to keep talking about the pointer idea, draw some pictures to try to get a visualization of what's going on, and then I'm going to go on to show you some of the things you do with pointers. What are pointers good for, right? Right now, it seems like it's kind of an obscure way to get at things you already know how to do. I'm going to show you some of the things that then it enables you to do that you couldn't do if you didn't have the pointer around.

So I've got a little bit of code, right, that declares some variables, and I'm gonna draw the same picture I had at the end, right, that what is the stack frame for main look like? It's got an integer num, and it has two pointer variables, P and Q, and those things actually all live in the stack.

So stack variables are those that you declare kind of as you open a scope, as you enter a function, things like that. Space for them is automatically allocated and de-allocated. You never see explicit news or deletes on those that are there. Those pointers, as they are set up here, have not been initialized.

So just like num has some junk contents, right, so if I have just a declared numb and then I do cout, num, endl, then I'll get some junk number. It might be zero, it could be 6,452, it could be negative 75. We don't know what it is, right, but accessing it, right, is technically legal, right? It won't complain to you about it, right, but it's unlikely to produce any valid result.

Similarly with pointers, if I say P and Q, right, without saying what they're assigned to, then these have junk addresses in them – junk numbers. Now this turns out to be a little bit more dangerous, the use of an uninitialized pointer relative than an uninitialized variable, that if I were just to look at that number, that's unlikely to get me into trouble.

If I try to follow that address as though it were a good address, that's when you end up with Binky's head getting blown off. If I've done nothing to P and Q and I say star P equals 42, it takes this junk contents, which is, like, some address to somewhere, and follows it and tries to write a 42.

There's no guarantee, for example, that that number, that address there is valid, that it actually makes sense, that it actually even kind of exists in terms of the process space so far. It's kind of like calling a phone number just by dialing random digits on the phone.

It's, like, more likely than not, right, you're gonna get the, you know, that number's been disconnected. It might be that you'll randomly call somebody and start having a phone conversation with them. The same with this 42. It's like it might be that it'll write somewhere and start storing a 42, but on top of something that wasn't intended to have that kind of adjustment made to it.

So it's a very dangerous operation, right, that can lead to strange results, immediate crashes, you know, other strange results later on, when you didn't plan on it, other things changing that you weren't expecting because of this improper access.

So P and Q, right, sort of that – if I see P equals new int, then over here in the heap, so stack here, heap over here, P points to a new integer variable stored out in the heap, and now that's star P equals 10.

So that word pointee, that's a Nick made-up word. No – you won't hear that pretty much anywhere other than the Binky pointer video, and sometimes I use it too, because I think it's a cute word. But it is not quite the official term for that, so P pointing to this integer out in the heap. And then I have these two lines that I talked about last time that I want to review what's going on that star P, right, it's a star Q, any time you see the pointer with that star on it, right, that's applying the dereference.

And so it's saying we're not talking about manipulating the pointer itself, but what's at the other end of it. So reaching out to the integer. So the way P is declared there, I want to say P is a pointer to an integer, then the type of the expression star P is integer type. It refers to the integer that it is pointing to, which in this case is out in the heap.

And so if I said Q equals new int, it gets a space out in the heap. And then when I say star Q equals star P, that says right to the integer out here, having read an integer from there to copy. So it copies the integer value that P is pointing to on top of the integer value Q is pointing to. So copying the integers.

The next line, where the stars are not present, where it just says Q equals P, that is doing just pointer assignment. In this case, it's manipulating the pointers themselves, not what they point – the integers that they're pointing to. So when I say Q equals P, it effectively copies the address that P has.

So P is pointing to the address, you know, 30,012, and so what's really at sort of the lowest level is actually written this number here. Then it writes that same number in this box, which causes Q to point to the same place as P now. And so changing star P, changing star Q, actually both of them actually are sharing or aliasing, and so there's only one integer that now we have two different ways of getting to via these two different pointers.

So here's a little trick about C++, right, that's unique to those of you who've come from the Java world, is that it is your responsibility to deallocate things that come out of the heap. So whereas on the stack memory is allocated and deallocated automatically, everything in the heap is explicitly allocated and explicitly deallocated.

If I say new, it made some space and set it aside and reserved it for my purpose in the heap, great. When I'm done with it, when I no longer need that piece of memory, I've discarded it, I've, you know, removed it, that student graduated, whatever it is that causes me to no longer need it, then it is the responsibility of the programmer to issue the delete call using the delete operator on the pointer to say, follow this pointer and take the space that it's currently pointing to and mark it as reclaimable – no longer in use, ready to be vacated for someone else's purposes.

So when I say delete P, it causes the heap manager to say, oh, okay, this address is now free or available. As a consequence of that, it may or may not actually do anything right away with that piece of memory. It might be that what it just did was mark it as available, kind of it's sort of like having a list of vacancies in an apartment building.

You might say okay, well, I'm moving out of apartment 100. You might say oh, okay, well, apartment 100 is empty. It may go in and clean the apartment, it might go in and kind of write something over it, it may not. There's actually no guarantees there. But it does mean that now if somebody comes in and wants an apartment, right, we could give them that one. So if somebody asks for a new integer, we might give them that address, 30,012, because we no longer have it marked as being in use.

So if you were to delete something and then start to use it again, so if I deleted P and then tried to do this – star P, you know, equals 100, I'm asking for trouble. That piece of memory has been deallocated, it might be in use somewhere else. At this point, I don't

own it anymore, I don't have that reservation. It's kind of like using your keys to get back into that apartment that you've moved out of.

It's like well, someday, somebody else's stuff is going to be in there, right? And you are squatting, right? You can be arrested for that. Well, in C++, right, you can receive program crashes and other kind of mystical behavior from that, and so it is something we have to be a little careful about.

When we're done we say delete, but not before, and that once we have said delete we don't reference that piece of memory after it's been deleted.

I put a little note here. I said, well, delete Q. So here's a thing to note, is that we delete things that we allocated new on. It is not the case that we delete every pointer, though. If we have five pointers that point to one piece of memory, if I delete it, no matter which pointer, you know, I used when I made the delete call, they all hold the same address, and I deleted what was at that address.

So if I've deleted the address 30,012, no matter how many other pointers hold that same address and point to that same place, I do not make multiple delete calls. So in this case, if I did delete P and delete Q where they're currently aliased, right, I would be trying to delete a piece of memory that's already deleted.

Again, no guarantees about whether this will produce an immediate bad effect or some later bad effect, or maybe just turn into a no-op, but no matter what, it's not a good practice to get into of accidentally deleting things more than once.

So a little bit more management on our part, right, than we're used to in Java, and then that last little bit of code I showed at the bottom, right, was just the use of Null, capital N-U-L-L, which is the C++ equivalent of the Java lowercase null, it's the zero pointer, and it's used as the sentinel value. Typically when you have a pointer that you know doesn't point anywhere, you use that to say, well, here it is, I'm going to initialize it to Null so I know that that's the known sentinel value that says there's no current address that it's holding.

So very much some of the things you've already seen in Java, right, this idea of having declaring things, and calling new to get new things that are out in the heap. The delete step is certainly new. It's one we'll work our way up to. To be honest, to be fair, if you just had no delete calls in your program, the consequence would be that pieces of memory like this one that I allocated under Q and then later lost track of would be orphaned. And so we'd end up having a bunch of memory that we had set aside and reserved for our use that we weren't currently using.

If we kept doing that over a long period of time, we'd eventually kind of clog the heap with a bunch of waste that would not be reclaimable and it could slow the program down and even in an extreme case, right, could cause your machine to have significant performance problems.

For the shape of programs we write and the size of them and the amount of running time they are, not having deletes in them is actually kind of not a particularly tragic result. So often, right, we'll encourage you to write programs without delete in them where you make news, and work your way up to putting deletes in, because in fact, right, they're tricky to get right and if you get them wrong, the consequences are sort of more deadly than just having the orphans get filled up in your heap.

All right, so any questions on kind of the basic little set up here?

Student: So the pointee or whatever, of Q original one?

Instructor (Julie Zelenski):Yeah?

Student: There's no way to delete it anyway?

Instructor (Julie Zelenski):There's no way to get – I lost its address, right? I didn't store it anywhere, so it gave me this address. Maybe this is a, you know, address, 48,012, that it gave me that address, I wrote it down, right, and then I overwrote it with something else. And so I no longer have it. So unless I put it somewhere, right, I have no way to know where that piece of memory is.

Student:[Inaudible] Why wouldn't it be, like, delete star P?

Instructor (Julie Zelenski):Well, you know, it's interesting, it kind of feels a little bit more like it's delete star this, right? It's like delete was at the other end. But it just happens to be that the operator takes the address itself, not the number that's at the other end or the string or whatever it's a pointer to, so, the operator.

So it takes a pointer and it says take the contents at that address and mark it as available for someone else's use.

Student: Is P [inaudible]?

Student: So P still holds the same value. That's a great question, right? So P had 30,012. I said delete 30,012. The heap manager took that request and kind of marketed it, but P still holds that address. And so if I did a star P equals 100, it tries to right back there. So in fact the delete operation doesn't change P. So it still holds the same address.

So to be really careful about it I might want to say if I set it to delete P, I could set P equal to Null right after it, kind of as a safety catch for myself, to remind myself that whatever number it used to have is no longer valid. That address, right, is no longer mine to reference.

So as a habit, some programmers do do that. Every time they make a delete call, the next thing they do is set that pointer to Null so that they don't actually even have a chance of reusing that address that no longer is valid.

So this kind of control actually is really valuable. At first glance it just seems like it's like extra work. It's like well now not only do I have to allocate, I also have to deallocate. But that gives you actually very good control of exactly when it got allocated and exactly when it got deallocated, and that means you can control the memory usage of your program very precisely in a way that in Java, Java doesn't actually let you get involved in this.

You can allocate things, but it decides when and where it's going to start doing deallocation. It may wait until it thinks that there's a need to do this what's called garbage collection, it may do it a little bit all the time, it may do a big bunch at one time. And it doesn't give you control in the way that C++ really just leaves it up to you.

You decide when you needed something and when you're done with it, which does actually, for professional programmers, considered a real advantage in very precisely dictating about how memory gets used, and when and where.

So a couple things that I just – things I wanted to remind you before we went on to do some other things, right, is that pointers are distinguished by the type of the pointee. That you can have a pointer to a double, a pointer to a string, a pointer to a scanner. You know, any of the types you have seen, right, there is a corresponding star of that type that is a pointer to one of those things. There are, in fact, right, pointers to pointers.

We're not going to see that too commonly in this quarter, but there are, for example, int double stars, which is a pointer to something which is a pointer to an integer. That has kind of two arrows that lead to an integer, finally.

All of these pointers, right, are distinguishable types that if, you know, two things that both point to double are the same type of pointer, but a pointer to a double is not the same thing as a pointer to an integer, which is not the same thing as a pointer to a string. So actually, the type system considers these different kinds of pointers to be different.

In truth, right, the way they're being managed behind the scenes is they are all just addresses, but it is important to know that what's at the end of that address is an int as opposed to what being at that address as a string. And the compiler does strongly enforce this idea that if you told me what's at that address is a string, then it wouldn't be appropriate to suddenly start treating what's at that address as though it were an integer. So it actually doesn't allow you to kind of mix those types up.

Now pointers are uninitialized until they're assigned. When you declare a pointer, right, you just get a junk address dereferencing it either to read or to write some of that contents, right, is just asking for bad trouble.

The consequences are not that predictable, so one of the things I had tried to say last time was one of the reasons I think pointers are considered a little bit scary is because there are certain opportunities to make mistakes here, and those mistakes have pretty drastic consequences.

Using an uninitialized integer is not that terrible. You know, if you're trying to sum the numbers in an array and you forget to initialize your sum, you get a sum that's junky but it doesn't crash your program, right? It just uses that number as the starting value and sums from there.

It produces results you kind of figure out what happened, but it doesn't crash. Many times, right, the use of a pointer that hasn't been initialized will cause an immediate crash, or it could cause sort of some later consequence to crash, right, that would make it even more mystical to try to sort out what happened.

So it's partly because those bugs are – have pretty serious consequences and kind of sometimes difficult to understand consequences.

So we do dynamic allocation via new, so dynamic meaning that just we get to choose what we're allocating, when and where, at a runtime situation saying how much memory we want and when we want it, using new.

And then when we're done with it we do a manual deallocation via delete. If we forget to delete, we orphan memory. As I said, an important thing to be conscious of but not in the early stages something to take too heavily. And then accessing any deleted memory, right, it has unpredictable consequences, similarly to using uninitialized pointers, right, going back to addresses that have been marked freed is kind of like going back to your apartment after you don't have a lease on it anymore, right, like you put your stuff in there, it might get stolen, right? Now a god idea.

So now let me tell you a little bit about how pointers work with arrays. We have not talked much about the C++ built-in raw array, that I've encouraged you to use vector where you would have used array because a vector is actually just much easier to deal with and it has a lot of convenience and safety features built into it.

But as part of kind of understanding how the basics of the language work, it's worth knowing that pointers and arrays have a little relationship that – in how they're expressed in C++.

So this piece of code that I have here, right, declares an int pointer called ARR. And it immediately initializes it to a new integer array in the heap that has 10 slots. I didn't quite draw 10, but that's a good idea of it.

So the new operator, in addition to taking a single type – making a single string or an int or a double out in the heap, also has an alternate form where you add the square brackets on it and then you say, inside the square brackets, how large an array of those things you would like.

So this is creating an array of integers 10 numbers long out in the heap, which array is going to point to. Having done that, right, array now has slots zero through nine available to it in which it can write and do things. And so the next thing it does is go through and

use the bracket notation that's standard for arrays to access in turn elements zero through nine index and assign them each their index number. It's missing two numbers that I couldn't fit there.

And so that is the syntax in C++ for creating a new dynamic array out in the heap, accessing it either to read or to write to those contents. It looks just like the vector there. And then when I'm done with it, because it came out of the heap it was dynamically allocated manually by me, it's my job to manually delete it.

And there's a slight variation on delete that matches the use of the new int bracket, the delete bracket. So if I made an array of things out there, then I need to use the delete array form of it, so the delete bracket as opposed to the standard delete.

If I use just the standard delete, it's like it thinks there's only one integer at the end of this, and it only kind of tracks that space. So the delete bracket says there's whole arrays worth of integers, make sure all of them get reclaimed.

Now again, like most memory errors, right, if you do the wrong thing it won't necessarily have really clear – it's not like it provides an error message and stops and says you used the wrong form of delete, it's more likely to actually kind of just misunderstand your intentions and kind of blunder on, producing a little bit unpredictable results from there.

So raw arrays in general are trouble. We are definitely gonna use them a little bit later, in fact to build things like vector and stack and queue, right, we need to actually have this facility, right. The [inaudible] facility is what those tings are layered on top of. But as a client, I'm going to say that traditionally, right, that things that you would use an array for, it's just much better practice to use a vector for.

So I'll mention sort of why it is that, you know, arrays are trouble, right? Well, they're mainly allocated and deallocated, so that means you have to make the new calls, you have to make the delete calls. So forgetting one or both of those, right, has pretty serious consequences. Arrays don't know their length. That once I have created this array and it points to these 10 members, there is no mechanism to ask the array, you know, array dot length or tell me your length – it doesn't know.

Internally, it may or may not have some housekeeping that's tracking it, but it's not exposed to you as a client. So it will be your job, if you are creating a manual array, to also be tracking its size. And so everywhere you were trying to use that array you'd also need to know well, here's where the address of that array, its location in the heap, and here's how many members go. You'll have to kind of just track those two things around and pass them in tandem.

There is no balance checking. If I access the 12th member of this, it actually just uses a very simple calculation, which says well, here's where the array starts in memory. All arrays are laid out contiguously. If you ask me to find the 12th one, then I just kind of

walk down to where the 12th spot should be, even if it's not really mine to own, and I will read or write to that location as you ask me.

I can ask for the 100th member of the 1,000th member, the 6 millionth member, and all the calculations work off of well, here's where the array starts. Where would the 6 millionth member be if it were present? That piece of memory, if it's not allocated to this array and it's not really part of its proper bounds will receive no error checking, no nice out-of-bounds message.

It will just kind of do the calculation to figure out what place in memory would be referred to and read or write to those contents, causing kind of unpredictable, weird behaviors, right, that would be hard to figure out.

You can't easily change its size once it's allocated, so if I've done a new int bracket 10, I have a 10-member array. If I later want to put 20 things in there, then there is no way to take this piece of memory and stretch it to say, well, ad some space in the back. What I will need to do is allocate an array that's twice as long, copy over all the numbers I already have, and then, you know, delete the old one and update my pointer to point here.

So the only mechanism for changing the size of an array, once it's been allocated is to really go through this process of creating a new array of the size you want, copying over what you wanted to retain, deleting the old one, updating your pointer, which, as you can imagine, gets to be just a bit tedious.

So you can certainly use the raw array to do things, and we will definitely have to build things on it later, but once you have vector around, it provides so many of these things, right, in such a nicer form that dealing with the raw array appears too troublesome and too little value, right, to do so.

So vector does use array behind the scenes, but it hides these issues, right. It does track its size, it does do balance checking, it does grow and shrink on demand, exactly by doing the kind of things that I said were kind of a little bit tedious to do, it does on your behalf. So you've just sort of seen the picture. Your hand's up.

Student: You said that the type of pointer determines what it's pointing to. So why doesn't an array have, like, an [inaudible]?

Instructor (Julie Zelenski):That's a great question. So it turns out that there is actually – maybe there's one example where I was being a little bit disingenuous. It turns out a pointer to an integer and a pointer to a sequence of integers are considered the same in C++. So it actually doesn't distinguish the idea that you gave it an address and you're telling it what kind of data's at that address.

You are not telling it how much of those are there, how many, right? You're saying there's at least an integer there; there could be a whole sequence. So in fact it considers a pointer to an integer and a pointer to an array of integers to be type compatible.

There are certain quirkinesses about that, but you can sort of see what – at least from a type system point of view, that actually kind of does make sense, right? They both are saying that well, we have these two addresses. What's at the other end of them? They're our integers. And how many are there? It turns out we're not – we don't track for you, so the fact that there's 10, the fact that there's one, is up to you to know.

Way over here.

Student: Why don't you have to dereference array when you assign it?

Instructor (Julie Zelenski):So that is also an excellent question. It turns out there's a kind of a dereference sort of hidden in the bracket operator that the bracket operator has, behind the scenes, is actually doing a calculation that says take this address array, add to it sort of I slots' worth, and then dereference that.

So in fact there's kind of a star hidden in there, and it's just part of what the bracket operator does on an array. So you are correct, there obviously is a dereference happening in there, and it is sort of hidden in the brackets themselves is all I can say.

Student: So [inaudible] when you just declare an array [inaudible].

Instructor (Julie Zelenski):Yeah, so if you just make a – yeah, so all that – it's basically the same deal, it's just where did the memory come from, right? So if you say int bracket 10, it's on the stack. If you say new in 10 it got [inaudible] heap. So it's just a matter of does the memory live over here and it was kind of automatically allocated or deallocated by entering and exiting, or is it explicitly allocated with new and delete over in the Heap?

The heap tends to give you that flexibility of resizing it and moving it and stuff like that in a way that the stack doesn't, so that's kind of the preferred place to usually get an array from.

So mostly I'm telling you this because we are going to come back to that one, but I just – it feels like it kind of fits with kind of getting a picture about how memory, right – C++ is very much exposing to you the sort of low-level where it puts memory and how it accesses things, right, that pointers are giving you that direct manipulation for.

One of the more common uses of pointers is this idea of sharing. So if I were designing the Access database and I have this record for a student with their first name and their last name and their address and their phone number and probably a bunch more data – their student ID number and maybe their transcript so far.

All these things that model the information track for a particular student in there, I also have a bunch of courses – CS106b, Math 51, you know, Physics 41. And each of those courses, right, has a list of the enrolled students in it.

There is a student record that represents Michelle, right? And there's Michelle's record in the system. It could be that every class that Michelle isn't enrolled in copies her whole structure into it, so if I had declared this vector here as holding student T without the star, it would hold real student T structures.

Then every class Michelle was enrolled in would have a copy of her structure. I would say add to the students in this class Michelle's record, add Michelle's record to this class. Well, each of those, right, would make a full copy – her first name, her last name, her address, her phone number, you know, however big this record is, right, would be duplicated throughout the system in every class she was enrolled in.

That means that first of all, there's a lot of space that's being taken, a lot of redundancy in that system. It also means that we have a little bit of an update problem. If Michelle gets a new phone number and we'd like it to change across the system-wide, then I have to find every place where her record has been duplicated and updated to get the update to kind of flow through the whole system. So finding every course she was enrolled in and updating it in every place.

If instead there really is just one copy of Michelle's student record – I allocated it in the heap, I set it up, I put things into it, and then every course that she enrolls in I store the pointer to that same record, then there really is only one copy of the student T structure per student, but multiple pointers to it in different places.

When I need to update her phone number, there really is only one copy of her phone number. If I change the phone number in that one record, all the pointers are pointing at that one record, and all the updates effectively happened automatically, just changing in one place only.

So this is one of the most common needs for pointers, is just any kind of data structure where there needs to be some kind of sharing, where you have information, right, that's being tracked in different ways. You want to be able to look at your iTunes library by genre, you want to be able to look at it by album. You want to look at it by artist. That there really is only one record for each song in the database that's being managed there, but there's a lot of different views on that data, and those views could each use pointers to refer to the one copy, and then make for sharing of the data without that redundancy, and then this easy update of only being one place to do it.

So without pointers, right, you're kind of stuck, right? If you don't know about pointers, right, you end up kind of duplicating this or designing some more funky system where maybe you have a number here that tells you where to go look up them in some other place, you know. The pointer gives you a really clear means of just modeling there is a sharing relationship.

That's pretty cool. So what else can pointers be good for? Ah. Pointers are good for something. Now here's the thing you wanted to do. You wanted to take pointers, which inherently are a little bit confusing, and then take recursion, which we also know is pretty

confusing, and then put them together and make something that will make your head completely explode.

We're talking about recursive data. For some people, this is actually the thing that actually does make recursion suddenly make sense, though, is to see it – to see recursion echoed in a structural way as opposed to a code way.

So when I talk about recursion in the form of looking at a structure that itself kind of has embedded within it a smaller version. So I brought with you my physical embodiment of recursive data, which is my lovely Matryoshka doll. Ikea, $15. If you open it up, what's inside? Whoa, check that out – it's another Matryoshka doll. You open her up – hey, it's the – she's facing the other way. She was shy.

You open her up, there's gonna be something good, you can feel it. And then – little, tiny – this is the base case. She sits there. What you got there is like a doll which herself is a doll wrapped around another Matryoshka doll, which has kind of this self-referential kind of inside of it, right? There is another doll like the one you saw before, but a little bit smaller, a little bit tinier. Then at some point you get the one that actually is just so small that we're just done.

And so there are things like this in the real world, things like onions. You know, you peel off the outer layer of an onion, there's really kind of a smaller onion underneath, and eventually you get to that kind of onion core, that base case. We're going to look at this in terms of a structure that contains a pointer to the same structure we started with as a way of modeling something recursively.

Think about managing an address book. I've got a name and address and a phone for each of the people I wanna manage. I could certainly put these entries together in a vector and have kind of a vector of all the people in my address book. I'm going to choose a different way to organize this by virtue of making this structure have an additional field, a pointer to another entry.

So I've got three fields that represent, you know, Jason's entry in my address book, and then he has a pointer to more entries, or at least an entry, let's say – the entry that follows this one, which itself can be the name and address and phone number of Joel, let's say, with a pointer to another entry, and so on.

So each entry points to another entry. That's what makes for a recursive structure. It's kind of funny – I'm not even really done defining what the entry structure is, and I'm already referring to it. It's that feeling of kind of like the way a recursive function is written, where it makes a call back to itself before you're done completing your whole thought, saying that there is a pointer to another one of these.

And so a link list is a pointer to an element which itself has embedded within it a pointer to another element, and so on. And if I use that Null as my terminating sentinel at the end

to say that, well, Monte has no one following him, then I have this chain that I can follow from the front to the back to visit all the entries in my list.

Okay, this guy is called a link list. So it's a list because there's a, you know, collection of things that are there. They are linked together by virtue of pointers, and they have a very recursive formulation, if you think about it, right, that any individual entry is one entry in the address book followed by another list.

So the whole list has a thousands entries in it, but in fact if you kind of break it down recursively, you can say well, there's an entry in the front and it's followed by a smaller, shorter list – one shorter, in this case, 999 entries. And similarly, I can do that at every stage, which is to take each element and think about it as well, it's an element, but it points to a smaller link list behind it.

So I'm going to actually just do this code in the compiler. But it's all in the handout, as well as on the slides that I'll post. Just talk a little bit about doing some stuff with link lists.

So the first thing I'm gonna do is I have the entry, the name of the phone number with the pointer, and I wrote a little print entry that printed a single one. So we're going to see a lot of pointers in here, we're going to refer to all these elements by their pointers, and then we're going to allocate them out of the heap, kind of on demand, and wire them together by using the pointers to get our flexibility here.

The one routine that actually I'm gonna – I have pre-written here is one that will create a new entry in the heap based on information typed by the user at the console. So it prompts them to enter something. If they enter a return that's our clue that they have no more names. And otherwise, we make a new entry in the heap.

So here's our call to make a new entry [inaudible] out there, store the pointer to it here, and then this access – I should mention this syntax is a little bit – should be – may feel a little bit new to you – is that new one dot name, that what I'm trying to do here is take new one, dereference it, and write to the name field.

And so the syntax that kind of comes to mind here looks like this: take the new one pointer, dereference it to get to the entry structure, and then dot to get the name field out of it. And that is actually a sensible sort of construction.

The problem with it, if I leave it as-is, is that the precedence of these two operators is not quite what you'd expect. That actually, the dot has a higher precedence than the star, and so in fact the way the complier interprets this is as first take new one, act like it's some sort of structure that has a name field in it, and then retrieve that name field which you suspect to be some kind of pointer, which you can then dereference.

It's doing it exactly backwards. If we wanted to use that syntax as-is, we'd actually have to introduce these parens that said please first dereference new one. It is a pointer to an

entry. Get the entry structure that's referred to on the other end and then dot name to get the field name out of it. And that would work just fine. Just because that's a little bit awkward to write every time, there's actually an alternate operator in C++, the arrow operator, that combines the star and the dot behind it to go reach out there.

So this says following the new one's pointer, reach into the structure at the other end and access the name field. So I assign the name, I assign the phone, and then I set the next field to Null as just a good practice to say well, I've just created a little entry all by itself. Rather than leave that pointer initialized, let's make sure we set its Null so we know it's just a cell by itself and it's not connected to anything yet.

Okay. Let me just write a little code to test that. Um, I'll just call it N and then I'll say get new entry. And then I'm going to call print entry on it, so I'm just gonna do that to see that I can allocate an entry and then do something with it.

I can say it's Julie and my phone number is – and then it prints out my name and my phone number. Okay. Try it again, now I can say Jason, Jason's phone number is – oh, Jason has lots of digits in his phone number, okay. So we've got a little bit of stuff going here.

Now what I'm going to do is I'm going to build a link list. I'm gonna use this get new entry function to give me individual cells, and then I'm gonna wire them together and build a link list out of it.

So I'm gonna start by having my list be empty so the standard sentinel value, Null, is used when you – to indicate that you have no more cells or you've started with a totally empty list. I'm going to keep asking for new cells until they don't have any more, and so get new entry returns Null if they have indicated they have no more.

So I'll just go ahead and break out of the loop when that happens. Otherwise, right, I have a new entry that they just gave me, and I'm gonna wire it into my list.

Okay, so let's think a little bit about how the pointers are gonna work on this. So here's my list pointer, and it points to a cell, which points to a cell, you know, which points to a cell and so on, that there's information here, you know – this is Jason, this is Joel, this is Brittany, okay.

I get a new cell, it is Jake. Back from my get new entry. And I wanna attach Jake into my list to join it with the ones I have. So I've got an existing list and a new cell to add to it.

There are two obvious places to add the cell. Typically, when you have a list, it seems like probably putting it at one of the ends, given that I'm not trying to maintain any sorting order at this time, is gonna be the easiest place to put it.

So tell me, think about this. Should I may Jake's first or last? Is there any reason to prefer one to the other? Is there any advantage to going with one way versus the other one?

Why not? Yeah, why not first? What's good about first?

Student: Because when you're making a new entry you can probably more easily decide what the next pointer's going to point to as opposed to going all the way back through the list and going to Brittany and assigning that pointer to something.

Instructor (Julie Zelenski):Yeah, that is exactly true. So there's something a little bit quirky about link list in this respect, which is because it's a chain, right, where you have a pointer to the front and then subsequent pointers get your way down at the end [inaudible] it's very easy to get to the front of the list, but it's actually a little more work to get to the end.

If I wanted to, if I had a thousand entries in my address book and I wanted to add each entry to the end, then I'd have to kind of work my way down to the end to kind of do that work.

So it turns out it's actually just a little bit easier – requires less housekeeping, less work to [inaudible] if I just – I already have a pointer to the front. Why don't I just go ahead and put Jake on the very front. So prepend Jake.

So the processes for this are going to be I have Jake in here. Typically when I am putting a new cell into a link list, there's going to be two pointers that need to be adjusted to splice it in. The new cell needs to have its outgoing pointer, so Jake needs to be attached such that if you got to Jake, it would lead away to some other cells.

In this case, I'm gonna have Jake point to what previously was the front most cell. And then I need to wire the pointer into that cell, so, you know, Jake has to kind of exist in context, he has to have something coming in and something going out. The one that's gonna come in is gonna be the one that previously was the front most – the list head or the front note of the list is now gonna point to Jake.

So the two assignments I need to make to get Jake into there is assign Jake's next field to what was previously the front most cell, and then make the front most cell point to this new one that we just got back. So we wired out, we wired in, and now Jake is the new leader of the list.

Okay, take a look at that. Like a little bit of code, but doing something pretty important. This is Jake's next field getting sent to what was the previous front most cell, and then updating the front most cell to point to Jake so that Jake will become the new leader.

So if you think of it, you know, on the very first iteration, it's always good to make sure those [inaudible] cases are gonna work fine. If at the beginning [inaudible] there's no cells in the list so I start with an empty list, I get a new cell from here and then I set that new cell's next field to be the list that basically says set its next field to be null, okay.

That's fine, there are no existing cells to attach. And now make the head cell to the list point to this new cell. So that made us a singleton list that has one node in it whose next field is Null, and then on a subsequent call – so we come back in to put the next node in, we'll make a new one. Its next field will point to what was previously our singleton cell, and then we update the front most pointer to point to the new cell.

So it should prepend each one onto the list. So if I type the names in A, B, C that when I go to print them back out I should probably get them in the other order.

So let's take a look at this. Let's write build list here, and call it. And then I'm – right now it's just gonna print entry, which is going to print the very first one, and then I'm gonna change that to print all of them. So if I put Julie and some phone numbers, I put Jason and some phone numbers, I put Marcia and some phone numbers, and then I say I'm done, then Marcia was the first one in the list so when I said print entry, I printed just the first one right now because she was the last one I put on there.

It's operating a little bit like a stack – last in, first out. Then it put them on the front, so what was ever the last one entered is the one that is the leader of the list at this time.

Okay. Well, I don't have time to show you much more today, so we're gonna have to do some more of this on Friday. But we'll come back and we'll look at this to do a little bit more work and start thinking about how recursion is an interesting partner with this kind of data structure.

[End of Audio]

Duration: 51 minutes

ProgrammingAbstractions-Lecture13

Instructor (Julie Zelenski):Welcome to Friday. I keep telling you I'm gonna talk about linked list and then we don't ever get very far, but today really we are gonna talk about linked list and do some tracing and see some recursive work we can do with linked list and then I'll probably give you a little preview of the next topic, which is the introduction algorithm analysis in Big O toward the end. We won't get very far in that, but we will be talking about that all next week, which is Chapter 7 in its full glory. That will be pretty much everything we cover next week is coming out of Chapter 7, so it's all about algorithm and sorting and different algorithms. I will not be able to go – hang out with you today after class because I have an undergraduate counsel meeting. But it's a nice day so hopefully you'll have some more fun outdoor thing to do anyway. We will meet again next Friday. So put that on your calendar if you are the type who keeps a calendar. Anything administratively you want to ask? Stuff going on? How many people have started Boggle? Gotten right on it? Okay, there are two of you. Yay. All right. The other 50 of you, okay, okay. Now's a good time though. It is due a week from today so we gave you a little bit longer for Boggle because it's kind of recognition and, sort of, a bunch of big things that have to come together for that. But certainly not one of those things you wait for the last minute.

Just a reminder about another administrative things, which is the mid-term is coming up. So the mid-term actually is the Tuesday after Boggle comes in. The 19th, I believe, is the date of that and it's actually in the evening. If you are not available Tuesday to 7-9, that's actually when we're hoping to get almost all of you together, but if it really just doesn't work for you and there's no way you can accommodate it we will get you into a different time on Tuesday. You can send Jason an e-mail about that. I'll reiterate that next week just to remind, but just kind of right now if you want to keep your scheduling together if you can clear your evening of the 19th that actually is really good for us. Okay. So actually I'm gonna not even really work in the things. I'm gonna go back to my editing and talk about kind of where we were at at the end of Wednesday's lecture. Was we were doing a little bit of coding with the linked list itself. So having defined that recursive structure that has the information from one entry plus a pointer to the next and then so far the little pieces we have is something that we will print an individual entry, something that will create a new entry out in the heap, filling in with data that was read from the console typed in by the user, and then the last bit of code that we were looking at was this idea of building the list by getting a new entry.

So a pointer coming back from there that points to a new heap allocated structure that has the name and the address of someone and then attaching it to the front part of the list. So we talked about why that was the easiest thing to do and I'm gonna just re-draw this picture again because we're gonna need it as we go through and do stuff. Here's the main stack frame. It's got this pointer list, which is expected to point to an entry out in the heap. It starts pointing at null. We have this local variable down here, new one that is also a pointer to an entry and based on the result from get new entry, which will either return null to say there's no more entries or a new heap allocated thing if it gives us this new entry that says here's Jason and his phone number. That the lines there – the new ones

next equals list so new ones next field, right? Gets assigned the value of list, which affectively just copies a null on top of a null and then sets list to point to the same place that new 1 does. So on the first iteration, Jason is the new front most cell on the list, has a null terminator in the next field, which says there's no further cells. So we have a singleton list of just one cell. Now the subsequent iteration we call get new entry returns us a new pointer to another cell out here on the list. This one's gonna be Joel. He starts off as his own singleton list and then here's the attach again. New ones next field gets the value of list, so that changes Joel's next to point to where list does now. So I have two different ways to get to Jason's cell now.

Directly, it's the front most cell of the list still, but it's now following Joel. And then I update list to point to new one, so doing pointer assignment on this. Copying the pointers, making an alias, and now at the bottom of that loop, right? I now have list pointing to Joel, which points to Jason, which points to null. So my two entry list. Every subsequent cell that's created is prepended, right? Placed in the front most position of a list and then the remainder of the list trails behind it. So we should expect that if we put in the list A, B, C that if we were to go back and traverse it we'll discover it goes C, B, A. And so I was just about to show you that and then we ran out of time. So I'm gonna go ahead and finish that because I'm gonna write something that will print an entire linked list. It will take the list that we have and I'm gonna show you that. So the idea is to traverse a pointer down the list and print each one in turn. I'm gonna do that using a four loop. It may seem a little bit like an odd use of the four loop, but in fact what we're doing is really comparable to how you would do iteration down a link, an array, using kind of zero to n, you know, I++. We're doing the same thing, but instead of advancing an integer, which indexes into that, we're actually keeping track of a pointer, which moves down.

So the initial state of that pointer as we assign it to be the value of the list, so we alias to the first cell of the list. While that pointer points this invalid cell not to null, so as long as there are still cells to process then we'll print to cell. Then we'll advance the kind of equivalent of the I++. Here's the Kerr equals Kerr next. So advancing – so starting from Kerr equals Joel, right? To Kerr equals Jason and then Kerr equals null, which will terminate that. So as long as our linked list is properly terminated, right? We should print all the cells from front to back using this one loop. And so if I change this down here to be print list instead of just print to the front most entry and then I run a little test on this. So I enter Jake and his phone number, and I enter Carl and his phone number, and then I enter Ilia and his phone number, and I see that's all I have. Then they come back out in the opposite order, right? That Ilia, Carl, Jake because of the way I was placing them in the front of the list, right? Kind of effectively reversed, right? Them from the order they were inserted. Do you have a question?

Student: Do people ever make blank lists so that you can traverse backwards through them?

Instructor (Julie Zelenski):They certainly do. So the simplest form of the linked list, right? Has only these forward links, right? So each cell gets to the one that follows it in the list, but that is gonna create certain asymmetries in how you process that list. You're

always gonna start at the front and move your way to the back. Well, what happens if you're at a cell and you'd like to back up, right? It doesn't have the information in the single chain to do that. So you can build lists that either link in the other direction or link in both directions, right? To where you can, from a particular cell, find who proceeded it and who followed it. It just makes more pointers, right? And more complication. We'll see reasons why that actually becomes valuable. At this stage it turns out that it would just be complication that we wouldn't really be taking advantage of, but in a week or two we're gonna get to some place where that turns out to be a very, very useful addition to the list because it turns out that we really will need to be able to kind of move easily in both directions and right now we're not too worried about any direction other than the front ways. So this idea of the four loop, right? Is just one of the ways I could have done this. I could do this with a Y loop, right? Other sort of forms of iteration.

I could also capitalize on the recursive nature of the list, and this is partly why I've chosen to introduce this topic now, is because I really do want to stress that this idea that it's a recursive structure gives us a little bit of an insight into ways that operations on it might be able to take advantage of that recursive nature. So let me write this alternate print list, and I'll call it recursive print list, that is designed to use recursion to get the job done. So in terms of thinking about it recursively, you can think of, well, a linked list is the sequence of cells and iteration kind of thinks of the whole sequence at one time. Recursion tends to take the strategy of, well, I'm just gonna separate it into something small I can deal with right now and then some instance of the same problem, but in a slightly smaller simpler form. The easy way to do that with a linked list is imagine there's the front most cell. Which I could print by calling print entry, which prints a single entry. And then there is this recursive structure left behind, which is the remainder of the list and I can say, well, I can recursively print the list that follows. So print the front most cell, let recursion work its magic on what remains. Looks pretty good. Something really critical missing? Base case. Better have a base case here. And the simplest possible base case is list equals equals null. If list equals equals null then it won't have nothing to do, so I can just return or I can actually just do it like this. Say if list does not equal null, so it has some valid contents, something to look at, then we'll print the front most one and then let recursion take care of the rest. So I'll change my called print list to be a recursive print list. And I should see some [inaudible] results here. I say that Nathan is here, I say Jason is here, I Sara, Sara are you an H? I don't remember. Today it doesn't. Sara, Jason, Nathan coming out the other way. Okay. So I want to do something kind of funny with this recursive call. So it's not really in this case – both of these look equally simple to write.

We're gonna start to look at some things where this actually might buy us something. For example, lets imagine that what I wanted to do was print the list in the other order. So I don't have the list that links backwards. So if I really did want to print them with the last most element first, the way that they were added at the console, and the way I'm building the list, right? Is putting them in the other order. So, well, could I just take the list and print it back to front? I mean, it's simple enough to do that on a vector, if you had one, to work from the way back. Could I do it with the linked list? In the iterate formulation it's gonna actually get pretty messy because I'm gonna have to walk my way all the way

down to the end and then print the last one, and then I'm gonna have to walk my way down to the one that points to the last one and print it, and then I'm gonna have to walk my way down to the one that printed to the penultimate one and print it, but it does involve a lot of traversal and a lot of work. I can make the recursive print one do it with just one tiny change of the code. Take this line and put it there. So what I'm doing here is letting recursion do the work for me of saying, well, print everything that follows me in the list before you print this cell.

So if the list is A, B, C it says when you get to the A node it says okay, please print the things that follow me before you get back to me. Well, recursion being applied to the B, C list says, well, B says, well, hold onto B, print everything that follows me before you print B, when C gets there it says, well, print everything that follows me, which turns out to be nothing, which causes it to come back and say, well, okay, I'm done with the part that followed C, why don't we print C, and then print B, and then print A. So if I do this and print it through I put in A, B, C. Then they get printed out in the order I inserted them. They happen to be stored internally as C, B, A, but then I just do a change-a-roo was printing to print from the back of the list to the front of the list in the other order. But just the magic of recursion, right? A very simple little change, right? In terms of doing the work before the recursive call versus after gave me exactly what I wanted without any real craziness inserted in the code. We'll do a couple of other little things with you and I'm gonna do them recursively just for practice. If I wanted to count them, I wanted to know how many cells are in my list, then thinking about it recursively is a matter of saying, well, there's a cell in the front and then there's some count of the cells that follow it. If I add those together, that tells me how long this list is.

Having a base case there that says when I get to the completely empty list where the list is null then there are no more cells to count that returns my zero. So it will work its way all the way down to the bottom, find that trailing null that sentinels the end, and then kind of count on its way back out. Okay, there was one before that and then one before that and add those all up to tell me how many entries are in my address book. So I can do that here. I can say what is the count in my address book? And maybe I'll stop printing it because I'm – and so I can test it on a couple of things. Well, what if I put an empty cell in so I have a null? If I get one cell in there, right? Then I should have one and I can even do a couple more. A, B, C. Got three cells. I'll do another little one while I'm here. Is that when I'm done with the linked list, all those heap allocated cells, right? Are just out there clogging up my heap. So when I'm done with this list the appropriate thing to do is to deallocate it. I will note, actually, just to be clear though, that any memory that's allocated by main and kind of in the process of working the program that when you actually exit the program it automatically deallocated. So when you are completed with the program and you're exiting you can go around and be tidy and clean stuff up, but, in fact, there's not a lot of point to it.

The more important reason, right? To deallocation would be during the running of the program as you're playing games or monitoring things or doing the data. If you are not deallocating – if the program, especially if it's long running, will eventually have problems related to its kind of gargantuan memory size if it's not being careful about

releasing memory. When you're done deleting it manually or just letting the system take it down as part of the process there's not much advantage to. So if I'm gonna deallocate a list, if the list does not equal null there's something deallocate. I'm gonna do this wrong first and then we'll talk about why it's not what I want to do. That's good exercise remembering things. So I think, okay, well, what I need to do is delete the front most cell and then deallocate all those cells that follow it. So using recursion, right? To kind of divide the problem up. It's a matter of deleting the front cell and then saying, okay, whatever else needs to be done needs to be done to everything that remains. List dot next gives me a pointer to that smaller sub list to work on. As written, this does try to make the right delete calls, but it does something really dangerous in terms of use of free memory. Anybody want to help me out with that?

Student: After it deletes the first element it doesn't know where to find the rest of it.

Instructor (Julie Zelenski):That's exactly right. So this one said delete list. So if I think of it in terms of a picture here coming into the call list to some pointer to these nodes that are out here, A, B, and C, followed by null and I said delete list. So delete list says follow list to find that piece of memory out there in the heap and mark this cell as deleted, no longer in use, ready to be reclaimed. The very next line says deallocate list arrow next. And so that is actually using list, right? To point back into this piece of freed memory and try to read this number out of here. It may work in some situations. It may work long enough that you almost think it's actually correct and then some day cause some problem, right? Just in a different circumstances where this didn't succeed by accident. That I'm digging into that piece of freed memory and trying to access a field out of it, right? Which is not a reliable thing to do. C++ will let me do it, it doesn't complain about this. Ether it will compile time or run time in an obvious way, but it's just, sort of, a little bit of ticking time bomb to have that kind of code that's there. So there's two different ways I could fix this. One way would be to, before I do the delete is to just hold onto the pointer that I'm gonna need. So pull it out of the memory before we're done here. So I read it in there.

So the piece of memory that's behind it is still good. The delete actually just deleted that individual cell, so whatever is previously allocated with new, which was one entry structure, was what was deleted by that call. But what I was needing here was to keep track of something in that piece of memory before, right? I obliterated it, so that I can make a further use of it. The other alternative to how to fix this would be to just merely rearrange these two lines. Same kind of fix I did up above where go ahead and delete everything that follows so when I'm doing a list like this it would say, well, delete the D and C and only after all of that has been cleaned up come back and delete the one on the front. So it would actually delete from the rear forward. Work its way all the way down to the bottom, delete the last cell, then the next to last, and work its way out. Which would also be a completely correct way to solve this problem. Okay. Any questions on that one so far? Let me go back over here and see what I want to do with you next. So I talked about this, talked about these. Okay. Now, I'm gonna do something that's actually really pretty tricky that I want you guys to watch with me. All right. So this is the same code that we had for building the address book. The listhead, the new one, and so on.

And what I did was, I just did a little bit of code factoring. Where I took the steps that were designed to splice that new cell onto the front of the list and I looped them into their own function called prepend that takes two pointers. The pointer to the new entry and the pointer to the current first cell of the list and it's designed to wire them up by putting it in the front. So it looks like the code that was here, just moved up here, and then the variable names change because the parameters have slightly different names. The code as I'm showing it right here is buggy. It's almost, but not quite, what we want and I want to do this very carefully. Kind of trace through what's happening here, so you can watch with me what's happening in this process. So the variable is actually called listhead and this – I guess let me go ahead and do this. Okay. So let's imagine that we have gone through a couple of alliterations and we've got a good linked list kind of in place, so I have something to work with her. So let's imagine that listhead points to a cell A to B and then its got two cells, let's say, and then it's empty. Let's say I get a new one and this ones gonna be a J, let's say. Okay. So that's the state, let's say, that I'm coming into here and hit the prepend and I'm ready to take this J cell and put it on the front so that I have J, A, B on my list. Okay.

So let me do a little dotted line here that distinguishes my two stack rings. I'm gonna make this call to prepend and prepend has two variables, ENT and first, that were copied from the variables new one and listhead that came out of the build address book. Actually, it's not called main here. Let me call this build book. Okay. So I pass the value of new one as ENT. So what we get is a pointer to this same cell, a copy, and so this case, right? We copied the pointer, so whatever address was being held there is actually copied into here as this parameter. Similarly, listhead is copied into the second parameter, which is the pointer to the first cell. Okay. I want to do this in such a way that you can follow what's going on. So let's see if I can get this to work out. So it's pointing up there to first. Okay. So I've got two pointers to this cell A. One that came off the original listhead and one that came there from the copy in first. And then I've got two pointers to this J. One that came through the variable new one in the build an address book frame and one that was ENT in the prepend frame. Now, I'm gonna trace through the code of prepend. It says ent next field equals first. Okay. So ent's next field is right here. It gets the value of first. Well, first points to this cell. Okay.

So we go ahead and make the next field of J point to that cell. So that looks pretty good. That first line worked out just fine. It changed the cell J to point to what was previously the front of the list. So it looks like now it's the first cell that's followed by those. And then the next line I need to do is I need to update the listhead to point to J. So it says first equals ENT. So first gets the value of ENT. Well this just does pointer assignment. Sorry, I erased where it used to point to. And then I make it point to there, too. So at the bottom of prepend, if I were to use first as the pointer to the initial cell of the list it looks good. It points to J, which points to A, which points to B, which points to null. Everything looks fine here, right? At the end of prepend. But when I get back here to come back around on this loop; where is listhead pointing?

Student:

It's pointing to A.

Instructor (Julie Zelenski):It's pointing to A. Did anything happen to listhead in the process of this? No. That listhead points where it always pointed, which was whatever cell, right? Was previously pointed to. This attempt to pass it into prepend and have prepend update it didn't stick. Prepend has two pointers that are copies of these pointers. So like other variables, ents, and strings and vectors and anything else we have, if we just pass the variable itself into a function call then that pass by value mechanism kicks in, which causes there to be two new variables. In this case, the ENT and the first variables, that are copies of the two variables listhead and new one that were present in build address book.

And the changes I make down here, if I really try to change ENT or change first, so I try to make ENT point somewhere new or make first point somewhere new, don't stick. They change the copies, not the originals. This is tricky. This is very tricky because it's entering that the first line was actually okay. That what the first line tried to do actually did have a persistent affect and the affect we wanted, which is it followed the ENT pointer to this shared location and changed its next field. So dereferencing that pointer and mucking around with the struct itself did have a persistent affect. It wasn't another copy of the entry struck. There really is just this one entry struck that new one and ENT are both pointing to. So both of them, right? Are viewing the same piece of heap memory. Changes made out there at the structure, right? Are being perceived from both ends, but the pointers themselves – the fact that I had two different pointers that pointed to the same place, I changed one of my pointers, the one whose name is first, to point somewhere new. I didn't change listhead by that action. That listhead and first were just two aliases of the same location, but they had no further relationship that is implied by that parameter passing there. How do you feel about that? Question?

Student:

And first and ent are gonna disappear now, right, afterwards?

Instructor (Julie Zelenski):No. First and ent would go away, but that just means their pointers would go. So when prepend goes away, right? This space gets deallocated, so it's like this pointer goes away and all this space down here goes away. Then what we have is list still pointing to A and B and then new one, right? Is pointing to this J, which points to A, but, in fact, right? Is effectively gonna be orphaned by the next come around of the loop when we reassign new one. So that J didn't get prepended. It got half of its attachment done. The outgoing attachment from J got done, but the incoming one didn't stick.

This is pretty tricky to think about because it really requires really kind of carefully analyzing what the code is doing and not letting this idea of the pointer confuse you from what you all ready know about variables. If I pass – if you see a function call where I say binky of A and B, where A and B are integer values, if these are not by reference coming into binky then you know that when they come back A and B are what they were before.

If A was 10 and B was 20, they're still that. That the only way that binky can have some persistent affect on the values of A and B would be that they were being passed by reference.

If I change this piece of code, and it's just one tiny change I'm gonna make here, to add an ampersand on that first, then I'm gonna get what I wanted. So I want to draw my reference a little bit differently to remind myself about what it is. When I call prepend passing new one it got passed normally. New one is still just an ordinary pointer. So now I have two pointers where ENT and new one both point to that J. This one currently doesn't yet point up there. It's gonna in a minute, but I'll go ahead and put it back to its original state. Now, first is gonna be an alias for listhead, so that first is not going to be a copy of what listhead is. It's actually gonna be a reference back to here.

And a reference looks a lot like a pointer in the way that I draw it, but the difference is gonna be I'm gonna shade or, sort of, cross hatch this box to remind myself that first is attached a listhead and there's nothing you can do to change it. That first becomes a synonym for listhead. That in the context of the prepend function, any access to first is really just an access to listhead. So trying to read from it or write to it, you know, do you reference it, it all goes back to the original that this is just standing in for. That there actually is a relationship that's permanent from the time of that call that means that yeah, there really is now new variable first. First is actually just another name for an existing variable, this one of listhead.

So when I got through the ENT it gets the value – ENT's next gets the value of first. Well, first's value really is what listhead is. So that means it points up to A, so that part works as we were hoping before. Then when I make the change to first equals ENT, first is a synonym for listhead and that made listhead stop pointing to A and start pointing straight to J. And so this is still attached here. Let's do that. Let me see if I can just erase this part and make it a little clearer what's going on. So new one's pointing to J now, ENT is pointing to J, and listhead is pointing to J, and then this is still the reference pointing back to there. So now when prepend goes away, this extra pointer is removed, this reference is removed, all this space is reclaimed here in the stack, but listhead now really points to J, which points to A, which points to B as before.

So we wired in two pointers, right? One going out of the new cell into what was previously the first cell and then the first cell pointer now points to the new one rather than the original first cell. It was all a matter of this one little ampersand that makes the difference, right? Without that, right? Then we are only operating on a copy. We're changing our local copy of a pointer to point somewhere new; having no permanent affect on the actual state of it. So, actually, without that ampersand in there it turns out nothing ever gets added to the list. The listhead starts as null and stays null, but all these things will copy the value of listhead as a null, change it in the local context of the called function, but then listhead will stay null.

So, in fact, if I just ran this a bunch of times and I threw in a bunch of things without the ampersand I would find that all my cells just get discarded. I'd end up with an empty list

when I was done. All the orphaned out in the heap attached and not really recorded permanently. That's pretty tricky. Question?

Student: Why is the ampersand after this [inaudible]?

Instructor (Julie Zelenski):Because it is the – the ampersand applies to the – think of it more as applying to the name and it's saying this name is a reference to something of that type. So on the left-hand side is the type. What type of thing is it? Is it a string of anti-vector events or whatever? And then the ampersand can go between the type and the name to say and it is a reference to an existing variable of that type as opposed to a copy of a variable of that type. So without the ampersand it's always a copy. With the ampersand, you're saying I'm introducing first as a synonym for an existing entry star variable somewhere else and that in the context of this function access to this named parameter really is reaching out to change a variable stored elsewhere. It's quite, quite tricky to kind of get your head around what's going on here.

I'm gonna use this, actually, in running another piece of code, so I want to be sure that at this point everybody feels at least reasonably okay with what I just showed you. Question?

Student: What happens if we stop ampersand [inaudible]?

Instructor (Julie Zelenski):Jordan, you want to invert them or something like that?

Student: Yes.

Instructor (Julie Zelenski):It turns out that that won't work. Basically, it won't compile because in this case you can't take a pointer out to an entry reference. In fact, it just won't let you. So it doesn't really make sense is the truth. But think of it as like the – sometimes people actually will draw it with the ampersand attached even without the space onto the variable name. It's just a notation. A really good clue that this name, right? Is a reference to an existing variable and that its type is kind of over here. So type, name, and then name as a reference has that ampersand attached. Question?

Student: Is there a wrong side to passing both the pointers by reference?

Instructor (Julie Zelenski):Nope, not really. If I pass it by reference, right? Then it – nothing would change about what really happens, right? It would still leach into the thing and copy it. There is – for variables that are large, sometimes we pass them by reference just to avoid the overhead of a copy. Since a pointer is pretty small, that making a copy versus taking a reference it turns out doesn't really save you anything and actually it makes a little bit more work to access it because it has to kind of go one level out to get it. But effectively no.

In general though, I would say that passing things by reference, as a habit, is probably not one you want to assume for other kinds of variables. Just because that once you've passed

it by reference you've opened up the access to where if it changes it it does persistent and if you didn't intend for that that could be kind of a mystical thing to try to track down. So I think you should be quite deliberate about – I plan on changing this and I want to be sure I see that persistent effect. That I need that pass by reference there. Okay.

So let me go on to kind of solve a problem with the linked list that helps to motivate what a linked list is good for. Why would you use a linked list versus a vector or an array style access for something that was a collection? And so the built in array, which is what vector is built on, so they have the same characteristics in this respect. They store elements in contiguous memory. You allocate an array of ten elements. There's a block that is ten integers wide, let's say, where they were all sitting next to each other. In address 1,000 as one, 1,004 is the next, 1,008 and so on. What that gives you is fast direct access by index. You can ask for the third member, the seventh member, the 28th member, the 6 millionth member depending on how big it is, right?

By, sort of, directly computing where it will be in memory. You know where it starts and you know how big each element is. Then you can say here's where I can find the 10th element. That's an advantage, right? To the array vector style of arrangement. However, the fact that it's in this big contiguous block is actually very bulky. It means that if you were to want to put a new element in the front of an array or vector that has 1,000 elements you have to move over the 1,000 elements to make space. If it's starting to address 1,000 and you have them all laid out and you want to put a new one in the front, everybody's gotta get picked up and moved over a slot to make space. So imagine you're all sitting across a lecture hall and I decided I wanted to put a new person there, everybody gets up and moves one chair to the left, but there's not a way to just stick a new chair on one end of it using the array style of layout. Same for removing, right? So any kind of access to where you want to take somebody out of the array and close over the gap or insert in the middle and the beginning and whatnot requires everybody else moving around in memory, which gets expensive. Especially for large things, right? That's a lot of work. It also can't easily grow and shrink.

You allocate an array, your vector, to be a certain size underneath it. That's what vector's doing on your behalf. If you have ten elements and now you need to put in ten more what you need to do is go allocate a piece of memory that's twice as big, copy over the ten you have, and now have bigger space. If you need to shrink it down you'll have to make a smaller piece, copy it down. There's not a way to take a piece of memory and just kind of in place in memory where it is kind of extended or shrink it in the C++ language. So that the process of growing and shrinking, right? Requires this extra effort. So behind the scenes that's what vector is doing. When you keep adding to a vector, eventually it will fill to capacity. It will internally make more space and copy things over and you're paying that cost behind the scenes when you're doing a lot of additions and removals from a vector. The linked list, right? Uses this wiring them together using pointers, so it has a lot of flexibility in it that the array does not. Each element individually allocated. So that means you can pick and choose when and where you're ready to take some more space on and when you don't. So when you're ready to add a new person to the vector there's no worry about if there's a million elements there and now you don't have any

space you have to copy them. It's like you don't – you can leave the other million elements alone. You just take the heap, ask for one new element, and then splice it in. So the only allocation of the allocation concerns the individual element you're trying to do something with. You want to take one out of the middle; you just need to delete that one and close around the gap by wiring the pointers around it. So all the insert and remove actions, right? Are only a matter of wiring pointers. Wiring around something you're taking out, attaching one to the end or into the middle is splicing a pointer in, and is splicing the pointer out. So basically you typically have two pointers you need to reassign to do that kind of adjustment. If the element has ten members, it has a million members, right? Same amount of work to put a cell in there. No dependency on how big it is, right? Causing those operations to bog down.

The downside then is exactly the, sort of, opposite of where it came out in the array in the vector, which is you don't have direct access to the 10th element, to the 15th element, to the 260th element. If I want to see what that 260 of my linked list is I've got to start at the beginning and walk. I go next, next, next, next, next, next 260 times and then I will get to the 260th element. So I don't have that easy access to say right here at this spot because we don't know. They're all over the place in memory. Each linked list cell is allocated individually and the only way to get to them is to follow those links. Often that's not as bad a disadvantage as it sounds because typically, right? That you're doing things like walking down the array or the vector to look at each of the elements at the end you would also be walking down the linked list while you did it isn't really that bad. So in the case where you happen to be storing stuff in an index and you really want to reach back in there is where the linked list starts to be a bad call.

Student:

Can you take it on, like, from the list and find something?

Instructor (Julie Zelenski):Well, it can take – the thing is it – with a question with me like a long time. Like if I were looking through to see if there was an existence of a particular score in a vector of integers I have to look at them all from the beginning to the end. Now, the way I can access them is subzero, subone, subtwo, right? But if I'm walking down a linked list I'm doing the same sort of work. I have to look at each element but the way I get to them is by traversing these pointers. It might be that that's a little bit more expensive relative to the vector, but they're still gonna be about the same. If there's a million elements they're all gonna look at a million elements and whether it looked at them in contiguous memory or looked at them by jumping around it ends up being in the end kind of a very comparable amount of time spent doing those things.

What would be tricky is if I knew I wanted to get to the 100th element. Like I wasn't interested in looking at the other 99 that preceded it. I really just wanted to go straight to the thing at slot 100. The array gives me that access immediately. Just doing a little calculation it says it's here. Go to this place in memory. And the linked list would require this walking of 100 steps to get there. And so if there are situations, right? Where one of these is just clearly the better answer because you know you're gonna do that operation a

lot, right? You're gonna definitely prefer this. If you know you're gonna do a lot of inserting and rearranging within the list, the linked list may buy you the ease of flexibility of a kind of rewiring that does not require all this shuffling to move everything around. Question here?

Student: But if you were to know the size of the linked list?

Instructor (Julie Zelenski):Typically a linked list won't limit the size unless you tell it, but that's actually true about an array too, right? Which is you – the vector happens to it on your behalf, but underneath it, right? That the array is tracking how many elements are in there, so would a linked list. So you could do a manual count when you needed, which would be expensive, or you could just track along with that outer most pointer. Well, how many elements have I put in there? And then update it as you add and remove. So you'd likely want to cache that if it was important to you to know that. If a lot of your operations depended on knowing that size you'd probably want to keep it stored somewhere. Over here?

Student: Why does the vector use arrays then instead of linked list?

Instructor (Julie Zelenski):That is a great question. It's one that we actually will talk about kind of in about a week. It's like, well, why? So sometimes array – the vector is kind of taking the path of, well, it's just presenting you to the array in a slightly more convenient form. We will see situations where then the vector is just as bad as a choice as an array and then we'll see, well, why we might prefer to use a linked list and we will have – we will build linked lists instead basically. It has to make a choice and it turns out that for most usages the array is a pretty good match for a lot of ordinary tasks. The linked list happens to be a match for other tasks, right? And then we will build stuff out of that when that context comes up.

Student: Well, if you just, like, then you just, sort of, like in a vector class, like encapsulate all that and I – its final functionality would be the same.

Instructor (Julie Zelenski):You certainly could. So you could totally build a vector class that actually was backed by a linked list and we will see how to do that, right? And then it would have different characteristics in terms of what operations were efficient and which ones were inefficient relative to do an array backed one. And over here was also. Yeah?

Student: Well, I, just to – for using linked lists, like, so first in, first out or first in, last out seems to make really good sense, but are there – am I like missing –

Instructor (Julie Zelenski):You are way ahead of us, but, yeah, you're right on. The things like the stack and the Q, right? Seem like they're gonna be natural things that will work very, very well with a linked list. Things that required a lot of shuffling in the middle, right? It's a little unclear because you have to find the place in the middle, which is expensive, but then the insert is fast. The question is, well, which of these is actually gonna work out better? They both have some things they can do well and some things

they can't do well. So it will have to have know maybe a little bit more about the context to be sure which ones seem like the best solution and maybe in the end it's just a coin toss, right?

Some parts will be fast and some parts could be slow and anyway I can get the inverted form of that as an alternative that doesn't dominate in any clear sense. It's just like, well, they both have advantages and disadvantages. Question back here?

Student: Why was that infused to be based on a vector instead of a linked list?

Instructor (Julie Zelenski):So they – I think your question is kind of like, yeah, they likely are built on linked lists. We're gonna see how we can do it both ways, right? We'll see how we can do it on array, we'll see how we can do it on a linked list, and then we'll talk about what those tradeoffs look like. Well, which one ends up being the better call when we're there? So that will be about a week. So don't worry too much about it now, but when we get there we'll definitely see kind of both variants and then we'll talk about them in great detail. Okay.

So I'm gonna show you, just the last operation I want to show you on a linked list is doing an insert in sorted order and this is gonna require some pretty careful understanding of what's going on with the pointers and then we're also gonna see a recursive form of it that actually is a really clever and very elegant way to solve the same problem. So I'm gonna first do it iteratively just to kind of talk about. What's the mechanism for doing stuff iteratively on a linked list? It gets a little bit messy and so just be prepared for that. That I'm planning on taking my address book and building it up inserted order. So instead of doing a prepend where I just take each cell and put it on the front I'm actually assume that I'm gonna keep them all in order by name and whenever I get a new cell I want to put it in the right position relative to the other neighbors in the list that are all ready there.

So this is one of those places where it seems like the linked list is actually gonna do very well. I have to search to find the spot. Okay? Well, you're gonna have to search to find the spot in a vector, too. Once I find the spot it should be very easy to just wire it in without shuffling and rearranging anything around. It should be a pointer in and a pointer out. So I'm gonna start this insert sorted. It's gonna take the head of the list, right? As a bi-reference parameter because there are situations, right? Where we're gonna be changing where the list points to when the cells in the first one and then the new cell that's coming in is the second parameter. So we're gonna search to find the place where it goes.

So let's watch this go down it's loop. I do a four loop here that's like, well, Kerr starts this list while it's not null, advance by this, and if the cell that I'm trying to place, which in this case is the cell Rain, proceeds the one we're currently looking at, then this must be the place that goes in the list. So I look at Rain, I compare it to Cullaf and if Rain goes in front of Cullaf then this is where Rain should go in the list. Well, it doesn't actually proceed it, so I advance again. Then I look at Cullaf – Rain versus Lowery. Does Rain

proceed Lowery? It does not. So I just keep going. So I'm gonna break out of the loop as soon as I get to the place I compare Rain to Matt. Doesn't come out. I compare Rain to Thomas.

Now, this is the first time when new ones name came up as being less than Kerr's name. So that says, okay, well, Rain needs to go in front of Thomas. Okay. All is well and good almost. I have a pointer to Thomas. That's where my curve points to, right? And I want to put Rain kind of in the list right in front of Thomas. How do I get from Thomas to Matt? But it's not just enough to know Thomas, right? Because Thomas tells me like what's gonna follow Rain. So this is the cell where Rain's next is gonna point to where Thomas is because Rain is kind of gonna replace Thomas in the list and push Thomas one element down, but the situation is that I also need to know what proceeds Thomas. The cell that led into Thomas is the one whose next field needs to get reassigned to point to Rain.

So the way the loop is going, right? It has to go kind of one too far to know that it was there because, like, if I look at Matt I can say, well, does Rain go after Matt? Yes, it does. Right? But that's not good enough to know that it goes right after Matt. The only way I can tell if it goes right after Matt is to say, well, if the cell that comes up next, right? Is behind Rain is that it goes in between these two.

Student:[Inaudible] Thomas and then [inaudible] Thomas?

Instructor (Julie Zelenski):Yes. So at least I can do it by maybe kind of overwriting Thomas with Rain and then overwriting Rain with Thomas and kind of wiring them up. I'm gonna try to avoid that. I'm gonna actually say, well, let's just think about can we do it with leaving the information in the cells where it is? I don't know who else might point to Thomas, right? It would seem kind of weird if all of a sudden Thomas turned into Rain in some other way. That I don't know for sure who else is using this pointer. So lets assume that I can't do that variation of it, but I'd like to put Rain in between Matt and Thomas and the pointer I have is not quite good enough.

The cells as is, right? Have this asymmetry. They know what follows or not. What precedes them. So, in fact, what I'm gonna change here is I'm going to run two pointers down the list, kind of in parallel, where they're always one step apart. Then I'm gonna track what was the one I last looked at and what is the one I'm currently looking at and I call this dragging a previous pointer or trailing a previous pointer behind. So each time I'm about to advance Kerr equals Kerr next, the thing I'll do right before it is to assign previous to be what Kerr is now. So then it'll advance to the next one, test against null, and keep going. So at any given state, right? Previous is exactly one behind where Kerr is and there's one special case, right? Which is that at the very first time through the loop, right? Kerr is set to be the list. Well, what's the previous of the head of the list? There actually is no previous, so actually initialize previous to be null.

So the first time through, right? We've assigned previous to null and Kerr gets to go to the first one and then we say, well, is Cullaf – does Rain come before Cullaf? No. And so I'll advance both so move previous up to where Kerr is and then move Kerr up to the next

one. So now they're pointing at Cullaf and Lowery together and, again, I'm comparing Lowery to Rain. Does Rain go in front of Lowery? No. So then I move up previous and move up Kerr. Does Rain go in front of Matt? No. I advance them both, right? And then I have previous at the Matt entry, Kerr at the Thomas entry, and then Rain does go right in between those, right?

We know that it didn't precede Matt because we all ready made it through the loop. We know it does precede Thomas and that tells us it goes exactly in between them, right? That Matt should lead into Rain and Rain should lead into Thomas and that will have spliced it in the list at the right position relative to the other sorted entries. So I give myself just a little note here. Well, what are the possible values for previous? If the cell is gonna go anywhere in the list other than the very front, so as long as Rain follows at least one cell on the list, it's not the alphabetically first cell of all the ones I've seen, then previous will be some valid cell and potentially Kerr could be null, but there is a non-null prev in all of those cases.

There's also the possibility though that previous is null in the case where the new cell was the alphabetically first. So if that one actually began with an A, let's say it was Alena. Then the first test would be if Alena is less then Cullaf. It is. It would immediately break and so previous would be null, Kerr would point to the front most cell, and I've got a little bit of a special case there where inserting at the front of the list isn't quite the same thing as inserting further down. So let me look at the code that I added to this and I ran out of room to draw pictures, so we'll have to do a little bit on the board here.

That there are two pointers we need to assign for that new cell that's coming in. One is it's outgoing one and then another is it's incoming one. And so I have this list and, let's say, this new cell is going here. So this is like K, L, M, T, R. That the first line splices in the outgoing pointer in all cases. It says that the new ones next field is Kerr. So Kerr is the first cell, right? That we found that follows the cell we're trying to insert. It might be that Kerr is null, but that's totally fine too. In this case it happened to point – this is where Kerr was and this is where previous was. That it will make R point to T, so that is the outgoing pointer for the new cell and now there are two cases here.

The ordinary case is that there is some previous cell. In which case the previous cell's next field needs to get updated to point to the new one. So we use the pointer to this one. It's next field no longer points to T, but instead points to R. And so I inserted R in between M and T with that adjustment. In the case where the new cell, let's say, was the A, then Kerr would be here and previous would be null. I will set A as next field to the Kerr, which spliced the outgoing pointer of A to attach to the rest of the list. But here's the special case. There is no previous next field that's gonna point to A. A actually needs to get reassigned to be the front most cell of the list. So it looks like our prepend operation where I'm saying list equals new one and the list is some reference parameter to some list elsewhere that then gets permanently set to the new cell.

So I would call insert sorted from, like, the build address book function to attach it to the very front in this case. This is very common in linked list code, right? Is to have there be

a bit of a special case about either the first cell or the last cell or on the empty list, right? But that there is a large – all the cells in the interior kind of behave the same. They have a cell in front of them and a cell in back of them and that insert has certain properties, but there's a little bit of an oddness with that maybe that first cell is sometimes the very last cell that requires a little special case handling. In this case, right? Putting A in the front means somebody doesn't point into A. It's the listhead that points to A and so it's not somebody's next field that we're updating. It really is the listhead pointer itself. Questions about that one?

So I need to show you the recursive one. I'm not gonna be able to talk about it very much, but I'm gonna encourage you to take a look at it. Which is the same activity of inserting a cell into a list that's being kept in sorted order. Now, not using this iterative, not using this dragging of our previous pointer, but instead using a strong dense recursive powerful formulation here that basically has the idea that says, well, given a list that I'm inserting into, let's say it's the M through Z part of the list. I'm trying to insert R is I compare R to the front of the list and I say, well, does that become the new front of this list? That's what my base case is looking at.

Is the list empty or does this one go in front of it? If so, prepend it. Otherwise just insert it in the remainder of this list. So if it doesn't insert at the front of M then we pass the N or the O whatever is following it and have it kind of recursively insert into the remainder of the list. Eventually it will hit the space case when it either gets to the end of the list or when there is the cell we're trying to insert belongs in front of these remaining cells. In which case, we do a prepend right there and stop the recursion. A dense little piece of code. One I'll encourage you to kind of trace a little bit on your own to kind of get an idea of how it's working, but it kind of shows in this case that the recursion is really buying you something by making it, I think, a much simpler and more direct way to express what you want to do and then kind of the more mechanical means of doing it iteratively.

What we will talk about on Monday will be algorithms. We'll talk about algorithms, Chapter 7. Start talking about Big O and formal announcements algorithms and do a little bit of sorting. So I will see you on Monday and have a good weekend.

[End of Audio]

Duration: 52 minutes

ProgrammingAbstractions-Lecture14

Instructor (Julie Zelenski):Good afternoon. Apparently, Winter Quarter is over and it's now Spring. I don't know what happened to the weather, but I'm pretty happy about it. I hope you are too. Let's see. Where are we at? We are picking back up with the text. Looking at Chapter 7. Chapter 7 is gonna be the theme for pretty much the whole week. Talking about algorithms and different ways of analyzing them, formalizing their performance, and then talking about sorting as an example problem to kind of discuss in terms of algorithms and options for solving that problem. The mid-term, there's something. I'll say mid-term and everybody will shut up. Mid-term next Tuesday, so in terms of your planning and whatnot. Our mid-term is pretty late in the quarter. We had till we got to a substantial body of stuff to test and so that will come up next Tuesday.

What I give out today was some sample problems that are taken from old exams to give you an idea of the kind of things we're gonna ask you to do. Also a handout that was built up over time by the section leaders, which just includes kind of strategies and ways to think about prepping and getting yourself ready for the exam coming up. So hopefully those will be useful to you. I will put out a solution to the problems that I gave you in next time's class, but I'm trying to make sure that you have the problems without the solution in hand as a way of encouraging you to think about the problems from a blank piece of paper answering standpoint, which I think is the best way to get yourself ready for thinking about the actual exam.

So it is Tuesday evening and it is at Terman Auditorium, which is kind of just over the way in the Terman Building. All right. Yeah. Anybody totally done with Boggle? Ooh, look, way in the back. Hey, you can raise your hand high. You say I rock. Good for you. Other people making good progress on Boggle? Anybody who's gotten somewhere along the way run into anything they think they can save heartache and sadness for their fellow comrades by telling us up in advance? Smooth sailing? No problems? If you've got your recursion mojo, you are set for Boggle actually. Okay.

Okay. So let's talk about algorithms. Algorithm is one of the most interesting things to think about from a CS perspective. Kind of one of the most intellectually interesting areas to think about things is that often, right? We're trying to solve a particular problem. We want to put this data into sorted order. We want to count these scores and place them into a histogram. We want to search a maze to find a path from the start to the goal. For any of those tasks, right? There may be multiple ways we could go about solving that problem that approach it from different angles, using different kinds of data structure, solving it in forward versus solving it in reverse, is it easier to get from the goal back to the start? In the end the path is invertible, so maybe it doesn't matter which end we start from.

Would it be better to use an array for this or a vector? Would a set help us out? Could we use a map? Could I use iteration? Could I use recursion? Lots of different ways we could do this. Often sometimes the decision-making will be about, well, do I need to get the perfect best answer? Am I willing to take some approximation? Some estimation of a good answer that gives me a rough count of something that might be faster to do than

doing a real precise count? So what we're gonna be looking at, right? Is taking a particular problem, sorting is gonna be the one that we spend the most time on, and think about, well, some of the different ways we can go about putting data in sorted order and then talk about what the strengths and weaknesses of those algorithms are compared to one another.

Do they differ in how much time it will take to sort a particular data set? Does it matter if the data is almost sorted? Does it actually affect the performance of the algorithm? What happens if the data set is very large? How much memory is gonna be used by it? And we'll be thinking about as we do this, right? Is often that probably the most important thing is how does it run? How much time does it take? How much memory does it use? How accurate is its answer? But also given that brainpower is often in short supply, it's worth thinking about, well, is the code easy to develop, to write, to get correct, and kind of debug? It may be that a simpler algorithm that doesn't run as fast, but that's really easy to get working, might actually be the one you need to solve this particular problem and save the fancy thing for some day when really you needed that extra speed boost.

So I'm gonna do a little brainstorming exercise with you. Just to kind of get you thinking about there's lots of different ways to solve what seems like the same problem. All right. I'd like to know how many students are sitting in this room right now. All right. I look around. Michelle, what's the easiest way and most reliable way to just get a count, an accurate count, of everybody in this room?

Student: Start by identifying what kind of [inaudible].

Instructor (Julie Zelenski):Yeah. Just have a strategy. Some people will use the front row. And I see one, two, three, four, five and then I go back and I just work my way through the room all the way to the end, right? What I've done, if I actually remember my grade school counting, right? Then I should have come up with a number that matched the number of people in the room and tells me, okay, there's 100 people, let's say, that are here. So that will definitely work. That's the easiest most ways to, sort of, likely approach you'd think of. You need to count something, well, then count them, right?

Now, I'm gonna talk about ways that we could do it that maybe aren't so obvious, but that might have different properties in terms of trading it off. Let's say that the goal was to count all of the people in Stanford Stadium, right? And so I've got a whole bunch of people sitting there. The count in turn walking around the stadium doesn't scale that well. However long it took me to work my way through the rows in this room, right? When you multiple it by the size of the Stanford Stadium is gonna take a long time and as people get up and go the bathroom and move around and whatnot I might lose track of where I'm at and all sorts of complications that doesn't really work well in the large.

So here's another option. Anyone else want to give me just another way of thinking about this? Yeah?

Student: Multiply yourself into more Julies.

Instructor (Julie Zelenski):Because we can only – one Julie's not really enough, right? So recursion. Can recursion help us out here, right? Can I do some delegating, right? Some subcounting to other people? So in the case of this room, it might be that I pick somebody to count the left side, somebody to count the right side, and somebody to count the middle. And maybe even they themselves decide to use a little delegation in their work and say, well, how about I get some volunteers on each row? That's sort of idea would work extremely well in the Stanford Stadium as well because you just divide it up even further. You say, well, here's this row, this row, that row and kind of having all of these delegates working in parallel to count the stadium, right? Getting recursion to kind of solve that same problem.

What if I wanted – I was willing to tolerate a little bit of inaccuracy? What if I wanted an estimate of the people that were in this room and I was willing to accept a little bit of sampling or estimation error?

Student: Maybe you could take, like, a little area of seats and count how many people are in those seats and then multiply it by how many of those areas are in the room?

Instructor (Julie Zelenski):Yeah. So it's like this capacity. Somewhere there's a sign, right? On one of the doors here that will say, oh, this room seats 180 people. So I know that there are 180 seats without doing any counting. So if I took a section. So I take ten seats, let's say, or 18 seats for that matter. I take 18 seats and I count those 18 seats how many are occupied and let's say half of them of the 18 that I look at were occupied then I can say, well, the room's about half full, right? And then I say, well, okay, 180 seats, half of them, right? Means 90 people are here. So in that case it's not guaranteed to be accurate, right? If I happen to pick a particularly dense or sparsely populated section, right? I'd be getting some sampling error based on that, but that would also work in the Stanford Stadium, right?

We know how many seats are in the Stanford Stadium, right? You pick a section and you count it. You count 100 seats worth to get a percentage of how full it is and then just multiply it out to see what you get. There's a variation on that that's kind of a – maybe not the first thing that would occur to you, right? Is just that the idea of counting some subsections is kind of interesting as a way to kind of divide the problem and then say, well, in the small can I do some counting that will then scale up? So, for example, if I had everybody in this room and I said, okay, think of a number between one and ten. Everybody just think of a number independently on your own. Okay.

If you were thinking of the number four, raise your hand. I got one, two, three, four, five, six, seven. Seven people. And said, okay, well, if I figure that everybody was just as likely to pick a number between one and ten as four, then those seven people, right? Represent 10 percent of the population. So maybe there were 70 of you. Again, totally based on randomness and sampling error, right? If I happen to pick a number that's very unpopular with my crowd, right? Or very popular, I get kind of an artificial estimate. But

it does tell you a little bit about the nature of just solving this problem. It's like how much error am I willing to tolerate and how much time I'm willing to invest in it, how big is the problem I'm trying to solve? What kind of things can I do that might, in the end, give me some estimate or accurate count, right? Depending on what I'm willing to invest in the time spent. So random thought for you. Here's another one. I can take the access class list, right? And I can take the first ten people and call out their names and find out if they're here and on that, right? I would get this estimate of, well, what percentage of my students actually come?

So if my class list says that there's 220 students, which it does because where are they? And I take the first of ten students, right? And I call them out and I discover three of them are here I can say, oh, about 30 percent of my 220 students come, 66 of you. And then I would also have the advantage of knowing who was a slacker and I could write it down in my book. No, no, no. I'm sure you guys are all at home watching in your bunny slippers. All right. So let's talk about in terms of computer things. Like that the situation often is you have a problem to solve, right? You need to do this thing. You need to solve this maze or compute this histogram or find the mode score in your class or something like that and you have these different options about how you might proceed.

Which data structure you use and how you're gonna approach it and stuff like that. Certainly one way to do it, and not a very practical way to do it, would be to say, well, I'll just go implement the five different ways I could do this. For example, when I was running your maze assignment that's what I did. I wrote, like, ten different maze solvers until I decided which one I was gonna give you. That's not very practical, right? If your boss said to you, you know, you need to solve task A and you're response was I'll solve it ten different ways and then I'll come back to you with the best one. That might take more time than you've got. So you would have to write it, you'd debug it, you'd test it, and then you could kind of time it using your stopwatch to find out how good did it do on these data sets.

Yeah. Well, there's some issues with that. First of all, you have to go do it, right? Which is kind of a hassle. And then also it's subject to these variations, like, well, what computer were you running on? What OS were you running? What other tasks were running? Like did it – it may not be completely reliable whatever results you saw. What we're gonna be more interested in is doing it in a more formal abstract way, which is just analyzing the algorithm from a pseudocode standpoint. Knowing what the code does and the steps that it takes, right? What can we predict about how that algorithm will behave? What situations will it do well? What situations will it do poorly? When will it get the accurate answer or an estimate? How much time and space can we predict it will use based on the size of its input?

That would tell us based on just some descriptions of the algorithm which one might be the best one that's gonna work out in practice. Then we can go off and really implement the one that we've chosen. So this is actually working more the mathematical sense, less on the stopwatch sense. But helps us to analyze things before we've gone through all the process of writing the code. So what I'm gonna do with you today is a little bit of just

analysis of some code we didn't write to talk about how it behaves and then we're gonna go on and talk about the sorting problem and some of the options for that.

So this one is a function that converts the temperature. You give it a temperature in Celsius and it converts it to the Fahrenheit equivalent by doing the multiplication and addition. So the basic, sort of, underpitting of kind of looking at it formally is to basically realize that we're gonna count the statements that are executed. Assuming, right? In this very gross overgeneralization that each action that you take costs the same amount. That doing the multiply and the divide and the addition and a return, that all those things are, like, one unit. In this case, I'm gonna call it a penny, right? It took a penny to do each of those operations. That's not really accurate to be fair. There's some operations that are more expensive at the low level than others, but we're gonna be pretty rough about our estimate here in this first step here. So there's a multiply, there's a divide, there's an add, there's a return, right? And so I have, like, four statements worth of stuff that goes into asking a temperature to be converted. The other thing you'll note is that, does it matter if the temperature is high or low? If we ask it to convert Celsius of zero, Celsius of 100, Celsius of 50 it does the same amount of work no matter what. So actually that the – for whatever inputs, right? You give it this function pretty much will take the same amount of time. That's good to know, right? There's certain functions that will be like that.

So we would call this a constant time function. It's, like, no matter what you ask it to convert it takes about the same amount of time. That doesn't mean that it takes no time, but it is a reliable performer of relative inputs. So let's look at this guy. This is one that takes in a vector and then it sums all the values in the vector and then divides that sum by the total number of elements to compute its average. Okay. So, again, kind of using this idea that will everything you do cost you a penny? How much is it gonna cost you to make a call to the average function?

Well, this is a little tricky because, in fact, there's a loop in here, right? That's executed a variable number of times depending on the size of the input. So first let's look at the things that are outside the loop, right? Outside the loop we do things like initialize the sum, we initialize I before we enter the loop, right? We do this division here at the bottom and this returns. There are about four things we do outside of getting into and then iterating in the loops. So just four things we pay no matter what. Then we get into this loop body and each iteration through the loop body does a test against the I of the size to make sure that we're in range, right?

Does this addition into the sum and then increments I. So every iteration has kind of three statements that happen for every element in the vector. Zero, one, two, three, and whatnot. So what we get here is a little bit of constant work that's done no matter what and then another term that varies with the size of the input. If the vector has ten elements, right? We'll do 30 steps worth of stuff inside the loop. If it has 100 elements we do 300. In this case, right? For different inputs, right? We would expect the average to take a different amount of time. Yeah?

Student: If n, the vector size MP though we still initialize I?

Instructor (Julie Zelenski):We still initialize I. So I was actually counted in here, right? The init I, right? That was in there. We actually have one more test though. We do a test and then we don't enter the loop body. So there's actually one little piece maybe we could say that there's actually like three n plus five. There's one additional test that doesn't enter into the loop body. We're gonna see that actually that we're gonna be a little bit fast and loose about some of the things that are in the noise in the end and look more at the big leading term, but you are right. There is one more test than there are alliterations on the loop. That last test that has to fail.

The idea of being that, right? We have three statements per thing. A little bit of extra stuff tacked onto it if we double the size of the vectors. So if we compute the average of the vector that has 100 elements and it took us two seconds. If we put in a vector that's 200 elements long, we expect that it should about double, right? If it took two seconds to do this half-sized vector, then if we'd have to do a vector that's twice as long we expect it's gonna take about four seconds. And that prediction is actually at the heart of what we're looking at today. Is trying to get this idea of, well, if we know something about how it performs relative to its input can we describe if we were to change the size of that input to make the maze much bigger or to make the vector much smaller?

What can we predict or estimate about how much time and memory will be used to solve the problem using this algorithm? So here is one that given a vector computes the min and the max of the vector and it does it with two loops, right? One loop that goes through and checks each element to see if it's greater than the max it seems so far and if so it updates the max. And similarly almost identical loop here at the bottom that then checks to see if any successive element is smaller than the min it's seen so far and it updates the min to that one. Okay.

So a little bit of stuff that happens outside the loop, right? We init I two times. We init the min and the max, so we've got, like, four statements that happen outside. And then inside the loop, right? We've got a test and an assignment, a comparison, an increment, and we have two of those loops, right? And so we have a little bit of stuff outside the loops, about eight instructions worth that happens per every element. So we look at it once to see if it's greater than the max. We actually look at everything again, right? To see if it's the min. I'm gonna use this actually as written, right? You might think, well, I could make this better, right?

I could make this better by doing these two things together, right? Inside that loop, right? So that while I'm looking at each element I can say, well, if it's the max or if it's the min, right? If I look at it just once I can actually kind of do both those comparisons inside the loop and what we're gonna see is that in terms of kind of this analysis that's really not gonna make any difference. That whether we do a loop over the vector once and then we go back and loop over it again or whether we do twice as much stuff to each element inside one loop, it's for the purposes of the analysis we're looking at it ends up coming down to the same things. Which is, yeah, there is something that depends on the size of the input directly. Which is to say that if it were it will increase linearly with the change in size.

So I have these numbers, like, four statements for the Celsius to Fahrenheit. Three n plus four. Eight n plus four. These tell us a little bit about, well, will get extremes always take more time than average or C to F? That given the way those numbers look it looks like, well, if you plugged in the same value of n you'd have the same size vector. That it should take more time to compute get extremes versus compute the average. We're gonna discover that, actually, we're not gonna, actually, try to make that guarantee. That what we're – we're not really so much interested in comparing two algorithms and their constant factors to decide which of these two that kind of look about the same might be a little bit faster. What we're gonna try and look at it is giving you kind of a rough idea of the class an algorithm fits in. What its growth term, the kind of prediction of its growth, is.

In this case, both averaging get extremes in terms of growth, both have a leading term that depends on n with some other noise and a little bit of a constant thrown in the front. That means that both of them, right? We'd expect to grow linearly in a straight line based on the change in n. So if you doubled the size of the thing then average would take twice as long as it previously did. So should get extremes. But the data point for does average of a vector of 100 elements take the same amount of time as get extremes of 100 is not what we're looking at predicting, right? We're trying to just tell you things about average against itself over different ranges of inputs.

So if we know that average takes two milliseconds for a 1,000 elements. If we double the size of that we expect it to take twice as long, four milliseconds. If we increase it by a factor of ten we expect to increase the time by a factor of ten, right? So it should go up to 20 milliseconds. Get extremes might take more or less for a particular element. We expect it probably will take more because it does a little bit more work per element, but we really want to time it to be confident about that rather than making any estimation here.

So in terms of what's called big O notation we're gonna see that we're going to kind of take those statement counts and we're gonna summarize them. That we can do a little bit of niggling to figure out what those numbers are, but what we're gonna then do is take the leading term, the largest term with the largest coeff – value of the input number, in this case n. Ignore any smaller terms and then drop all the constants, all the coefficients, right? If you know it takes n over 2, it's just gonna be n. If you know that it takes ten statements, then it's gonna just be constant if there's no term of n in there. If it takes 10n then it's still n. 10n and 2n and 25n and 1/15n all end up just being n.

So we might have this idea that we've estimated the time using n and having those constants in. Well, when it comes time to describe it in terms of big O, we focus on what's the – oh, that, the subscript got lost on that. And that just makes no sense as it is right there. I will fix it in a second. So the 3n plus 5, right? The leading term is n, the coefficient 3 gets dropped. It's just linear. We expect that as we change the input the time will change linearly with that change. The 10n minus 2, same class. O of n. Even though it didn't have kind of the same constants coming into it, right? It has the same growth pattern that they both are lines.

The slope of them might be slightly different, but in terms of big O we're being pretty loose about what fits into this class. 1/2n squared minus n, the leading term here is the n squared. The n, subtract it off. When n is small n squared and n are actually kind of in range of each other, but then what we're looking at is in the limit. As n gets very, very large, n gets to be 1,000, n gets to be – n squared is a much bigger number, right? When n is a million then n itself was and so as we get to these larger points kind of out into the limit this term is the only one that matters and this is just a little bit of the noise attached to it. So that's why we're gonna summarize down to that.

What this one is supposed to say, and it actually is incorrect here, it just should be two to the n and n to the third power, which summarizes to two to the n. Two to the n grows much, much more rapidly than n cubed does and so as n gets to be a very large number. Even a fairly small number, you know, two to the tenth, right? Is all ready 1,000. Two to the 20th is a million, which is much bigger than these guys over here. So as we get to the long run it must be much bigger. I'm gonna put a note to myself though to fix that before I put that on the web.

Student: It should be O to the 2n?

Instructor (Julie Zelenski):Yeah. It should be O to the 2n. So that should be two to the n, n to the third, and my two to the n there. My subscripts got blasted out of that. And so we're trying to describe this growth curve, like, in the limit, right? We're avoiding the details when they don't matter and they don't matter when n gets big enough, right? That only the leading term, right? Is predicting without this other stuff kind of just ignoring it. So this is very sloppy math, right? So those of you who are more trained in the, sort of, mathematical sciences may find this a little bit alarming. Which is just how kind of fast and loose we're gonna be. The only way all these terms left and right just kind of summarizing down to, okay, here's this leading term and how it relates to n. Everything else, right? Totally uninteresting.

We could have a function, for example, that did 1,000 conversions of Celsius to Fahrenheit, but if every time you call it does 1,000 conversions that means no matter what input you give to it doesn't change. That's considered O of one or a constant time. Similarly, right? For doing average. If we did an average that somehow operated over the vector one time and another one that did it ten times or 20 times looked at each element 20 times, right? They would still both be O of n. Whatever time they took, right? On a vector of a certain length we double that length. We'd expect to double the time. More formally, right? In terms of what the math really means is that we say that O of F of n, so O, if it's big O of some function, it describes an upper bound on the time required.

Meaning that for sufficiently large values of n and some constant C that we get to pick that that would bound the curves. So if you imagine what the real curve looks like, it grows and maybe it flattens out or maybe it goes up very sharply. That what C of F of n gives you kind of some upper bound of that. A curve that stays above it at – for at some point of n and the limit, right? Will dominate it from there to affinity. So describing kind

of where it's gonna hit at that upper bound gives us some measure of what's going on. Okay.

So we can use this to predict times. That's probably what's most valuable to us about it is knowing that I have an n – a linear algorithm. So an O of n algorithm we'll call linear. And n squared algorithm I'll call quadratic. N to the third I'll called cubic, right? That's gonna tell us that those curves, right? You know what a line looks like. Well, you know what a parabola looks like coming out of an n squared where it's growing much more sharply and early kind of making the decision. So it might be, right? That if I know that it takes three seconds to do something for 5,000 elements then I have a linear algorithm. That 10,000 elements, right? Should take twice as long. 20,000 should take another, a factor of two on that. So 12 seconds worth. That I may have an n squared algorithm.

So one that I expect to actually perform more poorly in the long run, right? This n squared versus n tells you that it's gonna more sharply grow. That for some values of n in simply 5,000 is the case where the n squared algorithm actually outperforms it in a small case. That's not uncommon actually. But, right? As it grows, right? As I get to larger and larger values of n the fact that it is going by factor of four, right? The square of the doubling as opposed to linear means it's gonna quickly take off. And so I take my 5,000 elements that took two and a half seconds. I put an input that's twice as large into it. It's gonna take a factor of four, right? From there. And if I go from 10,000 to 20,000 another factor of four is gonna bring that up to 40 seconds or so compared to my more modestly growing linear algorithm there.

So if I was facing some task, right? Where I had an option between a linear algorithm and a quadratic algorithm it's telling me that in the long run the quadratic algorithm for sufficiently large inputs, right? Is going to bog down in a way that the linear one will be our kind of strong performer in the larger cases. So some algorithms actually have a variable run time expectation that you cannot say with all assuredness how much time it will take because actually depending on the input and the characteristics of the input it may do more or less work. So average looks at every element in the vector and sums them all up and does the division. It doesn't matter what elements are in there. It's very reliable in that sense.

Something like search. So this is a linear search function that given a vector of names and a particular key just walks the vector looking for a match. If it finds it, it returns true. If it exhaustively searched the whole thing and didn't find it, it returns false. Okay. So we've got what looks like an O of n loop that we'll look at the things, but there are some cases, right? Where it just doesn't do very much work at all. For example, if it finds the key in the very first spot, right? If I'm looking for Bob and Bob is the first thing in there, then I immediately return true and I do no more work. And so it doesn't matter whether Bob was followed by a million more names or ten more names, right? That, in fact, it is a constant time operation to just access that first element and return it. Similarly for other things that are close to the front.

It looks at a few of them and it's done. And so we would call those the best case of the algorithm. So we can divide this into things. It's like, well, what of the best case input, right? What can we expect out of the performance? Well, the best-case input would be it's found in the first few members. In which case it's an O of one algorithm for those situations. That's not often very interesting to say, well, here's a particular input that causes it to immediately be able to calculate something in return. Yeah. Not so interesting. So it's worth knowing that such a thing exists, but it turns out it's not likely to tell us a lot about the general performance of this algorithm.

If it's in the middle or it's in the last slot, right? We're gonna be looking at a lot of the elements to decide that it's there or not. In the absolute worst case, right? So what's the input that causes it to do the most amount of work is the one where it doesn't find it at all. Where it looks at every single element, never finding the match, and then comes out and returns that false. And the worst case, in this case, is a fairly likely thing to happen, right? We're searching because we happen to believe it might not be there and that gives us this upper bound on how bad it gets. So in the worst case it looks at everything and it is definitely O of n. So we have this kind of constant best case, an O of n worst case, and then our average case is gonna be somewhere in the middle there. This actually is a little bit harder to predict or to compute precisely in most situations because you have to know things about, well, what are the likely range of inputs? So typically the definition is that it's averaged over all the possible inputs. Well, given the search it's pretty hard to say what are all the possible inputs for it? It's like you can make some assumptions, like, well, all in – all permutations of the list are equally likely and the name is in the list about half the time, let's say.

You can just – you have to make up some parameters that describe what you believe to be the likely inputs and then you can say, well, if it were in the list, if it's equally likely to be in the front as in the back, then on average it's gonna look at n over two. It's just as likely to be in all the front slots. So, let's say, if you were gonna call it n times and the name you were looking for was going to be in each individual slot exactly one time, then the total random perfect permutation case. So then it would have looked at n over two of them. Sometimes it looked at one, sometimes it looked at n minus one, sometimes as it looked at n over two, sometimes n over two plus one, and whatnot. That over all of them it looked at about half.

And then if there was another set of calls to search for it, right? Where it never found it, it would be looking at n, right? And so you can say, well, sometimes we look at n over two, sometimes we look at n. On average we're looking at about three-quarters of them, right? In big O, since we get to be a bit sloppy here, we can say, well, this all just boils down to being linear. That O of n still described to the growth in the average case, which is the same number we got for the worst case. So this is a little bit tricky, but this is actually the one that's probably the most interesting, right? Which is just in normal practice, how is this thing gonna behave?

So the last little thing I need to complete before we can go on and start applying this to do something interesting, is to talk a little bit about how we do recursive algorithms because

they're a little bit trickier than the standard iteration and counting. So the counting statements are kind of saying, oh, I've got this loop followed by this loop and this thing happening, right? Will get you through the simple cases. Well, what do we do when we have a recursive algorithm, right? Well, we're trying to compute the time spent in solving something that, in affect, is making a call to the same algorithm and so we're gonna likely have some recursive definition we need to work through.

So this is the factorial function. If n equals zero, return one, otherwise return n times the factorial n minus one. We're interested in doing kind of the statement counts or kind of summarizing the time for an input whose value is n. And so what we write down is what's called a recurrence relation that largely matches the code in terms of saying how much work is done in the base case? In the different cases that are being handled? Typically there is a base case or two where you say, well, if it's in these cases, right? We do these easy things. So if n is exactly zero then we do O of one worth of work, right? We do a little bit of constant work here. We do a comparison and a return.

In the otherwise case when it is not zero, then we do a little bit of work. In this case, a return, a multiply, a little bit of constant work plus whatever time it takes to compute the factorial of n minus one. So we have T of n defined in terms of T of n of something else, right? Which is exactly what we'd expect in a recursive function. This is called a recurrence relation, so that solving for T of n means working with a side that refers back to that same T, but on a smaller version of the problem in n over two and n minus one. Some other variation of that recursive call. So let me show you how we make that go into closed form.

The idea – and actually I'm just gonna do this on the board because I think it's easier if I just write what's going on and you can watch me develop it. So I have this recurrence relation. Even equals one if n equals zero and then it's one plus T of n minus one otherwise. So I'm trying to solve for T of n into a closed form. So I'm gonna start with it's non-recur – the non-base case form. So the recursive step. And then what I'm gonna do is I'm actually just gonna reply repeated substitution to take this term and expand it out. So I know it's one plus whatever it takes to do T of n minus one. So if I go back over here and I say, well, T of n minus one must be one if n minus one equals zero or it's one plus T of n minus two. So kind of plugging it into the original formula and getting the expansion one layer in.

So I can say, well, this is one plus one plus T of n minus two. And if I apply that again, right? I should get another term of one. So after I have done this I times then I will have a bunch of ones added up together and I will have subtracted an I from T down to some term. So I'm just gonna – each of these represents kind of like this is a little bit of work from the n, which does the n minus one, which brings the quality of my two. So each of the cul's kind of in the stack frame has a little bit of a contribution to add to the whole thing and then what we're looking at is how many times do we have to do that expansion and substitution, right? Before we hit this base case, right? Where T of n exactly equals one.

So we're looking for the case where n – actually it's n equals zero. So where n – so I want to do this until n minus I equals equals zero. Okay. So I need to have done this I times where I is n. So if I say I set I equaled to n, then I'll have one plus one plus one n times, and then I have plus T of the n minus n over here, which is my T subzero. T subzero, right? Immediately plugs into that base case and says, well, there's just one more thing to do when you get to that and so what I basically have here is n plus one. So n multiplications plus one little tack on for the base case, which in terms of big O is just linear. A little bit of extra work in the noise there, but that means that kind of as it seems more predictable, right? That factorial over particular input, right? Is linear in the number you ask to compute the factorial.

The factorial of ten takes ten multiplications, right? The factorial of 20 takes 20 multiplications. So if you change the size of your input, right? You double it; it should take twice as long. However much time it cost you to compute the factorial of ten is gonna take twice as much time to compute the factorial of 20. Okay. That kind of makes sense, but it's good to know kind of how I can do this math, right? To work this out, right? So this idea of taking the term, repeatedly substituting and expanding, generalizing my pattern, and say, well, after I substitutions worth where am I at? And these correspond and kind of thinking about the recursion tree. What calls are made and then how deep does the recursion go before I hit the base case and that tells me how to stop that expanding and then substitute back in for the base case to compute my total result.

I'm gonna do another one, so if you want to just watch and we'll do the math for this together, too. This is the Towers of Hanoi example that moves the tower away of n minus one off, moves the single disk on the bottom and then moves that tower back on. And so the recurrence that we're working with is one when n equals zero. So when n equals zero, right? We have a zero height tower to move and we actually do no work in the function, right? We just do that test and return, so if n equals zero there's no work to be done. So that's the easy case for us, right? Is when it's zero do no work. Otherwise, right? We move the bottom most disk. So we do a little testing and moving of that disk. We'll call that the constant amount of work that's in the function itself.

And it makes two recursive calls each of a tower of height n minus one. So otherwise, right? Two calls, which gives a little clue to what the tree looks like. It'll branch twice at each stop and then it's one closer to that base case gives us a sense that the recursion depth is likely to be linear here. So let me go through the process of making this work. I've got T of n equals one plus two, T to the n minus one. So then I take T to the n minus one and I plug it in over here to get one plus two T to the n minus two. This whole thing is in a – multiplied by two though because I have two of those, right? From the original call which then itself made two. So, in fact, if I multiply through, right? I've got one plus two plus four T to the n over two.

If I apply my substitution again, one plus two plus four times T to the n minus two is one plus two, T to the n minus three, and then let the multiplication go through again. One plus two plus four plus eight T to the n minus three. And so each expansion of this, right? Is causing the number of towers that are being moved around to go up by a factor of two.

So each time I do this, right? I went from two towers to move to four towers to eight towers, but those towers each got one shorter in the process. So kind of telling us a little bit about what the recursion tree looks like, right? Is there is a branching factor of two all the way down and that the depth of this thing is gonna bottom out linearly because this was a tower by ten, nine, eight, seven, and so on down to the bottom.

So if I imagined this happened I times, so to generalize my pattern. I've got one plus two plus four plus two to the I. So after I've done this that number of times, right? Actually it's two I minus one plus two to the I, T n minus I. So I have subtracted I off the heights of the tower. I have gone up by a factor of two each time I did that and then I have these sums in the front, which represent kind of the single disk that got moved in there. One plus two plus four plus eight plus so on down to there. And so the place I want to get to, right? Is where n equals zero. So I actually want to set n, I equal to n here. I wrote that backwards, let's say I equals n. I plug that in I've got one plus two plus four plus all the powers of two to the n minus one plus two to the n, T to the n minus n, which is T to the zero, which is just one.

So what I've got here is following along. Is T to the n is one plus two plus four plus two to the n, right? So two to the n minus one plus two to the n times one. So I've got the geometric sum here. You may or may not all ready know how to solve this one, but I'm just gonna go ahead and solve it in front of you to remind you of how the process works. I've got the powers of two. One plus two plus up to the n. So that's the term I'm looking for. I want to call this A just to mark it. And then what I'm gonna compute for you is what two times A is and I'm gonna write it underneath. So if I multiply this by two I have one times two, which is two. Two times two, which is four, right? Four times four at two, which is eight. And so on.

And so I should get basically the same sequence of terms, but shifted over by one. I don't have the factor of one and I do have an additional factor of two to the n times another power of two at the end. So I've got the whole term kind of shifted over by one. That's my two A so that's the same thing I'm looking for, but doubled. And then I'm gonna basically take this thing and I'm gonna add negative A to two A. So subtracting A off of two A so that all these terms will go away. If I have this minus this, this minus this all the way across, right? The terms I'll be left with is two to the n plus one minus one is A subtracted from two A so the two to the T to the n that I'm looking for is two to the n plus one minus one. That's my term there.

And if I summarize in my big O way, ignoring the constants, throwing away these little parts of the terms, right? That are in the noise. That what we really have here is something that grows, right? Exponentially by about a factor of two for each additional disk added to the tower. So however much time it took to move a tower of height ten, right? A tower of height 11 we expect to take twice as long. So not growing linearly at all, right? Much, much, much sharply growing, right? What exponential growth looks like. Two to the n, right? Even for small values of n. Ten, 20, right? Is up into the millions all ready in terms of disks that need to be moved.

So this sort of strategy, right? Is good to know. When you're looking at this recursive things, right? Kind of having this analysis. It says, okay, well, where did I start from? What's the time? And then just being pretty mechanical. Sometimes you can get to something here that requires a little bit of algebra to work out, but the most common patterns, right? You'll start to recognize over and again, right? And this one, right? The fact that it's branching by a factor of two telling you that at the kind of base case, right? You're gonna expect to see a two to the n expansion.

So just to give you a little bit of some idea that you can use big O to predict things based on getting some run time performance for some small values of n and then what you'd expect to see for larger values. So I have three different algorithms here. One that was just logarithmic in terms of n, so it should divide the input in half and kind of work on it. Something like binary search actually fits that profile. Something that's linear. Something that's n log n and something that's n squared. Then for different values of n I've plugged in actually the first ones I ran and then actually I – some of the other ones I had to estimate because they were taking way too long to finish.

So that for an input of size ten, all of them are imperceptible. These are in terms of seconds here. Taking fractions of seconds. This is on my machine that is running at a couple gigahertz, so it's got about a million instructions per second. You up that by a factor of ten, right? And they're still kind of in the subsecond range, but you can see that from the n squared algorithm took a bigger jump, let's say, than the linear algorithm did, which went up by a factor of ten almost exactly. Going up by another factor of ten and, again, sort of climbing, right? 10,000, 100,000, a million, a trillion. You get to something that's like 100,000, right? And an algorithm that's linear, right? Is still taking a fraction of a second.

But something that was quadratic now has climbed up to take several hours. So even for inputs that don't seem particularly huge, right? The difference between having an algorithm that's gonna operate in linear time versus quadratic is gonna be felt very profoundly. And as you move to these larger numbers you have things that you just can't do, right? You cannot run an n squared algorithm on a trillion pieces of data in your lifetime. It's just too much work for what the input is. Clinton?

Student: Yeah. How did it go down for a billion from like a million? From –

Instructor (Julie Zelenski):Oh, you know what, that just must be wrong. You're totally right. That should definitely be over some. Yeah. Well log n should really be – I think that when I copied and pasted from these terminal window. So, of course, I must have just made a mistake when I was moving something across, but you're totally right. This should be going up, right? Ever so slightly, right? This algorithms going very, very slowly logarithmic function, right? Almost a flat line, but that definitely should be up a little bit, right? From where it is. When you get to a trillion, right? Even the linear algorithm is starting to actually take some noticeable time, right? But things that are logarithmic still running very, very slowly. So this is an example of binary search.

Binary search operating on a million or trillion items is still just in a flash, right? Telling you whether it found it or not. Doing a linear search on a trillion or billion is several minutes' worth of my old computers time. And so just another way to look at them, right? Is to think about what those things look like in terms of graphed on the Cartesian plane that a constant algorithm is just a flat line. It takes the same amount of time no matter how big the input is. It's pretty small values of n down here, so it just shows the early stages, right? But logarithmic, right? Almost a flat line itself. A little bit above constant, but very, very slowly growing. The linear scale, right? Showing the lines and then the n squared and to the n showing you that they're kind of heading to the hills very quickly here. So even for small values of n they are reaching into the high regions and will continue to do so as that will be more pronounced, right? As you move into the higher and higher values of n.

All right. I get to tell you a little bit about sorting before we go away. Because some of them I'm just setting the stage for, like, okay, how can we use this to do something interesting? Knowing about how to compare and then contrast. That it tells you something. Let's try to solve a problem, right? That lends itself to different approaches that have different algorithmic results to them that are worth thinking about. So sorting turns out to be one of the best problems to study for this because it turns out sorting is one of the things that computers do a lot of. That it's very, very common that you need to keep data in sorted order. It makes it easier to search. It makes it easier to find the minimum, the maximum, and find duplicates. All these sort of things, right? Like by having, keeping it inserted or it tends to be more convenient to access that way.

It also makes a bunch of other things, other properties about the data, easy to do. If you want to find the top ten percent, right? You can just pick them off, right? If they've all ready been sorted. You want to group them into buckets for histograming, right? All these things actually are enabled by having them all ready be in sorted order. So it turns out that a lot of times that data that you get from other sources tends to first need to be put in sorted order before you start doing stuff on it. Okay.

Well, there's lots of different ways you can sort it. There are simple ways. There are complicated ways. There are fancy ways. There are ways that are dumb, but easy to write. Ways that are smart, but hard to write. And everything in between. So we're gonna be looking at it in terms of sorting vectors because that's probably the most common data structure that needs the sorting. But you can also imagine taking the same kind of algorithms and applying them to different kinds of data structures you have. Like starting a linked list. Okay.

So I'm gonna show you a sorting algorithm. It's probably the simplest and the easiest to write. If somebody – if you were stuck on a desert island without a textbook, but you happen to have a compiler and you needed to write a sorting algorithm to get your way off the island. It comes up all the time. This is probably the algorithm you're gonna use. It's called selection sort. And the idea behind selection sort is that it's going to select the smallest element and put it in the front. So if I have a big stack of test papers and I want to sort them in order of score, then I'm gonna go through there and find somebody who

got the absolute lowest score. It's said, but true. Somebody had to be there. So I kind of work my through it and maybe I hold my finger on the one that I've seen that looks smallest so far.

So this one's a 40, oh, well, this one's a 38. Okay. Oh, look there's a 35. Oh, look, there's a 22. Nobody gets these scores in my exams, I'm just kidding. And then finally get down to the bottom. You say, okay, that 25, that was the smallest. I have a hold of it, right? And I'm gonna basically take that and bring it to the front. And in terms of managing a vector that actually there's a slight efficiency to be had by instead of kind of pulling it out of the stack and sliding everything down I'm actually just gonna swap it with the one that's in the very front. So whoever was the current first one is gonna booted to pull in this one and I'm gonna put them back where this other one came from.

So I have moved the very smallest to the top of the vector. Then I just do the same thing again, but now excluding that smallest one. So I've all ready seen that one and kind of set it aside. I start looking at the remaining n minus one and I do the same thing again. Find the smallest of what remains, kind of just walking through, going to find it, and swap it down into the second slot, and so on. A little bit of code. Doing it kind of the way I described it, right? Of tracking the minimum index, right? Looking from here to the end, so this four loop in the middle here starts at the current position and only considers from here to the very end of the vector and then finds any element which is smaller than the one I'm kind of currently holding my finger on and then when I'm done after that inner loop has executed I will know what the min index is and then there's a spot function here that just exchanges those two things.

The I slot now gets replaced with the one at min index and, again, exchanged in the contents of the array. I just do that n minus one times and I will have done it all. I'd like to show you animation, but I guess I'll just wait for that until Wednesday. We will watch it doing it's kind of thing in progress and you can kind of be thinking about for Wednesday what other ways you might be able to sort data than just this strategy.

[End of Audio]

Duration: 50 minutes

ProgrammingAbstractions-Lecture15

Instructor (Julie Zelenski):Hey. Welcome to Wednesday. Things that have happened and we're talking about sorting. We've got lots of sorting to talk about today. Some more sorting we'll talk about on Friday. That will be covered in Chapter 7 there. And then things that are happening. Boggle is coming in this Friday, so hopefully you're making good progress on Boggle. How many people just totally done with Boggle? All done, all behind you? Oh, that's a very healthy number. Like to see that. I did put a note here about Boggle Wednesday. Boggle is due Friday and given our late day policy we actually count days class meets and since Monday's a holiday that technically means if you were to handle Boggle late it is due on Wednesday.

Here's something I also want to say while I say that. Don't do it. It's a bad idea. You need to prep for the mid-term. I'm not gonna hand out assignments on Friday. I'm gonna just try to clear your plate for getting your head together for the mid-term. I think if you were finishing a late Boggle in that period you're really short changing yourself. The mid-term counts a lot more, kind of covers more ground, is more important in the big scheme of things than working on a late assignment. So do make your plans to get it done on Friday so that you can actually not have it hanging over your head when you're trying to get ready for that mid-term next Tuesday evening. Terman Auditorium, which is a little hike away from here over there in the Terman Building, 7:00 to 9:00 p.m. will be the big party zone.

Today I put up the solution to the problems I gave you on Monday, so you have both of those to work on. I should encourage you not to look at them kind of in parallel. I think that's one of the bad ideas to say. Look at the problem, look at the solution, and say yeah, that looks right. They are right, actually. I'll tell you that. You don't need to do that. What you need to do is say good, I have generated that and the best way to check whether you've generated it is not to say yeah, that looks like what I would have done is to do it, right? Do it on the paper without the solution around and then look at your answer relative to the solution and see if there's something you can learn about how our solution, your solution aren't exactly the same. There's a lot of partial credits. It's not like you need to be able to reproduce and it's not like there's only one correct answer, but, sort of, that practice is the best practice for the exam.

Getting ready to think about how to do it on paper. The other handout also very worthwhile reading. It's kind of long, but it has a lot of information that the section leaders and students over time have kind of gathered their wisdom and tried to write down just about how to make sure you do well on the exam. I think it's one of the more challenging aspects is to try to come up with the fair way to give an exam in a class that is largely experience and skill based like [inaudible] is that we can capture an assessment of how you're doing, right? But in this funky format of being on paper and stuff. So definitely do read that handout for, sort of, our advice and thoughts on how to make sure that you do succeed in that endeavor. Anything administratively you want to ask or talk about? Question?

Student: So if you have a conflict with the mid-term and have all ready filled [inaudible] when will we hear back on that?

Instructor (Julie Zelenski):So we're gathering them all by Thursday and then we're gonna try to do this big intersection. So either late Thursday or, sort of, Friday is when we're kind of sort out all the possibilities and try to come up with a schedule that gets everybody taken care of. So you should hear by the end of this week you will know what's going on. All right.

So I'm going to pick up on where I left off on Monday. And this was the code for the selection sort algorithm. And, in fact, I'm actually not gonna – I'm gonna be showing the code for each of the algorithms that I'm using here, but I'm gonna encourage you to not actually get really focused on the details of the code. This is actually one of those places where the code is an embodiment of the idea and the idea is the one that actually we're interested in studying, which is the approach it takes to sorting, how it decides to get that data in sorted order, and then how that algorithm performs and the kind of details of where it's a plus one and minus one less than – it's actually a little bit less important for this particular topic.

So what I'm gonna show you actually is a little demo of this code in action that's done using our graphics library just to show you what's going on. And so this is the selection sort code that the outer loop of which selects the smallest of what remains from the position I to the end of the array and then swaps it into position after it's done that finding. So this inner loop is basically just a min finder finding the minimum index element from I to the end. So I'm gonna watch this code step through. So you can see as it moves J down, which goes bopping by in green it's actually updating this kind of min, the blue pointer there, that says, well, what's the min I'm seeing so far?

Early on, it was 92, then it bumped down to 45, then it bumped down to 41, then 28, then 8 and finally got to the end and found nothing smaller so it says, okay, that must be where the min is. Let me move that guy to the very front. So exchanges those two guys, flopping another one back in there. Now, we know we have the very smallest one out of the way. It's in the right spot. We have nothing more to do with that. There's no changes that'll need to be made to anything to the left of I and we do the same operation again. Select from the remaining n minus one elements, the minimum element. So watch the green, sort of, finger move down while the blue finger kind of stays behind on the smallest it's seen so far. So it's okay, so, well, then 15 is it. Swap that guy up into the second slot.

Do it again. Keep doing this, right? Working it's way down and then, as you see, the array's kind of going from the left to the right, the smallest most elements, right? Being picked off and selected and moved out and then leaving the larger ones kind of over here in a little bit of a jumbled mess to get sorted out in the later iterations of those loops. So there is select and sort in action, right? Doing its thing. Okay. Let me do another little demo with you because I have – there's lots of ways to look at these things. I'm gonna do one that actually can do slightly bigger ones and so I'm gonna set this guy up and it's

gonna show, in this case, the red moving, right? And it holding onto the smallest it's seen and updating as it finds smaller ones. Move it to the front and then I'm actually gonna speed it up a little bit which is, let's see, go a little faster.

So you'll note that it took in this case and it does a little bit of counting. I'm gonna look at the analysis in a minute about a lot of comparisons and not so many moves. That's actually kind of the model behind selection sort is to do a lot of testing. That first iteration looks at every one of the n elements to find the smallest and then it does one move to pull it into position. Then it does the same thing again. Does another task looking at this time only n minus one elements remain, but doing another full set of comparisons to decide who is the smallest of what remains and then swapping that to the front. So it's taking a strategy that seems to indicate that comparison, right? Which it's gonna do a lot of, a very few number of moves as a result, because it identifies where something goes.

It pulls it out, puts it to the front, and doesn't do a lot of shuffling around. If I've put kind of a higher number of things in here and let it go it appears to kind of move very slowly at first, right? Because it's doing a lot of work to find those small ones and then move them down to the front, but as it gets further along it will tend to kind of speed up toward the end and that's because in subsequent iterations, right? There's fewer of the elements remaining to look at and so kind of a tiny little portion of it's been sort of so far the first four. If I – I think if I let this go it will go way too fast. Yeah. Way too fast to see it.

There's a lot of things I want to show you because I'm gonna add in the option for sound. So one thing to learn a little bit about how to visualize sorts, right? I think that this kind of technique sometimes it's easier to look at than the code is to see how it's putting things in order. Several years ago I had a student who actually was blind and these visualizations do squat for someone who's blind it turns out. So she helped me work out some ideas about how to do a different, sort of, interface. One that visualizes it using it when you're different senses would just do sound and so the change I've made to the code here will play a tone each time it's moving an element and the frequency of that tone is related to the height of the bar. So the smaller height bars are lower frequency, so lower tones.

And then the taller bars have a higher frequency, higher pitch to them. And so you'll hear the movement of the bars being expressed with this tone loop so – and I'm not hearing any sound. Is my – are we getting sound? There we go. I'm not getting sound now, how about you? No, we're not getting any sound at all. Here we go. I'm getting sound now. How about you?

Well, and that's true. It's only when they move. That's a very good point. What about – I'm running a little bit slower than I'm thinking. All right. I can get it to – all right. Let's try. Now, we're – I feel like we're hearing something. We're still. Definitely getting sound now. Okay. Now it's gonna be a little loud. There we go.

The march of the low order tones can be moved into place doing a lot of comparison and then you'll hear kind of the speed up I talked about. Toward the end is the later iterations.

Finishing it up. So giving you a kind of a little memory for, sort of, what's the action selection sort. The small amount of tones and the kind of sparse distance meeting shows that it's doing a lot of work in between each of those moves, right? And then the fact that the tones kind of advance from low to high tells you that it's working on the smaller values up to the larger values. Kind of working it's way to the end and that speeding up it closing in on the remaining ones as it gets to the end.

Left that around because even if you are can see it you can also hear it and that may help something. What we're gonna look at is how much work does it do? So if I go back and look at the code just to refresh what's going on, right? This inner loop is doing the comparisons all the way to find the min elements and so this is gonna iterate n minus one times on the first iteration, then n minus two, then n minus three, and all the way down those final iterations, three to look at, two to look at, one to look at and then one swap outside that. So what I actually have here is the sum of the values n minus one compares, plus one swap, but as two comparisons one swap, so one just swaps but actually the terms I'm looking at here are the number of comparisons the sum of the numbers one to n minus one is what we're trying to compute here.

Then if you've seen this some are the [inaudible], some are the Gaussian, and some you may all ready know how to solve it, but I just kind of showed you how to do the math to work it out, which is the term you're looking for is this sum. If you add it to itself, but rearrange the sequence of the terms so that they cancel each other out you'll see that the n minus one plus one gives you an n and that what you end up with is n minus one n's being added together is what the sum of this sequence against itself is and so we can divide by two to get the answer we're looking for, so we have a one-half n squared minus n term. Which in the big O world, right? Just comes down to n squared.

So tell this – it's a quadratic sort, right? That we would expect that if it took a certain amount of time to do a hundred, three seconds, then if we double that input we expect it to take four times as long. So if it was three seconds before it's 12 seconds now. So growing as a parabola is a fairly sharp steep curve to get through things. All right. So let me give you an alternative algorithm to this. Just to kind of think, that's gonna be the theme is, well, that's one way to do it and that has certain properties. Let's look at another way and then maybe we'll have some opportunity to compare and contrast these two different approaches to see what kind of tradeoffs they make in terms of algorithm choices.

So the way you might handle a deck of a cards – sorting a deck of cards. So if I'm handing out cards to people you're getting each card in turn that one way people do that is they pick up the first card and they kind of – you assume it's trivially sorted, right? It's in the right place. You pick up the next card and you decide, well, where does it go relative to the one you just had. Maybe you're just sorting by number order and you say, well, okay, it's greater than it and goes on this side. You pick up your next one and it's like it goes in between them. And so you're inserting each new element into the position of the ones that are all ready sorted.

So if you imagine applying that same thing in terms of computer talk in a vector sense is you could imagine kind of taking the vector, assuming that the first element is sorted, then looking at the second one and deciding, well, where does it go relative to the ones all ready sorted? The ones to my left and kind of moving it into position, shuffling over to open up that space that's gonna go into and then just extending that as you further and further down the vector. Taking each subsequent element and inserting it into the ones to its left to make a sorted array kind of grow from the left side. So it grows from the left somewhat similar to selection sort, but actually it will look a little bit different based on how it's doing its strategy here.

So let me give you the insertion sort code. So my outer loop is looking at the element at index one to the element of the final index of the raise side minus one and it copies that into this variable current to work with and then this inner loop is doing a down to loop, so it's actually backing up from the position J over starting from where you are to say, well, where does this – it keeps sliding this one down until it's fallen into the right place. So we'll move over the 92, in this case, to make space for the 45 and then that's kind of the whole iteration on that first time. The next time we have to look at 67. Well, 67's definitely gonna go past 92, but not past 45. 41 is gonna need to go three full iterations to slide all the way down to the front. 74 is just gonna go over one number. Just needs to slide past one.

So on different iterations, right? A little different amount of work is being done, right? This loop terminates when the number has been slotted into position. We won't know in advance how far it needs to go, but we go all the way to the end if necessary, but then kind of sliding each one up by one as we go down. So that one moved all the way to the front, 87 just has one to go, eights got a long way to go. All the way down to the very front there. Sixty-seven goes and stops right there, 15 almost to the bottom, and then 59 moving down this way. So kind of immediately you get the sense it's actually doing something different than selection sort, just visually, right? You're seeing a lot more movement for a start that elements are getting kind of shuffled over and making that space so it's definitely making a different algorithmic choice in terms of comparison versus moving than selection sort did, which is kind of a lot of looking and then a little bit of moving.

It's doing kind of the moving and the looking in tandem here. If I want to hear how it sounds, because nothing's complete without knowing how it sounds, I can go back over here and let me turn it down a little bit. So you hear a lot more work in terms of move, right? Because of the signal I'll speed it up just a little bit. So, as you can see, a kind of big [inaudible] a lot of work [inaudible] as it slides that thing down into position and then finding it's home. Some will have come a long distance to travel like that last one. Other ones are very quick where they don't have so far to go in turn. I think that's it. I'll crank it up to like a, sort of, a bigger number, let's say, and turn off the sound and just let it go and you can kind of get a sense of what it looks like.

It seems to move very, very quickly at the beginning and then kind of starts to slow down towards the end. If I compare that to my friend the selection sort, but it looks like

insertion sort is way out in front and its gonna just win this race hands down, but, in fact, selection sort kind of makes a really quick end runaround and at the end it actually came in just a few fractions of a second faster by kind of speeding up towards the end. So it's like the tortoise and the hare. It looks like insertion sort's way out in front, but then actually selection sort manages to catch up. I'm gonna come back and look at these numbers in a second, but I want to go back and do a little analysis on this first.

If you take a look at what the code is doing and talk about kind of what's happening, right? We've got a number of iterations of the outside, which is sliding me to this position. This inner loop is potentially looking at all of the elements to the left. So on the first iteration it looks at one. At the second iteration it looks at two. The third one three and so on until the final one could potentially have n minus one things to examine and move down. If that element was the smallest of the ones remaining it would have a long way to travel. But this inner loop, right? Unlike selection sort that has kind of a known factor it's actually a little bit variable because we don't know for sure how it's gonna work.

Potentially in the worst case, right? It will do one comparison move, two and three and four all the way up through those iterations, which gives us exactly the same Gaussian sum that selection sort did. One plus two plus three all the way up to n minus one tells us its gonna be n squared minus n over two, which comes down to O of n squared. That would be in the absolute worst case, right? Situation where it did the maximum amount of work. What input is the embodiment of the worst case in selection sort? What's it gonna be? If it's flipped, right? If it's totally inverted, right? So if you have the maximum element in the front and the smallest element in the back, right? Every one of them has to travel the maximum distance in this form to get there.

What is the best case to do the least amount of work? It's all ready sorted. If it's all ready sorted then all it needs to do is verify that, oh, I don't need to go at all. So it actually – the inner loop only does one test to say do you need to move over at least one? No. Okay. So then it turns out it would run completely linear time. It will just check each element with its neighbor, realize it's all ready in the right place relative to the left, and move one. So, in fact, it will run totally quickly, right? In that case. And in the average case you might say, well, you know, given any sort of random permutation of it, each element probably has to go about half the distance. So I don't have to go all the way, some don't have to go very far, some will go in the middle. You would add another factor of a half onto the n squared, which ends up still in the big O just being lost in the noise.

You say, well, it's still n squared. But it probably does. It is a little bit less of an n squared than selection sort. So if I go back here to my – well, it was counting for me. That the mix of operations between insertion sort and selection sort, so that's selection sort on the top and insertion sort that's beneath it, shows that selection sort is doing a lot of compares. In this case, I had about 500 elements and it's doing about n squared over two of them, 250,000 divided by two there, and so it really is doing a full set of compares. A whole n squared kind of compares is doing a small number of moves. In this

case, each element is swapped so there's actually two moves. One in, one out. So it does basically a number of moves that's linear relative to the number of elements.

The insertion sort though is making it, sort of, a different tradeoff. It's doing a move and a compare for most elements, right? In tandem and then that last compare doesn't do a move with it. So there should be roughly tracking in the same thing. But they look closer to n squared over four. Showing that kind of one-half expected being thrown in there. But in the total, the idea is that it does about 100,000 compares a move. This one does about 100,000 compares that when you look at them in real time they tend to be very, very close neck in neck on most things. So if I do this again giving it another big chunk and just let it go, again, it looks like insertion sorts off to that early lead, but if you were a betting person you might be putting your money on insertion sort. But selection sort is the ultimate comeback kid and toward the end it just really gets a second wind.

In this case, beat it by a little bit bigger fraction this time. If I put the data into partially sorted order. So now, in this case, I've got about half of the data all ready where it's supposed to be and another half that's been randomly permuted and now let it go. Insertion sort takes an even faster looking early lead and selective sort, in this case, never notices that the data was actually in sorted order. But time actually is a little bit artificial here and, in fact, the number of comparisons is maybe a better number to use here, but it really did a lot more work relative to what insertion sort did because insertion sort was more able to recognize the sortedness of the data and take advantage of it in a way that selection sort totally just continued doing the same amount of work always.

That's kind of interesting to note, right? Is that when we're talking about all these different sorting algorithms, right? That we have multiple evidence, because actually they really do make different tradeoffs in terms of where the work is done and what operation it prefers and what inputs it actually performs well or poorly on. That selection sort is good for it does exactly the same amount of work no matter what. If it's in sorted order, or reversal order, or in random order, right? It always is guaranteed to do the kind of same amount of work and you don't have any unpredictability in it. That's both an advantage and a disadvantage, right? On the one hand it says, well, if you knew that you needed this sort to take exactly this time and no better, no worse would be fine then actually having that reliable performer may be useful to know.

On the other hand, it's interesting to know that insertion sort if you gave it to data that was almost sorted then it would do less work, right? Is an appealing characteristic of that. And so having there be some opportunity for it to perform more efficiently is nice. The other thing, also, is about this mix of operations. Whether it considers comparisons or moves are more expensive operation. For certain types of data those really aren't one-to-one. That a comparison may be a very cheap operation and a move may be expensive or vice versa depending on kind of the data that's being looked at.

For example, comparing strings is often a bit more expensive than comparing numbers because comparing strings has to look at letters to determine when they distinguish. If you had a lot of letters in the front that were overlapping it takes, sort of, more work to

distinguish at what point they divide and which one goes forward. On the other hand, moving a large data structure if it were a big structure of student information, right? Takes more time than moving an integer around. So depending on what the data that's being looked at, there may actually be a real reason to prefer using more comparisons versus moves. And so examples I often give for this are like if you think about in the real world, if you were in charge of sorting something very heavy and awkward like refrigerators you kind of line up the refrigerators by price in the warehouse or something.

You would probably want to do something that did fewer moves, right? Moves are expensive. I'm picking up this refrigerator and I'm moving, I don't want to move it more than once, right? And so you might want to go with the selection sort where you go and you figure out who's the cheapest fridge, let me pull that one and get it over here, and now I'm not gonna touch it again rather than kind of sitting there with your insertion sort and moving the fridges one by one until you got it to the right spot. But if I were in charge of finding out who was, let's say, the fastest runner in the mile in this class you probably would not enjoy it if my strategy were be to take two of you in a dead heat and say run a mile and see who won. And now whoever won, okay, well, you get to run against the next guy. And if you win again you get to run against the next guy.

Like you might just say hey, how about you let me get ahead of that person if I beat them once. I don't want to have to go through this again. Like fewer comparisons would certainly be preferred. So both of these, I would say, are pretty easy algorithms to write. That is certainly the strength of selection and insertion sort. They are quadratic algorithms which we're gonna see is not very tractable for large inputs, but the fact that you can write the code in eight lines and debug it quickly and get it working is actually a real advantage to the – so if you were in a situation where you just needed a quick and dirty sort these are probably the ones you're gonna turn to.

So just some numbers on them, right? Is 10,000 elements on my machine kind of an unoptimized contact was taking three seconds. You go up by a factor of two. We expected to go up by about a corresponding factor of four and the time it roughly did. Going up by another factor of two and a half again going up. By the time you get to 100,000 though, a selection sort is slow enough to really be noticeable. It's taking several minutes to do that kind of data and that means if you're really trying to sort something of sufficiently large magnitude a quadratic sort, like insertion sort or selection sort, probably won't cut it.

So here's an insight we're gonna kind of turn on its head. If you double the size of the input it takes four times as long. Okay. I can buy that, right? I went from ten to 20 and went up by a factor of four. So going in that direction it feels like this growth is really working against you, right? It is very quickly taking more and more time. Let's try to take this idea though and kind of turn it around and see if we can actually capitalize on the fact that if I have the size of the input it should take one quarter the amount of time. So if I had a data set of 100,000 elements and it was gonna take me five minutes if I try to sort it in one batch. If I divided it into two-50,000 element batches it will take just a little over a minute to do each of them.

Well, if I had a way of taking a 50,000 sorted element and a 50,000 sorted input and putting them back together into 100,000 combined sorted collection and I could do it in less than three minutes, then I'd be ahead of the game. So if I divided it up, sorted those guys, and then worked them back together and if it didn't take too much time to do that step I could really get somewhere with this. This kind of turning it on its head is really useful. So let's talk about an algorithm for doing exactly that. I take my stack. I've got a stack of exam papers. Maybe it's got a couple hundred students in it. I need to just divide it in half and I'm actually gonna make a very – the easy, lazy decision about dividing it in half is basically just to take the top half of the stack, kind of look at it, figure out about where the middle is, take the top half and I hand it to Ed.

I say, Ed, would you please sort this for me? And I take the other half and I hand it to Michelle and I say would you please sort this for me? Now, I get Ed's stack back. I get Michelle's stack back. They're sitting right here. All right. So that was good because they actually take a quarter of the time it would have taken me to do the whole stack anyway. So I'm all ready at half the time. What can I do to put them back together that realizes that because they're in sorted order there's actually some advantage to reproducing the full result depending on the fact that they were all ready sorted themselves?

So if I look at the two stacks I can tell you this, that someone over here starts with Adams and this one over here starts with Abbott. The very first one in the output has to be one of the two top stacks, right? They can't be any further down, right? This is the sorted left half. This is the sorted right half, right? That the very first one of the full combined result must be one of those two and it's actually just the smaller of the two, right? So I look at the top two things. I say, Abbott, Adams, oh Abbott proceeds Adams. Okay. Well, I take Abbott off and I stick it over there. And so now that exposes Baker over here. So I've got Adams versus Baker. And I say, oh, well, which of those goes first? Well, Adams does.

I pick Adams up and I stick it over there. So now that exposes, let's say, Ameliglue, my old TA. Ameliglue and Baker. And I say, oh, well, which of those? Ameliglue. And at any given point where there's only two I need consider for what could be the next one and so as I'm doing this, what's called the merge step, I'm just taking two sorted lists and I'm merging. So I'm kind of keeping track of where I am in those two vectors and then taking the top of either to push onto this collection I'm building over here and I just work my way down to the bottom. So that merge step, right? Is preserving the ordering. Just kind of merging them together into one sorted result.

If I could do that faster then I could have actually of sorted them and then I'm actually ahead of the game and there's a very good chance for that because that, in fact, is just the linear operation, right? I'm looking at each element, deciding, and so I will do that comparison and time to decide who goes next, right? I look at this versus that and I put it over there. I look this versus that and I put it over there. Well, I'm gonna do that until this stack has n things in it. All of them have been moved. So, in fact, it will take me n comparisons to have gotten them all out of their two separate stacks and into the one together. So it is a linear operation to do that last step and we know that linear much quicker to get the job done than something that's quadratic. Yeah?

Student: When you merge the halves are you taking the higher one in the alphabet or the lower one?

Instructor (Julie Zelenski):Well, it typically I'm taking it in the order I'm trying to sort them into, right? Which is increasing. So I'll typically start at A's and work my way to Z's, right? So it turns out it's completely –

Student: It doesn't really matter.

Instructor (Julie Zelenski):It doesn't really matter is the truth, but the – whatever order they're sorted in is the order I'm trying to output them in. So, in fact, these are the – if they're first on the sorted piles then they'll be first on the alphabet pile. So whatever that first is. So I'm gonna actually go – look at the code for a second before we come back and do the diagramming. We'll go over here and look at the stepping part of it. So this is the code that's doing merge sort and so it has a very recursive flavor to it. It does a little calculation here at the beginning to decide how many elements are going to the left and going to the right.

As I said, it does no smart division here. So this is called an easy split hard join. So the split process is very dumb. It says figure out how many elements there are, divide that in half, take the one from zero to the n over two and put them in one separate subvector and take the ones from n over two to the n and put them in a second subvector and then recursively sort those. So let's watch what happens here. Computes that and says, okay, copy a subvector of the first, in this case four elements, copy a subvector that has the second four elements, the second half. So I've got my left and my right half and then it goes ahead and makes a call under merge sort, which says merge that left half and merge that right half and then we'll see the merge together.

So I'm gonna watch it go down because the thing that's gonna happen is when I do a merge of this half it's kind of like it postponed these other calls and it comes back in and it makes another call, which does another division into two halves. So taking the four that are on the left and dividing them into two and two and then it says, okay, now I merge the left half of that, which gets us down to this case where it divides them into one on the left and one on the right. And then these are the ones that hit my base case. That an array of size one is trivially sorted. So, in fact, the merge sort never even goes into doing any work unless there's at least two elements to look at.

So when it makes this call to the merge the 45 it will say, okay, there's nothing to do. Then it'll merge the 92 and works for the 92, which also does nothing. And now the code up here's gonna flip. So be warned about what's gonna happen here is I'm gonna show you what the process of the merge looks like, which is gonna do the left-right copy to the output. So this code looks a little bit dense and, again, this is not the time to get really worried about what the details of the intricacies of the code are. I think you really want to kind of step back and say conceptually the process I described of taking them off the two piles and then merging them is very much the take home point for this. And so what this

upper loop is doing is it's basically saying well, while the two stacks, the two piles, the two halves, whatever, each have something left in them.

Then compare them and take the smaller one off the top. So it's keeping track of all these indices, right? The indices of the left subarray, the indices of the right subarray and the indices of the right output array and it's actually kind of – and each step is putting a new one, copying one from one of the left or the right onto the output. And so that upper loop there said is 45 less than 92? It is, right? So it copies 45 and then at this point, right? There's nothing left on the left, so it actually drops down to those last two pieces of code, which, actually, do to copy the remainder from if there's only one stack left then just dump them all on the end. So we'll do the dump of the end of the 92. And so I've reassembled it. And so when I get back to here then I have merge sorted the left side and then I go through the processes of merge sorting the right side.

Which copies the 62 and the 41 down to the things and then does a merge back. Let me get to the stage where I'm starting to do a little bit bigger merges. So here I am with indices on my left and my right side and then my output index and it's like, okay, is 45 less than 41? If so then take 41 here and kind of advance P2 over and still see the 41 go across and it moved both the P and the P2 up an index to say, okay, now we're ready to pick the next one. So it looks at P1 versus P2's, decides it's pulling from the left side this time, and then now it still has the last most members of the two piles to look at, takes from the right side, and then we'll just dump the end of the other side.

So then in going through this process to do it all again, kind of break it all the way down. So it's just using recursion at every stage. So at the small stages it's a little bit hard to figure out what's going on, but once it gets back to here, right? It's just doing the big merge, let it go a little too fast there, to build it back up. So there's one thing about merge sort I should mention, which is merge sort is using extra storage. And I'm gonna get to a stage where I can explain why that's necessary. But selection sort and insertion sort both work with what's called in place and that means that they are using the one copy of the vector and just rearranging things within it so it doesn't require any auxiliary storage to copy things over and back and whatnot. That it can do all the operations on one vector with a little bit of a few extra variables around.

That merge sort is really making copies. That it divides it into these two subarrays that are distinct from the original array and copies them back. So it copies the data away and then it copies it back in. And the reason for that actually is totally related to what's happening in this merge step. That if I had – well, actually this is not the step that's gonna show it. I'm gonna show it on this side. That if it were trying to write over the same array – oh, now I made it go too fast and that was really very annoying. All right. Let me see if I can get it one more time. Do what I wanted it. Is that when it's doing the copying, if it were copying on top of itself it would end up kind of destroying parts of what it's working on. Oh, it's – I see what. We're gonna get this mistake the whole time. Okay.

Is that as it was doing the copy, right? If it's – if this really were sitting up here and this really were sitting up her and if we were pulling, for example, from the left side we

would be overriding something that was all ready – pulling from the right side, I'm sorry. We'd be overriding something that was on the left side. And so if it did that it would have to have some other strategy for then, well, where did it put this one that it was overriding? Like if it's pulling from the left side it's actually fine for it to override in the output array, but otherwise, right? It would be writing on something it was gonna need later. So the easiest thing for merge sort to do is just to move them aside, work on them in a temporary scratch base, and then copy them back. There are ways to make merge sort in place, but the code gets quite a bit more complicated to do that.

So your standard merge sort algorithm does use some auxiliary storage that's proportional to the size of the array. So that ends up being a factor in situations where you're managing a very large data set where, in fact, maybe it requires, right? A large amount of memory all ready for the ones that making a duplicate copy of it to work on may actually cost you more than you can afford. So merge sort sometimes is ruled out just because of its memory requirements despite the fact that it has a performance advantage over insertion sort and selection sort. What does it sound like though? That's what you really want to know. So let's first just watch it do it's thing and so you'll see it kind of building up these sorted subarrays, kind of working on the left half, and then postponing the right and coming back to it and so you'll see little pieces of it and the little merge steps as it goes along.

And then you can get a little glimpse of kind of the divisions down there. Let me actually run it again in a little bit bigger thing. And I'll turn this on in just a second. Just like as it goes faster and faster let me make my sound go and we'll see how much we can stand of it. It's pretty noisy you'll discover. Volume, good volume? Oh, yeah. Yeah. A lot of noise. Oh, yeah. Okay. So you definitely get the kind of 1960's sound tone generation there, but you get this, I mean, you can hear the merge step, right? Is very distinguishable from the other activity.

It's doing the kind of smaller and smaller subarrays it sounds just a little bit like noise, but as you see the larger subarrays being joined this very clear merging sound emerges from that that you can hear and see that in the very end, sort of, one big long merge of taking the two piles and joining them into one. A lot of noise, right? So that should tell you that there is a pretty good amount of movement going on. Question?

Student:[Inaudible]

Instructor (Julie Zelenski):It's merging them after they've been sorted. So you'll – when you see a merge you'll always see two sorted subarrays being joined into one larger sorted subarray. So at the very end, for example, you'll have these two sorted halves that are being merged down.

Student: And how much is the sorting [inaudible]?

Instructor (Julie Zelenski):Well, it's recursion, right? It's like it sorts the half, which sorts the quarters, which sorts the A's. And so at any given point what it really looks like

it's sorting is these little one element arrays, which are being merged into a two element array. Like all the work is being done in merging really. That sorting is kind of funny. When does it actually sort anything? Like never actually. It only merges things and by dividing them all the way down into all these one element arrays the merging of them it says, well, here's these trivially sorted things. Make a two element sorted thing out of them and that you have these two element things, merge them into one-four element thing, which you merge into an eight element thing and all the way back. So all of the work is really done in the merge step.

Kind of on the sorting angle it actually is deferring it down to the very bottom. So that's a kind of odd thing to do, right? It really just divides it all up into one – a hundred piles, each of size one and then kind of joins those into one pile, these into one pile, these into one pile, and so now I have 50 piles, right? That are each a size two. Now I join them into 25 piles of size four and then 12 piles of size eight and so on. So this is the code for the outer point of the merge algorithm. I'm actually not gonna look at the merge algorithm very in depth. I'm really more interested in the kind of outer point of this algorithm and so the steps that are going on here is being a little bit of constant work. Let me actually – we do a copy operation that copies out the left half and the right half.

Both of those operations together require linear time. So I've looked at every element and I've copied the first half onto something, the second half onto something. So that took – I looked at every element in that process. I make two calls on the left and the right side and then I do a merge on the end. We talked earlier about how that was linear and the number of elements because as I build the two piles into one sorted pile every element is touched in that process as it gets chosen to be pulled out. So linear divide and a linear join and then there's this cost in the middle that I'd have to kind of sort out what's happening, which is there's sort of n work being done at this level, but then I make two recursive calls that should take time n over two.

So the input they're working on is half, again, as big as the one I have. How much time do they take? Well, there's my recurrence relation that allows me to express it. T of n is n, so at this level this kind of represents the copy step and the merge step at this level plus the work it took to get the two halves in sorted order. So I'm gonna show you a slightly different way of looking at recursive analysis just to kind of give you different tools for thinking about how to do this. I showed you last time how to do the repeated substitution and generalization. The pattern – I'm gonna show you this way to do it with a little bit of tree that kind of draws out the recursive calls and what's happening in them. That the merge sort of an input of size n, does n work at that level?

The copy in the merge step, right? For there plus it does two calls to merge sort of n over two. Well, if I look at each of those calls I can say, well, they contribute an n squared and an n squared on each side. So this one has n squared copy and merge. This one has n squared copy and merge and so, in effect, I have another n squared plus itself there and then this level, right? Looks at the n over four, which has four n over four components. So that actually at each level in the tree that every element, right? Is being processed in

one of those subcalls across it. So every element is up here and they're in two subgroups and then four subgroups and then eight subgroups.

And that each element is copied and merged in its own subgroup at each level of the recursion. So that kind of gives me this intuition that there's n work being done on every level, every element copied and merged as part of that there. And so then what we need to complete this analysis is just to know how deep this tree grows. How far we get down in the recursion before we hit the base case. So we're dividing by two each time. N over the two, over two to the second, to the third, to the fourth, and so on. So at any given level K down, right? We have n divided by two to the K. What we want to solve for is where n over two to the K equals one. So we've gotten to this smallest case where we have those trivially sorted one element inputs to look at and so we just do a little math here, rearrange that, right? Divide by two to the K or multiply by two to the K both sides when n equals two to the K. Take the log base two of both sides and it will tell me that K is log base two of n.

That I can divide n by K by two K times, where K is the log base two of n, before it bottoms out with those one element vectors. So log n levels, n for level tells me the whole thing is n log n. So n log n is a function you may not have a lot of intuition with. You may not have seen it enough to kind of know what it's curve looks like, but if you remember how the logarithmic curve looks, right? Which is a very slow growing almost flat line and then n being linear it – the kind of combination of the two it's called the linear rhythmic term here is just a little bit more than linear. Not a lot more, but it grows a little bit more steeply than the state of linear was, but not nearly, right? As sharply as something that's quadratic.

So if we look at some times and let selection sort compared to merge sort, right? On an input of 10,000, right? Took a fraction, right? To do a merge sort than it did to do a selection sort and as it grows, right? So if we go from 20,000 to 50,000 we've a little bit more than doubled it that the merge sort times, right? Went up a little bit more than a factor of two in growing in these things. Not quite doubling. A little bit more than doubling, right? Because of the logarithmic term that's being added there, but growing slowly enough that you can start imaging using a sort like merge sort on an input of a million elements in a way that you cannot on selection sort, right?

Selection sort – my estimate I did not run it to find this out, right? Based on the early times I can predict it'll be about eight hours for it to sort a million elements, taking just a few seconds, right? For merge sort to do that same input. So a very big difference, right? From the n square to n log n that makes it so that if you have a sufficiently large data set, right? You're gonna have to look to an n log n sort where an n squared sort would just not work out for you. I'll mention here that actually n log n is the theoretical boundary for what a general purpose sort algorithm can do. So that our search from here will have to be a quest for perhaps an n log n that competes with merge sort and maybe exceeds it in some ways by having lower constant factors to it, but we won't get a better big O for a general purpose sorting algorithm.

If we have to sort kind of any amount of data and any permutation we just – and it has to work for all cases, then n log n is the best we can do in terms of a big O. Let me show you these guys kind of run a little race because there's nothing more important than having a reason to bet in class. So if I put insertion and selection and merge sort all up against one another. Let's turn off the sound. I really don't want to hear it all go. Okay. So merge sort just really smoking and insertion sort and selection sort not giving up early, but definitely doing a lot more work. So if you look at the numbers here in terms of comparisons and moves that are being done in the merge sort case, right? So I have 500 elements here is substantially less, right? Than the quadratic terms that we're seeing on the comparison and move counts, right? For us in selection sort and insertion sort.

So this is still on something fairly small, right? Five hundred elements, right? That you get into the thousands and millions, right? The gap between them just widens immensely. Does merge sort make any good sense out of things being all ready sorted? So thinking about what you know about the algorithm, does the fact that it's mostly sorted or all ready sorted provide an advantage to the way merge sort works? Nope, nope. Just not at all, right? In fact, I can make the data actually totally sorted for that matter, right? And say go ahead and sort and see what happens, right? And it's like insertion sort did 511 comparisons, zero moves, realized everything was in sorted order and finished very quickly.

Merge sort still did a lot of work, thousands of comparison moves. It did a slightly fewer number of compares than it would typically do. That's because when it, for example, divided into the left half and the right half it had all the smaller elements on the left, all the larger elements on the right, and it will sit thee and compare to realize that all of the left elements go first. Then it'll kind of just dump the remaining ones on. So it actually shaves off a few of the comparisons in the merge step, but it doesn't really provide any real advantage. It still kind of moves them away and moves them back and does all the work. And then selection sort just still taking its favorite amount of time, which is, yeah, I'll look at everything. That might be the smallest, but I'm not taking any chances. I'm gonna look through all of them to make sure before I decide to keep it where it was.

If I put them in reverse order, just because I can, watch insertion sort bog down into it's worst case and that gives selection sort a chance to, like, show it's metal and there selection sort showing okay, well, you give me my absolute worst input I definitely do have a little bit of a hard time with it. But merge sort still just doing its thing. What is gonna make all the sound go on? Just because we can. Because we have two minutes and I'm not gonna sort quick sort. Ah, you're going. They're duking it out. Oh, no. That sounds like a cartoon, sort of, like a race. All right. I could watch this thing all day. In fact, I often spend all day. I show this to my kids. I'm still waiting for them to come up with the next great sort algorithm, but so far really it's not their thing.

So I will give you a little clue about what we're gonna do next. We're gonna talk about a different recursive algorithm. Same divide and conquer strategy kind of overall, but kind of taking a different tactic about which part to make easy and which part to make hard. So quick sort, right? As I said, is this easy split hard join. We divided them into half

using sort of no smart information whatsoever and then all of the work was done in that join. I've got these two sorted piles. I've gotta get them back into order. How do I work it out? Well, the quick sort algorithm is also recursive, also kind of a split join strategy, but it does more work up front that's not even in the split phase.

If I kind of did a more intelligent division into two halves then I could make it easier on me in the join phase. And it's strategy for the split is to decide what's the lower half and what's the upper half. So if I were looking at a set of test papers I might put the names A through M over here. So go through the entire pile and do a quick assessment of are you in the upper half or the lower half? Oh, you're in the lower, you're in the upper, these two go in the lower, these two go in the upper. I examine all of them and I get all of the A through M's over here and all of the N through Z's over there and then I recursively sort those.

So I get the A to M's all worked out and I get the N through the Z's all worked out. Then the task of joining them is totally trivial, right? I've got A through M. I've got N through Z. Well, you just push them together and there's actually no comparing and looking and merging and whatnot that's needed. That join step is where we get the benefit of all of the work we did in the front end. That split step though is a little hard. So we'll come back in on Friday and we'll talk about how to do that split step and then what is some of the consequences of our strategy for that split step and how they come back to get at us, but that will be Friday. There will be music. There will be dancing girls.

[End of Audio]

Duration: 50 minutes

ProgrammingAbstractions-Lecture16

Instructor (Julie Zelenski):There must be endless toil and travail. Let's ask another question. So [inaudible] was the biggest thing he had done so far. [Inaudible] on the trail. Not really anything new, but definitely the prospect of getting it all working and bigger aspects of its own [inaudible] challenges.

So I expect to take a little bit more, but I'm always open to figure out what really happened there. So let's, we'll start with something ridiculous, but just in case somebody was just super hot, did anybody manage to finish [inaudible] with less than ten hours, a total start to finish – hey I like to see that.

That's a very good sign. Ten to fifteen – a very healthy sized group there. Fifteen to 20 – there's some people there. More than 20; okay, good, good and hopefully when you got done you felt like, "Okay, I have accomplished something good," but there's something really satisfying about writing a program.

The best thing you can't do as well as the program you've just written, the idea that taking in your own brain and cracking into it to write a program that can actually play a game and beat you is really a pretty neat little twist on things.

So, hopefully that was an enjoyable and satisfyable place to get to right before the midterm here. We're gonna take a little break in assignments meaning I'm not giving you an assignment now because what I really want you to spend your weekend doing is enjoying the sunshine for five-ten minutes at a time and then prepping yourself for the midterm which is coming up Tuesday night.

So we don't have class on Monday and so the next time I'll see you is Tuesday, 7:00 to 9:00 over in [inaudible] Auditorium and certainly if you have questions that need answered, sending an email over the weekend is totally fine, and we're showing up in the Lair; we'll have Lair hours on Monday evening [inaudible].

Student:[Inaudible].

Instructor (Julie Zelenski):No, we're not. Try again. Okay, now we got my phone. So I will probably put up [inaudible] in advance of the midterms, so if you actually wanted to pick it up before the midterm or right after the midterm to get started on it, and it will be due the following week and it's a little bit lighter of a week than [inaudible] was just in terms of planning.

What we're going to pick up today is we're going to talk Quick Sort which is the last of the sorting algorithms I want to talk through and then I'm actually going to go through the process of picking one of the sources, in this case [inaudible] and turning it into a fully generic sorting template as kind of capitalizing on the futures of C++. So the material here is the last sections of Chapter 7, a little bit of material pulled out of Chapter 11, which is the client callback stuff, is there.

I was actually totally planning on going to the café today until I got a call from my son's preschool right before I came to class that says he has swollen lymph nodes and a fever, so I've got to go get a little guy after class, so I'm sorry that won't happen today, but hopefully we'll be back on track a week from today.

Okay. So this is the last thing we had talked about at the end of the merge sort was comparing a quadratic and N-squared sort, this left and right sort right here to the linear arrhythmic which is the N-log in kind that merge sort runs in, right, showing you that really way out-performing even on small values and getting the large values [inaudible] just getting enormous in terms of what you can accomplish with a N-log-in algorithm really vastly superior to what you're gonna get out of a N-squared algorithm.

I said well, N-log-I told you without proof that you're going to take on faith that I wouldn't lie to you, that that is the best we can do in terms of a comparison based sort, but what we're going to look for is a way of maybe even improving on those kinds of merge sort by kind of a one thing we might want to do is avoid that copying of the data out and back that merge sort does and maybe have some lower cost in factors in the process, right that can bring our times down even better.

So the sort we're looking at is quick sort. And I think if you were gonna come up with an algorithm name, naming it Quick Sort becomes good marketing here so it kind of inspires you to believe actually that it's gonna live up to its name, is a divide-and-conquer algorithm that takes a strategy like merge sort in the abstract which is the divide into two pieces and [inaudible] sort those pieces and then join them back together.

But whereas merge sort does an easy split hard join, this one flips it on its head and said what if we did some work on the front step in the splitting process and then that may give us less to do on the joining step and so the strategy for quick sort is to run to the pile of papers, whatever we're trying to work at in a partitioning step is the first thing that it does and that's the splitting step and that partitioning is designed to kind of move stuff to the lower half and the upper half.

So having some idea of what the middle most value would be, and then actually just walking through the pile of papers and kind of you know, distributing to the left and right portions based on their comparison to this middle-most element.

Once I have those two stacks – I've got A through M over here and N through Z over there, that we're cursively sorting those takes place, kind of assuming my delegates do their work and in the process of re-joining them is really kind of simple. They kind of, you know, joined together with actually no extra work, no looking, no process there.

So the one thing that actually is is where all the trickiness comes into is this idea of how we do this partitioning. So I was being a little big disingenuous when I said well if I had this stack of exam papers and I know that the names, you know range over the alphabet A through Z then I know where M is and I can say well, that's a good mid pint around to divide.

Given that we want to make this work for kind of any sort of data, we're not going to [inaudible] know, what's the minimal most value. If we have a bunch of numbers, are they test scores, in which case maybe 50 is a good place to move. Are they instead, you know, population counts of states in which case millions would be a better number to pick to kind of divide them up.

So we have to come up with a strategy that's gonna work, you know, no matter what the input that's somehow gonna pick something that's kind of reasonable for how we're gonna do these divisions. There's this idea of picking, called a pivot. So given this, you know, region, this set of things to sort, how do I know what's a reasonable pivot. We're looking for something that's close to the median is actually ideal.

So if we could walk through them and compute the median, that would be the best we could do, we're gonna try to get around to not doing that much work about it, though. We're gonna actually try to kind of make an approximation and we're gonna make a really very simplistic choice about what my approximation would be. We're gonna come back and re-visit this a little bit later.

But I want to say, "Well, I just took the first paper off the stack. If they're in random order, right. I look at the first one; I say Okay, it's, you know, somebody king. Well, everybody less than king goes over here. Everybody greater than king goes over there.

Now king may not be perfectly in the middle and we're gonna come back to see what that will do to the analysis, but at least we know it's in the range of the values we're looking at, right, and so it at least did that for us. And it means no matter what our data is, we can always use that. It's a strategy that will always work. Let's just take the thing that we first see and use it to be the pivot and them slot them left and right.

So let me go through looking at what the code actually does. This is another of those cases where the code for partition is a little bit frightening and the analysis we're looking at is not actually to get really really worried about the exact details of the less than or the less than or equal to. I'm gonna talk you through what it does, but the more important take-away point is what's the algorithm behind Quick Sort. What's the strategy it's using to divide and conquer and how that's working out.

So the main Quick Sort algorithm looks like this. We're given this vector of things we're trying to sort and we have a start index and a stop index that identifies the sub region of the vector that we're currently trying to sort. I'm gonna use this in subsequent calls to kind of identify what piece we're recursively focusing on. As long as there's a positive difference between the start and the stop so there's at least two elements to sort, then we go through the process of doing the division and the sorting.

So then it makes the call to partition saying sub-region array do the partition and we'll come back and talk about that code in a second and the thing we're trying to find the partition is the index at which the pivot was placed.

So after we did all the work, it moved everything less than the pivot to the left side of that, everything greater than the pivot to the right side and then the pivot will tell us where the division is and then we make two recursive calls to sort everything up to but not including pivot, to sort everything past the pivot to the end, and then those calls are operating on the array in place, so we're not copying it out and copying it back. So in fact, there actually even isn't really any joint step here.

We said sort the four things on the left, sort the seven things on the right and then the joining of them was where they're already where they need to be so in fact we don't have anything to do in the joined wall.

The tricky part is certainly in partition. The version of the partition we're using here is one that kind of takes two indices, two fingers, you might imagine and it walks a one end from the stop position down and one from the start position over and it's looking for elements that are on the wrong side.

So if we start with our right hand finger on the very end of the array, that first loop in there is designed to move the right hand down looking for something that doesn't belong in left side, so we identified 45 as the pivot value. It says find something that's not greater then 45. That's the first thing we found coming in from the right downward that doesn't belong, so only that first step.

So it turns out it takes it just one iteration to find that, and so okay, this guy doesn't belong on the right-hand side. Now it does the same thing on the inverse on the left-hand side. Try to find something that doesn't belong on the left-hand side.

So the left hand moves up. In this case, it took two iterations to get to that, to this 92. So now we have a perfect opportunity to fix both our problems with one swap, exchange what's at the current position of left hand with right hand and now we will have made both of our problems go away and we'll have kind of more things on the left and the right that belong there and then we can walk inward from there, you know, to find subsequent ones that are out of order.

So those two get exchanged and now right again moves into find that 8, left moves over to find that 74. We swap again. And as we just keep doing this, right, until they cross, and once they've cross, right then we know that everything that the right scanned over is greater than the pivot, everything in the left was less than and at that point, right we have divided it, right, everything that's small is kind of in the left side of the vector. Everything that's large in the right side.

So I swap these two and now I get to that place and I say okay, so now big things here, small things there. The last thing we do is we move the pivot into the place right between them and know that it is exactly located right in the slot where they cross.

Now I have two sub arrays to work on. So the return from partition in this case will be the index four here and it says Okay, go ahead and quick sort this front-most part and then

come back and do the second part. So in recursion, kind of think of that as being postponed and it's waiting; we'll get back to it in a minute, but while we work on this step, we'll do this same thing.

It's a little big harder to see it in the small kind of what's going on, but in this case, right, it divided that because the pivot was 8 into 8 and the three things greater than the pivot. In this case, the 41 is the pivot so it's actually going to move 41 over there. It's going to have the left of that as it keeps working its way down, it gets smaller and smaller and it's harder to see the workings of the algorithm in such a small case of it rearranging it, but we'll get back to the big left half in a second.

And so now that that's all been sorted, we're revisiting the side that's advancing above the pivot to get these five guys, six guys in the order, taking the same strategy. You pivot around 67 [inaudible] doing some swapping [inaudible] and we'll see it in slightly bigger case to kind of get the idea, but very quickly there's something very interesting about the way quick sort is working is that that partition step very quickly gets things kind of close to the right place.

And so now that we've done both the left and the right we're done. The whole thing is done. Everything kind of got moved into the right position as part of the recursive, part of the sorting and doesn't need to be further moved to solve the whole problem.

That first step of the partition is doing a lot of the kind of throwing things kind of left and right, but it's actually quickly moving big things and small things that are out of place closer to where they're gonna eventually need to go, and it turns out to be a real advantage in terms of the running time of this sort because it actually is doing some quick movement close to where it needs to be and that kind of fixes it up in the recursive calls that examine the smaller sub sections of the [inaudible].

Let me show you what it sounds like, because that's really what you want to know. So let's – it's gonna kind of [inaudible]. So you can see the big partitioning happening and then it kind of jumbling things up and then coming back to revisit them, but you notice that quite quickly kind of all the small ones kind of got thrown to the left all.

All the large elements to the right and then the kind of process of coming back and so the big partition that you can see kind of working across them and then kind of noodling down. If I turn the sound on for it and I'll probably take it down a little bit.

So the big partition steps making a lot of noise, right and moving things kind of quickly and it almost appears, you know, to hear this kind of random sort of it's hard to identify what's going on during the big partition, but then as you hear it make its way down the recursive tree it's focusing on these smaller and smaller regions where the numbers have all been gathered because they are similar in value.

And you hear a lot of noodling in the same pitch range region as its working on the smaller and smaller sub arrays and then they definitely go from low to high because of

the recursion chooses the left side to operate on before the right side that you hear the work being done in the lower tones before it gets to the finishing up of the high tones. Kinda cool.

Let us come back here and talk about how to analyze this guy a little bit. So the main part of the algorithm is very simple and deceptively simple because all the hard work was actually done in partition.

The partition step is linear and if you can kind of just go along with me conceptually, you'll see that we're moving this left finger in; we're moving this right finger in and we stop when they cross, so that means every element was looked at. Either they both matched over here and they matched over here, but eventually they let you look at every element as you worked your way in and did some swaps along the way.

And so that process is linear and the number of elements. There's a thousand of them kind of has to advance maybe 400 here and 600 there, but we'll look at all the elements and the process of partitioning them to the lower or upper halves.

Then it makes two calls, quick sort to the left and the right portions of that. Well, we don't know exactly what that division was, was it even, was it a little lop sided and that's certainly going to come back to be something we're gonna look at.

If we assume kind of this is the ideal 50/50 split sort of in an ideal world if that choice we had for the pivot happened to be the median or close enough to the median that effectively we got an even division, at every level of recursion, then the recurrence relation for the whole process is the time required to sort, an input of N is in to partition it and then 2, sorting two halves that are each half again as big as the original input.

We saw that recurrence relationship for merge sort; we solved it down. Think in terms of the tree, right you have the branching of three, branching of two, branching of two and at each level the partitioning being done across each level and then it going to a depth of login, right, the number of times you can [inaudible] by two [inaudible] single case arrays that are easily handled.

So it is an N login sort in this perfect best case. So that should mean that in terms of Big O growth curves, right, merge sort and quick sort should look about the same. It does have sort of better factors, though. Let me show you. I go back here and I just run them against each other.

If I put quick sort versus merge sort here, stop making noise, the quick sort pretty handily managed to beat merge sort in this case, merge sort was on the top, quick sort was underneath and certainly in terms of what was going on it was doing, looks like more comparisons right, to get everything there, that partition step did a lot of comparison, but fewer moves, and that kind of actually does sort of make intuitive sense.

You think that quick sort very quickly moves stuff to where it goes and does a little bit of noodling. Merge sort does a lot of moving things away and moving them back and kind of like as you see it growing, you'll see a lot of large elements that get placed up in the front, but then have to be moved again right, because that was not their final resting place.

And so merge sort does a lot of kind of movement away and back that tends to cost it overall, right. You know, a higher constant factor of the moves than something like quick sort does where it more quickly gets to the right place and then doesn't move it again.

So that was all well and good. That was assuming that everything went perfectly. That was given our simplistic choice of the pivot, you can imagine that's really assuming a lot right that went our way. If we look at a particularly bad split, let's imagine that the number we have to pick is pretty close to, you know, one of the extremes.

If I were sorting papers by alphabet and if the one on top happened to be a C, right, well then I'm gonna be dividing it pretty lop sided, right, it's just gonna be the As and the Bs on one side and then everything [inaudible] to the end on the other side.

If I got a split that was about 90/10, ninety percent went on one side, ten percent on the other, you would expect right for it to kind of change the running time of what's going on here, so I get one tenth of them over here and the low half and maybe be nine-tenths in the right half. Let's just say every time it always takes nine-tenths and moves it to the upper half, so I kept picking something that's artificially close to the front.

These like will kind of peter out fairly quickly diving N by 10 and 10 and 10 and 10, eventually these are gonna peter out very soon, but this one arm over there where I keep taking 90 percent of what remains.

I had 90 percent, but I had 81 percent. I keep multiplying by nine-tenths and I'm still kind of retaining a lot of elements on this one big portion, that the depth of this tree isn't gonna bottom out quite as quickly as it used to, but it used to the number of times you could divide N by 2, the log based 2 of N was where we landed.

In this case, I have to take N and multiply it by nine-tenths so instead of one half, right, to the K, it's nine tenths to the K and it's like how many iterations how deep will this get before it gets to the simple case and so solving for N equals ten-ninths to the K is taking the log-based ten-ninths of both sides, then K, the number of levels is the log-based ten-ninths of them.

We're kind of relying on a fact of mathematics here though is that all algorithms are the same value, so log-based A of N and log-based B of N, they only differ by some constant that you can compute if you just rearrange the terms around. So in effect this means that there is a constant in front of this. It's like a constant difference, the ratio between ten-ninths and 2 that distinguishes these, but it still can be logged based 2 of N in terms of Big O, right, where we can throw away those constants.

So even those this will perform, obviously in, you know, experimentally you would expect that getting this kind of split would cause it to work more slowly than it would have in a perfect split situation. The Big O [inaudible] is still gonna look in-log-in arrhythmic and [inaudible].

So that was pretty good to know. So it makes you feel a little bit confident, like sometimes it might be getting an even split and sometimes it might be getting one-third, two-thirds, sometimes it might be getting you know, nine-tenths, but if it was kind of in the mix still dividing off some fraction is still doing okay.

Now, let's look at the really, really, really worst case split. The really, really worst case split right, it didn't even take off a fraction. It just managed to separate one, but somehow the pivot here did a really terribly crummy job, right, of dividing it at all that in effect, all the elements were greater than the pivot or less than the pivot since they all landed on one side and the pivot was kind of by itself.

So starting with an input of size N, we sectioned off 1 and we had N minus 1 remaining. Well, the next time, we sectioned off another one, so this happened again and again and again. We just got really bad luck all the way through. Each of these levels is only making progress for the base case at a snails pace. Right, one element is being subtracted each subsequent level and that's gonna go to a depth of N.

And so now if we're doing N work across the levels, it isn't quite N work because in fact, some of these bottom out. It's still the Gaussian sum in the end is that we are got N levels doing N work each. We suddenly have what was a linear arrhythmic algorithm now performing quadritically. So in the class the selection sort, inscription sort in its worst-case behavior.

So quite a range whereas merge sort is a very reliable performer, merge doesn't care. No matter what the order is, no matter what's going on, it's the same amount of work, it means that there's some inputs that can cause quick sort to vary between being linear arrhythmic or being quadratic, and there's a huge difference as we saw in our earlier charts about how those things will run for large values.

So what makes the worst case – given how we're choosing a pivot right now is to take the first thing in the sub section to look at as the pivot, what's the example input that gives you this?

Student: Sorted.

Instructor (Julie Zelenski):If it's already sorted. Isn't that a charmer? Here's a sorting algorithm. If you ask it to do something and in fact if you accidentally [inaudible] twice, you already had sorted the data and you said, "Oh, you did something," and you passed it back to the sort, it would suddenly actually degenerate into its very worst case.

It's already sorted, so it would say, "I've got this stack of exam papers." I look at the first one and I say, "Oh, look it's Adam's." And I say, "Okay, well, let me go find all the ones that belong in the left side." None of them do. I said, "Okay, well put Adam's over here." I'll [inaudible] sort that.

That was easy. Oh, let me look at what I've got left, oh with baker on the top. Okay, well let me put Baker by itself, but could we find the other ones? Oh, no, no more. It would just you know, continue because I'm doing this thing, looking at all N, looking at all N minus 1, looking at N minus 2, all the way down to the bottom making no, recognizing nothing about how beautifully the data already was arranged and you know, getting its worst case.

There's actually a couple of others that come in, too. That's probably the most obvious one to think of is it's coming in in increasing order. It's also true if its in decreasing order. If I happen to have the largest value on the top, then I'll end up splitting it all into, everything to the left side and nothing to the right.

There are actually other variations that will also produce this. If at any given stage that we're looking at a sub array, if the first value happens to be the extreme of what remains whether it's the small to the large, and it could alternate, which would be kind of a funny pattern, but if you have the tallest and the shortest and the tallest and then another tallest and then a shortest, so those would look a little bit harder to describe, but there are some other alternatives, that product this bad result.

All of these we could call degenerate cases. There's a small number of these relative to the N factorial [inaudible] your data could come in. In fact, there was a huge number, and so there's lots and lots of ways it could be arranged. There's a very small number of them that would be arranged in such a way to trigger this very, very worst-case behaviors.

Some are close to worst case, like if they were almost sorted, they might every now and then get a good split, but mostly a bad split, but the number that actually degenerate to the absolute worst case is actually quite small, but the truth is is do we expect that those outputs are somehow hard to imagine us getting.

Are they so unusual and weird that you might be willing to say it's okay that my algorithm has this one or a couple of bad cases in it because it never comes up.

In this case, the only thing that your data came in sorted or almost sorted or reverse sorted is probably not unlikely. It might be that you know, you happen to be reading from a file on disc and somebody has taken the [inaudible] and sort it before they gave the data to you. If all the sudden that caused your program to behave really badly, that would really be a pretty questionable outcome.

So let's go aback and look at just to see, though. It's kind of interesting to see it happening in – this is against quick sort versus merge sort. If I change the data to be partially sorted and let it run again, if I still manage to beat it doing a little bit more, if I

change it to totally sorted or reverse sorted, for that matter, and so they look like they're doing a lot of work, a lot of work about nothing.

And you can see that quick sort really is still way back there. It has looked at the first 10 percent or so and is doing a lot of frantic partitioning that never moves anything. And visibly kind of traipsing it's way up there. Merge sort meanwhile is actually on the beach in Jamaica with a daiquiri mocking quick sort for all those times that it had said it was so much better than it was. "Duh, it was already sorted; you Doofus." Apparently it wasn't listening.

Merge sort almost looked like it did nothing and that's because it took the left half, which is the smallest half, moved it off, right, kind of copied it back and it ends up copying each element back to the same location it was already originally present in. There you go, quick sort taking its time and it if I ran it against, let's say, insertion sort and selection sort, why not [inaudible] nothing better to do than to sort for you all day long.

So there insertion sort actually finishing very early because it does nothing. It sort of looked at them all and said they were already in the right order, merge sort doing a little bit more work to move everything back and forth in the same place.

But in this case, selection sort and quick sort here at the bottom and selection sort here at the top doing really roughly about the same work if you look at the sum total of the comparisons and moves and then time spent on those suddenly shows that yeah, in this input situation quick sort is performing like an N-squared sort, and obviously very similar constant factors to those present in selection sort, so really behaving totally quadratically in that worst-case input.

If I make it reverse sorted, it's a little more interesting because you can kind of see something going on. Everybody kind of doing their thing. Insertion sort now kind of hitting its worst case, so it actually coming in dead last because it did so much extra moving, but still showing the kind of quadratic terms, right, that selection sort and quick sort are both having, but higher constant factors there, so taking almost twice as long because actually doing a little bit extra work because of that inverted situation.

[Inaudible] something big, just cause. We'll put it back into random, back into full speed and kind of watch some things happen. So quick sort right, just sort of done in a flash, merge sort finishing behind it and ordinarily the quadratic sorts taking quite a much longer time than our two recursive sorts.

But it's the random input really buying quick sort its speed. Is it gonna win – oh, come on. Oh, come on. Don't you want selection sort to come behind? I always wanted it to come from behind. It's kind of like rooting for the underdog. Yes.

And yet, just remember don't mock insertion sort. What it sorted is the only one that recognizes it and does something clever about it. [Inaudible] sort is anti-clever about it, so it still can hold its head up and be proud. This kind of [inaudible].

I was thinking about like what algorithms, right, there's a reason why there's four that we talked about and there's actually a dozen more. It's like different situations actually produce different outcomes for different sorts and there are reasons that even though they might not be the best sort for all purposes, there are some situations where it might be the right one, so if you had it, as data said that you expect it to be mostly sorted, but had a few [inaudible] in it.

Like if you had a pile or sand papers that were sorted and you dropped them on the floor and they got scuffed up a little bit, but they're still largely sorted, insertion sort is the way to go. It will take advantage of the work that was already done and not redo it or create kind of havoc the way quick sort would.

But quick sort, the last thing we need to tell you, though, about it is we can't tolerate this. Like the way quick sort is in its classic form with this first element as pivot would be an unacceptable way to do this algorithm. So quick sort actually typically is one of the most standard element that's offered in a library of programming languages, the sort element.

Well, it has to have some strategy for how to deal with this in a way that does not degenerate and so the idea is, what you need to do is just pick a different choice for the pivot, a little bit more clever, spend a little bit more time, do something that's a little less predictable than just picking that first most element.

So to take it to the far extreme, one thing you could do is just computer the median, analyze the data and compute the median. It turns out there is a linear time algorithm for this that would look at every element of the data set once and then be able to tell you what the median was and that would guarantee you that 50/50 split.

So if you go find it and use it at every stage, you will guarantee it. Most algorithms don't tend to do that. That's actually kind of overkill for the problem. We want to get it to where it's pretty much guaranteed to never get the worst case.

But we're not concerned with it getting 50/50. It's got 60/40 or something close enough, that actually, you know, 60/40, 70/30 and it was bopping around in that range it'd be fine, so the other two that are much more commonly used is some kind of approximation of the median with a little bit of guessing or something in it.

For example, median of three takes three elements and typically it takes three from some specified position, it takes the middle most element, the last element and the front element and it says, "Okay, given those three, we arrange them to find out what's the middle of those three, and use that."

If the data was already sorted, it turns out you've got the median, right because it wasn't the middle most. If data was just in random position, then you've got one that you know, at least there's one element on one side, right, and so the odds that that every single time would produce a very bad split is pretty low. There are some inputs that could kind of get it to generate a little bit, but it's pretty foolproof in most ordinary situations.

Even more unpredictable would be just choose a random element. So look at the start to stop index you have, pick an random there, flop in to the front and now use it as the pivot element. If you don't know ahead of time how your random number [inaudible] it's impossible to generate an input and force it into the worst-case behavior.

And so the idea is that you're kind of counting on randomness and just the probabilistic outcome if it managing to be such that the way it shows the random element was to pick the extreme and everything, it is just impossible, the odds are astronomically against it, and so it will, a very simple fix, right that still leaves the possibility of the worst case in there, but you know, much, much, much far removed probability sense.

So, just simple things, right, but from there the algorithm operates the same way it always has which is to say, "Pick the median, however, you want to do it, move it into that front slot and then carry on as before in terms of the left and the right hand and moving down and swapping and recursing and all that [inaudible]."

Any questions; anything about sorting, sorting algorithms? Why don't we do some coding actually, which is the next thing I want to show you because once you know how to write sorts, sort of the other piece I think you need to have to go with it is how would I write a sort to be generally useful in a large variety of situations that knowing about these sorts I might decide to build myself a really general purpose, good, high performance, quick sort, but I could use again, and again and again.

I want to make it work generically so I could sort strings, I could sort numbers, or I could sort students, or I could sort vectors of students or whatever, and I'd want to make sure that no matter what I needed to do it was gonna solve all my problems, this one kid of sorting tool, and we need to know a little bit of C++ to make that work by use of the function template.

So if you want to ask about sorting algorithms, now is the time.

Student:[Inaudible].

Instructor (Julie Zelenski):We don't bubble sort, which is kind of interesting. It will come up actually in the assignment I give you next week because it is a little bit of an historical artifact to know about it, but it turns out bubble sort is one of those sorts that is dominated by pretty much every other sort you'll know about in every way, as there's really nothing to recommend it.

As I said, each of these four have something about them that actually has a strength or whatever. Bubble sort really doesn't. It's a little bit harder to write than insertion sort. It's a little bit slower than insertion sort; it's a little bit easier to get it wrong than insertion sort.

It has higher constant factors. You know, it does recognize the data and sort order, but so does insertion sort, so it's hard to come up with a reason to actually spend a lot of time on

it, but I will expose you to it in the assignment because I think it's a little bit of a – it's part of our history right, as a computer science student to be expose to. Anything else?

I'm gonna show you some things about function templates. Oh, these are some numbers; I forgot to give that. Just to say, in the end, right, which we saw a little bit in the analysis that the constant factors on quick sort are noticeable when compared to merge sort by a factor of 4, moving things more quickly and not having to mess with them again.

Student: Quick sort [inaudible] in totally random order?

Instructor (Julie Zelenski):Yes, they are. Yes –

Student: So it doesn't have them taking –

Instructor (Julie Zelenski):So this is actually using a classic quick sort with no degenerate protection on random data. If I had put it in, sorted it in on that case, you would definitely see like numbers comparable to like selection sorts, eight hours in that last slot, right.

Or you [inaudible] degenerate protection, and it would probably have an in the noise a small slow down for that little extra work that's being done, but then you'll be getting in log and performance reliable across all states of inputs.

Student: So [inaudible] protection as it starts to even out time wise with merge sort, does it still –

Instructor (Julie Zelenski):No, it doesn't. It actually is an almost imperceptible change in the time because it depends on which form of it you use because if you use, for example the random or median of three, the amount of work you're doing is about two more things per level in the tree and so it turns out that, yeah, over the space of login it's just not enough to even be noticeable.

Now if you did the full median algorithm, you could actually start to see it because then you'd see both a linear partition step and a linear median step and that would actually raise the cause and factors where it would probably be closer in line to merge sort. Pretty much nobody writes it that way is the truth, but you could kind of in theory is why we [inaudible].

What I'm going to show you, kind of motivate this by starting with something simple, and then we're gonna move towards kind of building the fully generic, you know, one algorithm does it all, sorting function.

So what I'm gonna first look at is flop because it's a little bit easier to see it in this case, is that you know, sometimes you need to take two variables and exchange their values.

I mean you need to go through a temporary to copy one and then copy the other back and what not – swapping characters, swapping ends, swapping strings, swapping students, you know, any kind of variables you wanted to swap, you could write a specific swap function that takes two variables by reference of the integer, strain or character type that you're trying to exchange and then copies one to a temporary and exchanges the two values.

Because of the way C++ functions work, right, you really do need to say, "It's a string, this is a string, you know, what's being declared here is a string," and so you might say, "Well, if I need more than one swap, sometimes that swaps strings and characters, you know, my choices are basically to duplicate and copy and paste and so on. That's not good; we'd rather not do that.

But what I want to do is I want to discuss what it takes to write something in template form. We have been using templates right, the vector is a template, the set is a template, the stack and que and all these things are templates that are written in a very generic way using a place holder, so the code that holds onto your collection of things is written saying, "I don't know what the thing is; is it a string; is it an N; is it a car?"

Well, vectors just hold onto things and you as a client can decide what you're gonna be storing in this particular kind of vector. And so what we're gonna see is if I take those flat functions and I try to distill them down and say, "Well, what is it that any swap routine looks like?" It needs to take two variables by reference, it needs to declare a template of that type and it needs to do the assignment all around to exchange those values.

Can I write a version that is tight unspecified leaving a placeholder in there for what – that's really kind of amazing, that what we'Q1: gonna be swapping and then let there be multiple swaps generated from that one pattern.

So we're gonna do this actually in the compiler because it's always nice to see a little bit a code happening. So I go over here and I got some code up there that I'm just currently gonna [inaudible] because I don't want to deal with it right now. We're gonna see it in a second. It doesn't [inaudible].

Okay, so your usual swap looks like this. And that would only swap exactly integers. If I wanted characters I'd change it all around. So what I'm gonna change it to is a template. I'm gonna add a template header at the top and so the template header starts with a key-word template and then angle brackets that says, "What kind of placeholders does this template depend on?"

It depends on one type name and then I've chose then name capital T, type for it. That's a choice that I'm going to make and it says in the body of the function that's coming up where I would have ordinarily fully committed on the type, I'm gonna use this placeholder capital T, type instead.

And now I have written swap in such a way that it doesn't say for sure what's being exchanged, two strings, two Ns, two doubles. They're all have to be matching in this form, and I said, "Well, there's a placeholder, and the placeholder was gonna be filled in by the client who used this flop, and on demand, we can instantiate as many versions of the swap function as we need."

If you need to swap integers, you can instantiate a swap that then fills in the placeholder type with Ns all the way through and then I can use that same template or pattern to generate one that will swap characters or swap strings or swap whatever.

So if I go down to my main, maybe I'll just pull my main up [inaudible]. And I will show you how I can do this. I can say int 1 equals 54, 2 equals 32 and I say swap 1-2. So if the usage of a function is a little bit different than it was for class templates, in class templates we always did this thing where I said the name of the template and then it angled back, as I said.

And in particular I want the swap function that operates where the template parameter has been filled in with int that in the case of functions, it turns out it's a little bit easier for the compiler to know what you intended that on the basis of what my arguments are to this, is I called swap passing two integers, it knows that there's only one possibility for what version of swap you were interested in, that it must be the one where type has been filled in with int and that's the one to generate for you.

So you can use that long form of saying swap, angle bracket int, but typically you will not; you will let the compiler infer it for you on the usage, so based on the arguments you passed, it will figure out how to kind of match them to what swap said it was taking and if it can't match them, for example, if you pass one double and one int, you'll get compiler errors.

But if I've done it correctly, then it will generate for me a swap where type's been filled in with int and if I call it again, passing different kinds of things, so having string S equals hello and T equals world, that I can write swap S and T and now the compiler generated for me two different versions of swap, right, one for integers, one for strings on the [inaudible] and I didn't do anything with them. I put them [inaudible], so nothing to see there, but showing it does compile and build up.

That's a pretty neat idea. Kind of important because it turns out there are a lot of pieces of code that you're gonna find yourself wanting to write that don't depend, in a case like swap, swap doesn't care what it's exchanging, that they're Ns, that they're strings, that they're doubles. The way you swap two things is the same no matter what they are.

You copy one aside; you overwrite one; you overwrite the other and the same thing applies to much more algorithmically interesting problems like searching, like sorting, like removing duplicates, like printing, where you want to take a vector and print all of its contents right that the steps you need to do to print a vector of events looks just like the steps you need to do to print a vector of strings.

And so it would be very annoying every time I need to do that to duplicate that code that there's a real appeal toward writing it once, as a template and using it in multiple situations.

So this shows that if I swap, right, it infers what I meant and then this thing basically just shows what happened is when I said int equals four, [inaudible] use that pattern to generate the swap int where the type parameter, that placeholder has been filled in and established that it's int for this particular version which is distinct from the swap for characters and swap for strings and what not.

So let me show you that version I talked about print, and this one I'm gonna go back to the compiler for just a second. Let's say I want to print a vector, printing a vector it like iterating all the members and using the screen insertion to print them out and so here it is taking some vector where the elements in it are of this type name. In this case, I've just used T as my short name here.

The iterator looks the same; the bracketing looks the same. I see the end now and I'd like to be able to print vectors of strings, vectors of ints, vector of doubles with this one piece of code. So if I call print vector and I pas code vector events, all's good; vector doubles, all's good; vector strings, all's good.

Here's where I could get a little bit into trouble here. I've made this [inaudible] chord that has an X and a Y field. I make a vector that contains these chords. And then I try to call print vector of C. So when I do this, right, it will instantiate a versive print vector where the T has been matched up to, oh it's chords that are in there; you've got a vector of chords.

And so okay, we'll go through and iterate over the sides of the vector and this line is trying to say see out of a chord type. And struts don't automatically know how to out put themselves onto a string, and so at this point when I try to instantiate, so the code in print vector is actually fine and works for a large number of types, all the primitive types will be fine, string and things like that are fine.

But if I try to use it for a type for which it doesn't have this output behavior, right I will get a compiler error, and it may come as a surprise because a print vector appeared to be working fine in all the other situations and all the sudden it seems like a new error has cropped up but that error came from the instantiation of a particular version in this case, that suddenly ran afoul of what the template expected of the type.

The message you get, let's say in X code looks like this, no match for operator, less than, less than in standard CL, vector element, operator [inaudible]. So I'm trying to give you a little clue, but in its very cryptic C++ way of what you've done wrong.

So it comes back to what the template has to be a little bit clear about from a client point of view is to say well what is it that needs to be true. Will it really work for all types or is

there something special about the kind of things that actually need to be true about what's filled in there with a placeholder to make the rest of the code work correctly.

So if it uses string insertion or it compares to elements using equals equals, right, something like a strut doesn't by default have those behaviors and so you could have a template that would work for primitive types or types that have these operations declared, but wouldn't work when you gave it a more complex type that didn't have that data support in there.

So I have that peace code over here. I can just show it to you. I just took it down here. This is print vector and it's a little bit of template. And if I were to have a vector declared of ints, I say print vector, V that this compiles, there's nothing to print. It turns out it'll just print empty line because there's nothing in there, but if I change this to be chord T and my build is failing, right, and the error message I'm getting here is no match for trying to output that thing and it tells me that all the things I can output on a string.

Well, it could be that you could output a [inaudible] but it's not one of those things, right, chord T doesn't have a built-in behavior for outputting to a stream. So this is going to come back to be kind of important because I'm gonna keep going here is that when we get to the sort, right, when we try to build this thing, we're gonna definitely be depending on something being true about the elements, right, that's gonna constrain what the type could be.

So this is selection sort in its ordinary form, nothing special about it other than the fact that I changed it as so sorting an int, it's the sort of vector with placeholder type. So it's [inaudible] with the template typed in header, vector of type and then in here, operations like this really are accessing a type and another type variable out of the vector that was passed and then comparing them, which is smaller to decide which index.

It's using a swap and so the generic swap would also be necessary for this so that we need to have a swap for any kind of things we want to exchange. We have a template up there for swap, then the sort can build on top of that and use that as part of its workings which is then able to come under swap to any two things can be swapped using the same strategy. And this is actually where templates really start to shine.

Like swap in itself isn't that stunning of an example, because you think whatever, who cares if three lines a code, but every little bit does count, but once we get to things like sorting where we could build a really killer totally tuned, really high performance, you know, protection against a [inaudible] quick sort and we want to be sure that we can use it in all the places we need to sort.

I don't to make it sort just strings and later when I need to sort for integers, I need to copy and paste it and change string to everywhere, and then if I find a bug in one, I forget to change it in the other I have the bug still lurking there.

And so template functions are generally used for all kinds of algorithmically interesting things, sorting and searching and removing duplicates and permuting and finding the mode and shuffling and all these things that like okay, no matter what the type of thing you're working on, the algorithm for how you do it looks the same whether it's ints or strings or whatever.

So we got this guy and as is, right, we could push vectors of ints in there, vectors of strings in there and the right thing would work for those without any changes to it, but it does have a lurking constraint on what much be true about the kind of elements that we're trying to sort here.

Not every type will work. If I go back to my chord example, this XY, if have a vector of chord and I pass it to sort and if I go back and look at the code, you think about instantiating this with vector, is all chord in here, all the way through here, this part's fine; this part's fine; this part's fine, but suddenly you get to this line and that's gonna be taking two coordinate structures and saying is this coordinate structure less than another.

And that code's not gonna compile. Yes.

Student: Back when we were doing sets, we were able to explicitly say how this works.

Instructor (Julie Zelenski):Yeah, so you're right where I'm gonna be, but I'm probably not gonna finish today, but I'll have to finish up on Wednesday is we're gonna do exactly what Seth did, and what did Seth do?

Student: Comparison function.

Instructor (Julie Zelenski):That's right, he used a comparison function, so you have to have some coordination here when you say, well instead of trying to use the operate or less than on these things, how about the client tell me as part of calling sort, why don't you give me the information in the form of a function that says call me back and tell me are these things less than and so on and how to order them.

So that's exactly what we need to do, but I can't really show you the code right now, but I'll just kind of like flash it up there and we will talk about it on Wednesday, what changes make that happen.

We'll pick up with Chapter 8, actually after the midterm in terms of reading.

[End of Audio]

Duration: 50 minutes

ProgrammingAbstractions-Lecture17

Instructor (Julie Zelenski):Hey there. All right. So thank you very much for your duty to the CS106B midterm exam last night. An unusual event by all accounts, but we did all manage to get it done and have that past us. We'll be grading that this week, but given the staff member's scheduling, we're actually going to do most of the grading this weekend. So it should be a little while before you hear back from us on the grades for that, but never fear. We've got them and we're looking at them.

I look through them last night and am already getting good sense that there's a lot of things you guys know how to do well, which is very pleasing. So we'll have more reports on that later.

Assignment five, which we gave out as your parting gift for coming to the midterm – if you didn't pick it up, there's copies of the handout in the back, and the sources are up on the web.

Just to give you a little sense of the assignment, it's actually a smaller assignment, right? It's one that I think most of you will be able to complete in less than ten hours. And by design, kinda giving you something to kind of practice a little bit with sorting but give you a little bit of a break when everything else is getting crazy in the rest of your classes. There are two parts to it. One is kind of identifying some mystery source by doing some experimentation and kind of a scientific method on that. And then the second one is running your own template sort based on a new sorting algorithm that you go and explore and learn something about that. And that has a pretty different setup for what we're expecting to do, which is actually we are encouraging you to go look at other books and web resources and talk to people and think about things and come up with an algorithm that you'd like to learn a little bit more about and implement. You can make this task extremely easy. I don't recommend it, but it turns out you can go and find somebody who's actually almost done the job for you and just copy and paste their code. That is allowable by the bounds of the assignment to be clear, but we are hoping that you actually put a little more thought and effort and learn a little bit more from the process of exploring and thinking about a goal – like, having a goal. I'd like the kinda sort that does this well or would be useful in this situation and learn more about it. But that said, you are able to use a lot of resources. You need to be clear about citing your sources, right? That's the most important thing in terms of academic integrity – always – in this class and everything you do, both in the academic and the professional world – is that when you are making use of someone else's work as part of something you're producing – is to be clear that you give credit where credit is due. So you can make it into a bigger project if you've got the time and energy. You can also take it a little bit easy gearing up. There will be two more assignments to go out after this one that both are pretty substantial. So – in terms of planning this – maybe the time to take it easy and then get ready for the things coming. Today, I'm gonna finish the little bit about template functions that I had left hanging on Friday's lecture, and then we're gonna start talking about OOP and class design implementation. Pretty much from here to the end, we're gonna do just a lot of class implementation. We'll be talking about, "Well, how is it that vector works? How is

it that stack works? And queue works? And map works? And set works? How is it that we build classes up from the inside that have useful properties from the outside?" And there'll be a lot of neat data structures to look at and big old trade-offs to talk about pointers to be used, link lists and more dynamic structures and a great opportunity to kind of put into practice a lot of the hardcore stuff. The reading that we're at is chapter eight. This is when we talk about objects in classes today – kind of the general background material for that, and then we'll talk about class templates starting on Wednesday. Wednesday – Friday – whenever our next lecture is. And then we'll do a lot of class templates from here for several weeks before we're done. All right. Anything on the administration doc? Question?

Student: Just a quick thing. Could you give us, like, a general timeline as to when the next [inaudible]. [Crosstalk]

Instructor (Julie Zelenski):So typically – the point is that one's gonna go out Monday and be due on Wednesday, and then one's gonna go out Wednesday and be due Friday. And there's a week and a few days after it. So the size of boggles space for doing the work. It's not Friday. It's the Friday of dead week.

Anything else you wanna know?

Okay. So what did you think of the midterm? Thumbs up? Thumbs down? Loved it, hated it – two thumbs up. Oh, a thumb sideways. All right. Thumbs up, you get to talk first. What was great about it?

Student:I just loved that big O.

Instructor (Julie Zelenski):You loved that big O. You couldn't have enough of that big O. Okay. All right. Two thumbs up for the big O.

Anybody else who said thumbs up who wants to engage us with their positive memories of the exciting event that was the midterm? There's only one of those. Who wants to say?

Student: No link lists.

Instructor (Julie Zelenski):No link lists. Okay. Okay. You can call that a positive. That seemed to please a certain number of people. And those of you who were kind of a little more iffy? Where was the iffyness came in? Where did the iffyness come in?

Student:[Inaudible].

Instructor (Julie Zelenski):[Inaudible] the time. Within looking at them, everybody did have – not everybody. That's certainly no true. But almost all the ones that I was getting through had kind of stuff to say about every problem. But that is an indication you had all the time to say what you wanted to say. But there was at least some sense that I felt like,

from what I could tell, people had been able to allocate their time enough to get to a little bit of everything.

Somebody with their thumb way down? Wanna tell me what was trouble about that? Anything that surprised you or –?

Student:[Inaudible].

Instructor (Julie Zelenski):I'm a very kinda formulaic exam writer on purpose. So in fact – hopefully you felt like the comparison to the practice was pretty expected – that you're like, "Oh. These are either questions that looked a lot like this on the practice."

We moved stuff around and twisted up a little bit, but there shouldn't be a lot of surprises in terms of what I told you to expect and what there is to expect. In fact, the match to the assignments, also, I think is an important thing I try to maintain.

So that said, hopefully it wasn't too unexpected. Okay.

Well, good. I'll finish – this is the last slide that we had shown on Friday. And I had just blasted it up there and then didn't really talk about it. So I'm gonna talk about it now and then just finish the final adjustment we're gonna make to this, right? So this is – the selection sort code that we have worked to templatize and do a template function, and now it will sort vectors of unknown type. And the change we just made – the one that's highlighted here in blue – was rather than directly taking V sub J, V sub X and comparing them using a less than – is that we're using a call back function. Call back functions – the parameter CMP, or compare there, takes two parameters that are of the client's type, returns an N – which is their ordering – zero, negative or positive, and then, instead of making a direct expression involving less than, we're evoking the client's call back on the two things that we need to compare and saying if that result is less than zero – so V sub J precedes the V of the main index. So it's actually smaller than the main element we've seen so far. Then we update our main index to record J as the place we've seen it. And then it goes to the rest of the code normally swapping that small thing out to the front and then going back around for multiple iterations to keep doing that. With this change, right? We have now kind of generalized this fully in a way that you can sort vectors or strings or vectors of coordinate structures or students or vectors of sets of things as long as the client supplies the appropriate comparison function that explains how do you want those things ordered? Which ones go in the front? Which ones go in the back? It's up to the client to say by giving us that call back. And so a client could use this, for example, to sort coordinates – something that has kind of an X and a Y field by deciding that, "Well, if – first, from the base of the X coordinate, the first one's X coordinate precedes the other, than it's considered less than." You have to make up an ordering, right? That represents what you want, and in this case, I'm gonna say that, "Well, first sort on the X dimension, moving smaller values of X to the front. And where they are tied in the X dimension, then use the Y to break ties here – looking at the Y fields – if it has gotten through those first cases of X compared between C1 and C2. And if all of those things have kind of not been through, then we know that, at this point, X

and Y are exactly equal. And so we have two elements that are the same." And so, given a vector of chord, right? I would invoke sort passing that vector and then the matching comparison function that lets the sorting routine know how to take two chords and decide which goes in front. Yes, sir.

Student: Where we use the type-name template –

Instructor (Julie Zelenski):Um hm.

Student: Is type name a special word?

Instructor (Julie Zelenski):It is. So the type name is a C++ key word that says, "This template depends on a type name."

There are actually other things you can templatize on. We won't see that this quarter, but I'll just kind of leave it as a – things – that there are other things that you may actually have as kind of parameters to the pattern that you're using. The one we're gonna see is when you are actually using a type as a placeholder. And so type name means there's a type in the body that follows, right? There will be usage of the key word of a – our chosen word, type, as a placeholder for something that is supposed to be a type name.

The other word there – they type, right? That was our choice. That could be T or E or F or my type or whatever it is I wanted. That was just a name we got to pick. That my core compare – and the last thing I'm gonna do with this is just make it a little bit more convenient that – given the form I have right now, it says that really, "Oh, there's gonna be two parameters. You're gonna give it a vector, and you're gonna give it a comparison call back function." Before I went through all this trouble to make it have a comparison function, it used to actually just work by default for things like ents and doubles and strings that actually already operated correctly with less than. And so what I'm gonna do is I'm gonna – having made it generalizable, I'm also gonna go back and add in a default behavior which says, "If you don't care, otherwise, and the default less than would work for you, then let's go ahead and use that unless otherwise indicated." So that, kinda, we got back the original behavior we wanted, which is you get to say, "Sort a vector of numbers and have it do the right thing without having to go to the trouble of building a compare ent function to pass to it." So what we will do is we have an operator compare – I'll show you what it looks like here – that there is another header file in the 106B collection called CMPfunction.H. It looks like this. It is a template function itself. It's called operator compare that takes two type things, and then it uses equals, equals and less than to decide whether to return a zero, a negative one or a one. And so operator compare will only work when instantiated for types for which the built in equals, equals and less than are defined and make sense. Now – and that includes all the built – the regular prototypes and double and whatnot, and also includes the string class and then potentially other classes, right? That you may know of that actually have behaviors that have implemented those operators. It will not work on things like struts or vectors or other things that don't have the behavior for this. And it's not intended for those. It's kinda just a pattern for which you can build this – the comparison function needed for the

form by sort. And so then I change the prototype here of sort. It takes a vector of that type. It takes a comparison function. It takes these two things, and then I said, "A deep-hole argument for that is to use operator compare." And so when somebody invokes sort not passing the second argument, then what that will cause the code to do is say, "Okay. Well, they didn't pass it. We need to use the default expression over here, and then operator compare will be instantiated for whatever the type is." So if it was string, then it will build operator compare that operates on strings. If they were ent or double or anything that has a meaningful way to apply the built-in operators if we use that – if I still called sort in this case, past let's say, vector of chords – if I went back to this picture – and I forgot this argument, right? I will get a compiler. And a complier will come from trying to instantiate an operator compare that works on chord where it will get to that line about equals, equals and less than, and say, "You're comparing two structure types, and that doesn't make sense to me." So in this case, the client has the choice of specifying it when they intend it to control that behavior or they need to control that behavior. They can also leave it off in the cases where that default of operator compare will do what they wanted anyway. So I have an example here at the end that says, "If I wanted to sort an array of numbers, I won't need to pass a second argument in the case where I just want to sort it in ordinary increasing order." If I wanted, for example, to sort it in inverted order – I want the largest value to come to the front – that I don't need to run a new sorting routine that then goes and looks for the max and pulls it to the front. I just need to trick selection sort here into believing that the larger numbers go in front of the smaller ones. So I basically wrote an inverted compare function that if A is less than B, it returns one, which basically says, "Well, B goes before A in the ordering I want, which is the larger one should precede it in the output, and similarly for the other case, and then returning zero when they're the same." So this gives us control when we want it in the case of the primitives as well as control when we need it for those types for which the built-ins don't have defined behavior for it. So this makes this, like, the end-all, be-all sorting template. Everything you could want in one package. You will be running something that looks just like this for the second part of the assignment this week, and then what will differ is – well how does it do its work? What strategy is it using to get things in order? But the overall design of it is to build this general purpose, could be used for everything, has a client call back option, also has a default used that makes for very nice, convenient as a client. Any questions about that? It is your life.

Student:I have a question on the [inaudible].

Instructor (Julie Zelenski):Yeah.

Student: Right below the void line –

Instructor (Julie Zelenski):Uh huh.

Student: So what does that do?

Instructor (Julie Zelenski):So the line that's underneath this one?

Student: Yeah.

Instructor (Julie Zelenski):This one here?

So this is the one that says, "The second parameter – the first parameter's a vector of type by reference. The second – it says the second parameter is a comparison function that takes two type arguments and returns it in, and the default assignment to that parameter, if you have not specified it, will be to use operator compare."

That's got a lot of things mashed in there. The syntax for the passing a function as a parameter is a little bit goopy in C++. It kinda includes what looks like a full prototype because that describes for the compiler, "Well, what kind of functions are they?"

And they have to have this parameters in this order – this return type. So we actually kind of have to give this full information about what the shape of such a function is so that it can match it. It can't just take any function, right? It can't pass it get line or convert to lower case – like, not just any function is good. I need to give a description of what kind of functions are right, and those are the kind of functions that take two type arguments and return an ent. And that's what – we're helping the compiler out there with all that syntax.

Student: How will it know if you have a comparison function based on that call?

Instructor (Julie Zelenski):So this isn't the call, right? This is the function I defined. So when I make a call to sort – so if I look at this one – it's all a matter of, "Do I pass one argument or two?"

Student: All right.

Instructor (Julie Zelenski):It's just a default argument. If I don't specify it, then it will use the default value, just like any of the default argument things we know about, like find on a string, which if you don't say, it will start at position zero doing a search. If you do specify, then it uses your index instead.

All right?

Yeah. Let me move on.

So in your 106A class, right? You did a lot of things in the genre of object-oriented programming. This class is not really an object-oriented programming class. It's a class of – that's, in many ways, very procedural, but we use a lot of objects. And what I want to do, just for a minute here, is just kind of reiterate the advantages of using objects in your code – what it is that they provide in terms of structuring and engineering and working to a solution. And then we're gonna go on and start talking about, "Well, we used a lot of objects. What does the other side look like? What is it like to implement an object starting from scratch? What pieces need to be done, and how does that work?

And so the idea of just object-oriented programming is actually – it's a relatively new concept in computer programming, right? It only dates back to the '80s, and the realization, right? That programs at that time – we were operating on a lot of data and moving around employees, or you're managing this calendar, which has these events that are at certain times. And that you had a lot of operations that were, obviously, intended to work on that data – like, move this event to another time or calculate whether this time perceives another – conflicts with this.

And that operations, though – the functions that manipulate that data and the data itself – were not really very tightly coupled in the kinda past. And the idea of [inaudible] used to really couple those things together – to say that, "If you have a data type like a time or a stack or a vector, that the operations that manipulate a vector are best made as part of the package of vector itself – that rather than there being a print vector function that operates somewhere else, having vector-heavy print method."

Putting something onto a stack or asking it for it's size is really an operation that should be owned by the stack – that a stack variable should be able to respond to requests to do things on your behalf through messaging. And so this idea has become kind of permeated the language design in the last decade or so to where pretty much every modern language has some facility for object orientation. Even older languages that didn't before have been updated and brought forward, right? To visual basic, right? Basic, which has been around for half a century, is now objectified, right? In these latest versions.

So [inaudible] that the leverage to the real world is actually an important part of the advantage that, when you talk about a time or a stack or an event or a message if you're doing a mail program – those things have real-world meaning. And so the idea of what they do and how they act actually has – there's a lot of leverage of what you already know to be true about those things in the real world. The notion of kind of dividing it up and saying, "Here's this abstraction of what a stack is," – so we've used stacks all along. Push and pop, right? Push and pop. What does it really do? How does it really manage stuff? Where's the memory coming from? What internal structures? It's not actually something we have to worry about. We're using it abstractly. We're saying, "It's a stack. It has push and pop behaviors."

All the other details of how it works are not something we have to worry about. So let's just focus on some of the other problems – more interesting problems than the kind of things that happen behind the scenes. They're also very tightly encapsulated. So we're not mucking with the stack. It's actually acting force as a black box with – it's kinda like one of those microwaves where it says on the back, "If you take this panel off, right? You void your warrantee." Like, I don't know how my microwave works. I never take that panel off. I hit the panels on the front, food gets hot – I'm very happy. And then things that happen in the large – as you get to building larger and larger projects, right? The idea that multiple engineers are working together and designing on different timeframes – that having things divided in these very strongly capsulated bundles makes it possible to design and test independently of your partners and then join back together and have a

much better eventual result that, if everybody's trying to write all the same code in the same places on top of each other without real discipline to it. Nice packaging for reuse

Something we've been seeing a lot if, right? Is the things that – Lexicon or stack or hue we've used in a lot of different ways – sort of a multi-purpose object. Let me tell you about what – who does what in here and what new rules we're moving into, right? We have done a lot of client use of objects. So a program like the maze program or the random-writer program is some code file containing actual code – .cpp. It messages to objects. It creates objects. It asks them to do things on its behalf. Every class that it is using, it includes the class.h – so stack.h or que.h or vector.h based on what it needs. And so this is the only role we have played so far. We've been client. We've used them, but we don't know anything about these other two things that need to get done.

In the middle, between the client and the implementation here, is something called the interface. We have looked at these header files like stack.h and que.h, and that tells you about what the class provides – what abstraction it is, what it's member functions are, what they're names are, what they're parameters are, what the usage – the correct usage of it is. And so this serves as information to the client about what you can do and what you can't do, what's legal, what's available – and it doesn't contain any inner information about how does it really work? What does it do behind the scenes? What kind of thing can you expect other than kinda what's the correct behavior expected? What we're moving into, right? Is looking at this role and this role over here on the right, which is what is that implementation side work look like? So if we have described what Lexicon is – it's a word list where you can check for the existence of a word or the match of a prefix – and it's our job now to implement that, it's like, "Well, what are we gonna do?" Now this is where we get down and dirty, right? Now it's not about the pretty abstraction. It's about making it work – making it work. Well, making it work efficiently, using appropriate data structures, making it robust, making it handle kinda all kinds of things you throw at it. So for example, the vector class – if you ask it to get something that's off the index, right? It's gonna tell you about it, right? So taking care to make sure the thing just works in all situations, is very bullet proof, is very sturdy, is very informative, right? And well designed, sort of clean, easy to use, does the right things, has the right functionalities, isn't missing anything and that implements it well.

And so in this file, the stack.cpp or the scanner.cpp, right? We have all the [inaudible] functions, so we make the things really happen. We include the class interface, too, because the interface is actually used by both people. The interface on this side tells the client what's there – what you can do. The interface tells the implementer what needs to work, right? You have offered up a method called contains prefix or contains word that is the implementation's job to make that thing do the right thing and return the right answer. So let's look at a simple class. The class I'm gonna talk you through is one of creating a time – sort of a moment in time – 2:15 – that potentially, right? You might be using, let's say, a calendaring program to decide what events you have scheduled for a particular day. You'd like to say, "What's at this time?" And then the idea of time and the things that manipulate time – it actually makes a very good object. It has a very real world analogue about what a time is. And there's a lot of behavior that seems to go with a time.

So in the .h file, which is the interface for this, will be the listing of the class. So the outer structure for this – our class time and that open [inaudible], closed [inaudible] and then a closing semicolon. This kind of models the same syntax that C++ uses for struts. When you say strut something, and then you have these fields, and you also have this closing semicolon. That semicolon, right? Is not present in Java, and it's a pretty easy thing to leave off and then get a lot of [inaudible] error messages that are a little bit goofy from it. So one thing you must wanna register as something to be attentive to. What the interface declares – so this is separate in Java – not separate in Java. I'm saying in C++ it is – is there actually really is a file that says what the class provides, and there's another class that says how it works. In Java, those are kind of one and the same – that the definition of the class and the kind of interface of it were not separated and maintained separately. There are advantages to this and disadvantages. It means there's two separate files. That means that what you give to somebody who's using your code doesn't contain any of the things you don't want them to see – how it works and how it's internally configured. But it also means that you have to keep the two in sync – just the same way any kind of separate prototyping does – is if you change the name in one place, you have to change it in both places to kind of make sure that they're always in match.

So the two main things that get declared within the class declaration or its interface are the data members – so the fields that are a part of a particular time object. And these are declared like ordinary variables in hour, minute. They can be variables of other types, string, vector and things like that as long as anything I'm using, I would have to [inaudible] include at the top. And in this case, I have introduced them under a private section. So unlike Java where every single field has it's own private or public modifier, in C++, you actually introduce a section with private [inaudible], and then everything from there on down is private until it sees something that changes that back to public. So typically, right? You'll have one big, public section, one smaller, private section – they can be in the other order – either way. I typically use this form where the public things sit at the top and the private beneath.

The other things that get declared as part of your interface is what member functions are available. What are the operations that you can manipulate a time with? And so maybe right here, I have something that allows you to change the hour or get the current hour from a time or move it forward by some amount of hours and minutes. I wanna push back that meaning by an hour and 50 minutes. I can call shift by 1:45 to do that. Something that converts time into a string format may be suitable for printing or using in a display is a two-string member function I could add in the time class as part of its public interface. And so the whole list of what I have here – and so kind of as a rough rule, right? Most of your functions will end up public because they're operations you're offering. Your data is almost always private. You don't wanna make that accessible outside of the class. And then occasionally, right? There are reasons, actually, to have some member functions that are private. They're used internally as helpers, and they're part of the strategy, but they're not something you want a client to be able to directly call. So you actually keep them in the private section to indicate they're not part of the interface a client needs to know about. You see it all the time. Any questions about what it looks like there?

So the time [inaudible] I have declared here has two data members – an integer hour and an integer minute. The mechanics of how this works is that the object is basically about the same size as the comparable strut. So it has fields for hour and minutes. It doesn't actually have a bunch of storage for the member functions. Since those are shared across all times, there's no reason that every time carry around a duplicate of that. So what it really carries is its own field – the data members that were declared in the class. And so when I say time T, what I'm getting space for – it looks a lot like a strut – something that has an hour and a minute field. By default, the initialization of this field is as it would be if they were to be declared on the stack. So if I said N to hour, N to minute, I would just get junk contest. So it doesn't set them to zero or do anything clever on our behalf. With these primitive types, it just lets them stay uninitialized. And we'll see about how to fix that in a minute, but just to know.

And then when we talk about time objects, right? Every single time object has its own hour and minute. So this time is 2:15, this time is 7:00 p.m., and the numbers that they are storing and maintaining, right? Are different for their hour and minute. So it's kinda like across all the different time objects, there is individual hour and minute fields associated with each particular time. It all kinda seems to make sense back to the 106A days where you did a lot of this. But if you have a time object – and you've already seen this, but just to mention that – yeah, you access its member functions and its field using dot notation. If you happen to have a pointer, you can use the arrow, which combines the star and the dot, setting – calling set hour, calling get hour, trying to access a field that both the member function use and the field use is dependent on if the – the feature that I'm trying to access being declared public. It was in a private section, and if you don't specify – if you forget to put any of the [inaudible] specifiers on it – by default, they were all private. And so my attempt to access any of these things that were private will result in a compiler error. So it won't let me get past that kind of mistake.

Now the message, we call that the receiver. And we talked about that before – T being the receiver to the set hour, and the assumption being that what I'm trying to do is tell this time object, right? To set the hour it is to 3:00, overriding whatever value was there before. Now what does it look like on the other side? If you are the implementer of time, what kind of things do you have to do to make time behave the way you said it would? You have a file timed out CPP – that's the class implementation file for it. The first thing it will always do is include the class file that it's working on. That's because if I start defining the features of class, I need to know what they are so the compiler can check and make sure that I've – I'm telling the truth, right? That the functions that I'm trying to implement and their parameters and the instance – the data members I'm trying to use match the description I earlier gave about what time was. So both the client needs to see it to use it. The implementer needs to see it to implement the right things that have the right names or the right prototypes.

This is where all the code for the member function goes – is in this file. And there's a little bit of a syntax that you'll need to know about this, which is when I'm ready to implement the set hour member function, that the name that it goes by – its full name – its kinda real name – is time colon colon set hour – that everything that was defined in the

time class is considered to be within this scope, and C++ is not the word for this. The scope time and – that the way to access something from within a scope is to use the name of the scope colon colon and then the thing you wanted to get out – you were trying to find, trying to use. So when we're defining all these member functions, we're saying, "It's the times set hour member function that I'm writing, not just a function called set hour." If I leave this off, I have a function that says, "Boy, its set hour N to new value."

What the compiler thinks I'm writing there is just an ordinary global function. It thinks there's a global function, just name this set hour. It takes one primary to reach its end, and it returns to void. It doesn't think it has anything to do with the time class, and the next thing it will notice is in here, when I'm trying to access features that are relevant to time, it will start giving me compiler errors. It'll say, "Hour? Where did hour come from? I know about new value. It's a [inaudible]. But hour just came out of nowhere. That was completely undeclared to me. Something must be wrong." So if we – that's a mistake you will certainly make at least once, and we'll want to commit to being attentive about. It's like it is the time set hour. It is the times two string. And that is distinguished from other functions of that same name.

That actually is kind of a neat little feature – the fact that that worked that way [inaudible]. The member function side shows up in a lot of our classes on vector, on stack, on string, on map – that the idea that all of those things can be called size is actually really handy because who wants to remember that one of them's length and one of them's size and one of them's numb entries and one of them's depth and one of them's length or something. That – having them all be size means that you have this one name you use when you want to get the information about how big a collection is. And the fact that the compiler can keep them all straight is based on this scoping mechanism. There's a vector size, which is different than stack size, which is different than map size, and it doesn't confuse them up because it has this scoping to keep them all straight.

So when we're writing the implementation of our member functions, there's a couple things, right? That we need to know about how to make them work the way they're supposed to. So in the time member function like shift by, that if we just refer to the field hour or minute – one of the data members of a time object, it is assumed that the hour or minute we're talking about is of the receiver. So shift by is never called without a receiver. There is not mechanism that lets you just call shift by without something in front of it. So shift by got called – there's some time object somewhere where it says t.shiftby an hour and 15 minutes that moves that – the T time forward by that amount of hour and minute change. So in the context of shift by, you can count on that hour and minute refer to the fields of a receiving time object that currently has some value in the hour and minute that we're changing by some delta to move it forward. So any reference to the hour and minute fields with no other qualifying marks means my hour, my minute – the one that got the message.

In the body of the member function, we also – so we can directly access these fields. We can also make further calls to other member functions. If there is a set hour and a set minute setter that are available on the time object, then we can say, "Set hour, hour plus

D – hour – the minute plus D minute, and then that will go through our own setter to update the hour and minute field to the new values. So this is a case where you're seeing a call to a member function without an explicit receiver, and in this case, the receiver is assumed to be the same receiver who originally got the shift by message. So this really isn't dropping the receiver or losing it, it's actually just assuming that, without any other explicit receiver, that it means me. So the time object that was asked to shift tells itself to change the hour and change the minute.

We're gonna talk a little bit about why that might be an important strategy relative to the alternative here, but just to know that mechanically, they both work. And then the last thing [inaudible] is this secret variable – this idea that it knows who the receiver is and that it's actually using it as part of the access for the hour and the minute and the caller – that there actually is a way to explicitly access that syntax if you wanna know. There is a special variable called this that is only valid within the member function of some class – any class. This is a pointer toward the receiver object. So this, in a time member function, is a time pointer. Now in a stack, it would be a stack pointer, and so on.

And I can use the longhand form of this arrow hour to mean take the time this – do you reference it to get to the time object and access its hour field. [Inaudible] this arrow is doing the dereference and then sending itself the member function. This is equivalent, right? To dropping that entirely. So you will rarely see a C++ program where you use this longhand form. But it is the mechanism behind it – you can see explicitly in situations where you might wanna be very clear about it. Sometimes you'll use that. For example, when you have two time objects, and you're trying to compare them, you wanna be distinguishing the parameter from this. You might explicitly say, "This hour – if this hour equals other hour." You might wanna make that clear which one you're talking about rather than dropping it in that situation. Any questions about a little bit mechanics? So I said that I would talk a little bit about, "Well, why might you want to have your member functions just make calls to other member functions rather than directly modify those fields?"

And one of the advantages, right? Of that object sorting programming is that you can actually really be very tightly managing the object's state and trying to make sure that the object never gets inconsistent. For example, if you're a stack, you wanna be sure that if you think the depth of the stack is ten, that there are ten valid elements. You never wanna get in a situation where somehow you think there are eight, but really there are ten or vice versa. It's your job to make sure that it makes sense. In the case of time, there's actually kind of a lot of invalid or illegal time values that you can imagine sneaking into your system where suddenly, you have the time – 55 negative 14. Right? You wanna be sure that the hour and minute really make sense for what you know to be the domain of valid times. By having the instance variables be private – the data members – nobody can muck with them other than you and the implementation. And so if you actually are very disciplined about making sure they can't ever get wrong – that no matter what operations you offer – shifting it forward, shifting it back, resetting it to new values – you make sure that there's no way a bad value could creep in.

And probably the easiest way to do that is to build a gatekeeper – is to have there be one central point where you change hour, and everything goes through that one central point. And that central point is designed to make sure that nothing can sneak past it. So if somebody tries to change the hour to something that's not between 1 and 12, you can just decide to bound it. This may not be the right thing to do. Maybe I should raise an error, but at the very least, I need to do something if they give me negative 14 or 82. And so right now, I choose to say, "Well, I'll just bring it to the closest value that's in range." An error would be a completely valid alternative.

Similarly, something about the minute – if they give me a minute that's not something between 0 and 59, that doesn't make sense. And so I just [inaudible] it – just to take the lower order part of it. If – these are the only places – let's say, imagine all of the time class where I ever assign to hour and minute. And everywhere else where I wanna change it, I say, "Well, here. Change it by this much and stick it through. Here's the value I'd like to set it to. Let me ask set hour to do it on my behalf – that if I've made some error in the calculation or I've been asked to do something kind of ridiculous that it will cause the right eventual assignment to be made." And so that's why I say, "What's this advantage? Why would we wanna do this?"

It's about control, right? Kind of centralizing the access and the robustness of your interface. Just say, "Well, here at the one place wherever change it. Let's make sure everything goes through that same interface. And that way, there's no way you can sneak a – through some back door – a bad value in." So a couple other things you need to know about class mechanics. There are a few special member functions. One is the constructor. So the constructor is a part – it's just automatically tied in with allocation – that when you allocate an object – either with new or on the stack – the constructor is automatically invoked as part of that process. And so that gives you your opportunity, if you define a constructor, to set up your data members the way you want. If you don't specify a constructor, you get the default constructor. The default constructor takes no arguments and basically does nothing to your arguments. So it leaves them uninitialized.

So for most things, that's not gonna be appropriate. It's probably the first thing you wanna do when you make a class is to write a constructor that sets yourself into a known, good state. It has a special prototype. It has to have exactly the same name as the class. It has no return type – not void, not anything. So it looks a little strange when you first see it. And then it can have parameters if you need them. It also can be overloaded. If you want more than one constructor, you can have multiple of them. So if I were to add one to my time class, I put in a function you notice has no return type, right? So the first thing you're seeing is the name of the constructor, which has to exactly match the case and name of the class itself – time. And in this case, choosing to make a constructor that takes two integer arguments – the starting hour and minute that I want.

In the CPP, right? The implementation of the constructor – time – within the scope of the time class – time colon colon time – again, no return type here – taking the hour and minute and setting it. I can actually make calls to my member functions, and given my earlier little speech about this, it seems like it would be even better, for example, to be

calling set hour of HR and set minute of MIN to just make sure that even if they give me initial values that are total garbage, right? That I don't let the time start off with some negative 45 – 100 – kind of state that doesn't make sense.

Once this constructor is in place, right? Then the calls to create a time will show two arguments are required to construct a time that's – to specify the hour and minutes is no longer optional, whereas prior to having a constructor that was a default constructor – the default constructor took no arguments. Once you declare any constructors, the compiler stops giving you anything for free. And so if you want both the default constructor and another several argument constructor, you'll just make several of them. You can have a time here that just says, "Time, open [inaudible] close [inaudible]," and then have a time colon colon time open [inaudible] close [inaudible] and set the hour and minute to sub default of 12:00 or whatever you want it. You can have as many constructors as you need, but typically, I would say most classes have one or two if not that many starting configurations you need to support typically.

So something that's new to you, if you're coming from the Java world, the idea that there's a corresponding parallel function that is tied into deallocation that when an object is going away – the two times when an object is going away. One is when it's leaving scope for a stack allocated object. So if you have time T declared in some foreloop, and when you exit the foreloop, it automatically deallocates that object or destructs it. [Inaudible]. If you have new – that object new time out in the heap that when you delete that time object as part of the deallocation of memory, it will destruct it. So what it does is it calls the destructor when that happens. The destructor is your hook for, "I need to clean up when this object is going away." It has the same name as the class, but prefix with a tilde. So tilde time is the destructor for the time class. It can never have any parameters, and it never has any return type. So time is the constructor name. Tilde time is the destructor's name.

You don't always need a destructor. It's actually a little bit more unusual with some simple classes that – what you would need it for is if there were something that you did as part of constructing or building the object that you needed to clean up. And the most common need for that is you have dynamically allocated some members. So you have an object that holds onto a link list and has just a pointer to the head note and the bunch of cells that follow it that when you were destructing, you would wanna go trace that link list down and delete all those cells if you were gonna clean up after yourself. If you don't do that, you don't have a destructor, right? Then the pointers just get [inaudible], and all that memory is just hanging out there, clouding your heap.

Sometimes there are things about opening files – like maybe you have a bunch of buffered output that you wanna be sure got committed to disc or a network connection you wanna close or something like that – so things that are open that you need to release. So typically, it's like resources that you need to clean up around that goes in the destructor. For the time class, there'd be no need, right? We have just simple integer variables. There's nothing special about them. So we don't actually need to go build a destructor. But some of the classes we'll see later will have a reason for that.

So this idea of object design is sort of one of the more important points to think about when you're setting up and building an object, right? Is it's your responsibility to make an object that kind of does what it says it's gonna do and tries it's best to be robust in the face of all situations. Even in this case that, maybe, the client uses it incorrectly, that the point of this sort of programming is – it's not to think that we're probing among enemies or we're malicious, but it is people make mistakes, right? People accidentally have the time, and they changed it to something that doesn't make sense, or they do the calculation incorrectly or they access off the end of the vector – that having the response to that by your object be to crash or to just produce incorrect information or to return an erroneous result – is really not doing the best you can. So part of your goal in designing this object is just to build something that really is a lovely work of art that does exactly what you want and catches all the mistakes, and even things that are really just the client's fault, right? Asking you to do irrational things are handled gracefully.

And so part of that is really taking kind of a big picture view of all the operations on the class and making sure that you have published the operations that are appropriate, right? That they respond to all forms of input gracefully, and that they kind of manage the object's state as it moves through the different operations in a way that's consistent. And so part of that, right? Is thinking about, "Well, who accesses the data where?" And making sure that those accesses are really understood and controlled. Never having any public data members – once you have a public data member, all bets are off. If hour and minute are public, anybody, anywhere, anytime can just reach in and change it to whatever they want. All guarantees are lost. So you would never want to do that, right? You are exposing part of your internals, right? For everyone to see. It's like having the door on the back of the microwave – you just open it up and you can start, like, rewiring. It's like – that's not gonna end well. You correctly initialized all your members in your constructor, and then you consider where and when to allow access to directly modify those things if – like through a setter – if it's needed and if it is properly constrained.

The other thing that's an important part of kind of designing the interface is making sure that the object [inaudible] is fully formed. That it does all the things you'd like it to do and kind of simplifies the client's use of it as opposed to kind of going half the job and then leading them to have to do a bunch of other things. It has a complete kind of coherent set of applications. So if you needed to print a time that one way, you could supply those. You can say, "Well, you can get the hour. You can get the minute. And you can convert them to string. And you can put a colon in the middle of them. And then you can print it. Why bother me about it? Like, you wanna print time? That's your business." On the other hand, right? Printing seems like a pretty fundamental operation that would just be nice for time to support. Maybe it supports it by having just a print method. Maybe it supports it by giving you a string form like two string where it does those calculations – it converts it and gives it back to you, and then you can decide to just output it as a string.

But having that be a part of the class really makes the client's job just easier. They're likely to need something like that. Why make everybody who uses your time class have to reproduce this functionality that really feels like something the time class itself should

support? And so when you think about your set of operations, you have some lists you might need in the near future. You're also trying to think a little bit more big picture about what – well, in all the situations you use times, what kind of operations come out as being useful. While I'm implementing time, why don't I just kinda build the full-fledge version that has the functionality for the things I imagine to be likely to be needed.

So comparing, right? Two times – is this is one before this one? Are they equal? Right? Moving them around – just kinda giving you the whole package. So here's a little thought about what are the advantages, right? Of the object to design or object in capsulation means that you have described what time does. You say it allows you to set the hour. You have described that it'll compare them to see if one's less than another. You have a two string or a shift by, but you have made no guarantees to them about what you really do – how you really internally work it. The time interface might describe things, like, in terms of hour and minute. "Oh, it's an hour and a minute in time, and you can shift it by hours and minutes. You can ask me for the hour and the minute."

But internally, there's no rule that says – because our interface is hour and minute – that we have to have fields called hour and fields called minute. That in fact manipulating it, right? As hour and minute is messy. And if you throw in a.m./p.m., it's even worse. That all the things, like, people decide if one time is less than another – it's like, "Well, if this one's a.m. and that one's p.m., then it's less than." Right? And if they're both a.m. or p.m., right? Then we start looking at the hour. And if the hour's the same, we start looking at the minute. Like, it just doesn't – it gets goopy. Right? We can make it work, but why do it if we don't have to? So if we just, instead, pick a different representation – it's our internals. Nobody needs to know what we're doing. It's like, "I don't wanna keep track of hour and minute. Hour and minute is some other artificial sort of human representation."

Computers are really good at much more simple things, like what if I just said, "Look. Every day – every minute within the day is assigned a number: zero is midnight, right? And then 720, right? Is noon." And just whatever. You just – you said, "Minutes since midnight." Two minutes into midnight is 12:02. Right? Twelve minutes is 12:10. Sixty minutes is 1:00 a.m. Right? I just mark everything down there. Then it turns out, right? A lot of my operations become really easy. I wanna see if this time is less than the other? It's like, "Okay. Just compare their two minutes since midnight."

I wanna move a time 60 minutes forward? Well, I just add 60 minutes to whatever number I have. You can do the wraparound really easily. So like, if you were close to the end of the day at 11:30, and you add an hour. Right? It's very easy to just use [inaudible] to wrap you back around to the 12:30. But if you're doing it with hours and minutes and a.m./p.m., all those things are very ugly. So the thing is we can still describe our interfaces hours and minutes because that's how clients want to see it. But internally, we get to make all the decisions we want, and we might as well make the ones that make our lives easier – make it easier to get the code right. It's actually – it takes less time to do the operations right. It'll run more efficiently. It uses less space, in fact. And it just made everything easier from our point of view.

And it turns out we can still have operators – like set hour and get hour and set minute and get minute, right? They just internally kinda convert to the internal representation. But they provide an interface, right? That seems appropriate for the client. It doesn't mean it has to match what we do inside. So that's actually kind of a really neat – a neat thing to keep in mind, right? Is that that's part of what we bought – is that they don't actually have any guarantee about how we implemented it, and we're not gonna tell them. So I will leave you with this thought about abstraction because that is sort of a key advantage of working this way, right? Is there really is a wall between the client and the implementer that is an important part of – it feels like a restriction. It feels like a jail. It feels like a prison if you think about it that way. But in fact, it's liberating. It means we're not in each other's hair. We're not dependent on each other in any meaningful way. We have this little, tiny conduit – that chink in the wall we call the interface – that describes what things we both can depend on.

But then everything else is at the liberty of the person who's on their side of the wall to do what they want without regard to what the other person's expectations are. So we'll think about that in terms of, let's say, Lexicon and scanner and vector and all these kinda neat things on Friday to kinda dig around the backside and see what we've been missing out on. I will see you guys then.

[End of Audio]

Duration: 50 minutes

ProgrammingAbstractions-Lecture18

Instructor (Julie Zelenski):Okay. Well welcome, welcome. We got a couple parents who have braved the other activities of Parents Weekend to come here, so thank you for coming to see what it is that we do here in CS106B.

We are still in the process of grading the midterm. In fact, most of our grading, we're planning on taking care of tomorrow, no, Sunday, in some big, massive grading sessions. So with any luck, if it all goes well, we'll actually have them ready to go back on Monday, but if somewhere it turns into some disaster, it may take us a little longer, but hopefully. That's the plan. So you get to have your weekend without worrying about it. Just put it aside and have some doing some stuff.

What I'm gonna talk about today is I'm gonna briefly go through scanner, really fast. Actually, what I really want to talk about is vectors, so maybe I won't even bother with the scanner. We'll just talk about vector, which is, the reading going on with this, Chapter 10: Implementing the Class Template.

So what we're gonna do here for the next two weeks is we're gonna take and put all those templates we did a lot of stuff with. Right? We saw vector, we saw stack, we saw a queue, we saw map, we saw set. And now it's time to say, "Well, how do they work? How do they manage to do the things they do, efficiently, safely, robustly; what is it like to implement those things?"

And so it's – we've had a good run of being the client, which hopefully has impressed you with like why you want these things around. And now, with some knowledge about linked lists and pointers and Big O, how can we make some make those operations run efficiently and make them do cool things? What does it take to be the backend side?

So that's kind of gonna be our role for pretty much the next two weeks straight. And then at the end, we'll kinda come back and try to join both sides together and get a vision of where we're at. But it is time to be implementer.

We'll be doing the Terman Café today after class, so hopefully those of you whose parents aren't here, or your parents can come with us too, we'll – anybody who has time is definitely welcome to come over and have a nice latte with me and hang out. And tell me about their midterm success or their midterm troubles, or just things that are on your mind. So hopefully some of you will be able to join me.

So I'm gonna finish up a couple slides from last time, and then I'm gonna mostly go on to do some code with you together. But I just wanted to try to give you some perspective on the value of abstraction and the idea of an abstract data type, or an ADT, and why this is such a powerful and important concept in management of complexity.

So we saw this, for example, in those first couple assignments that the client uses classes as an abstraction. You have the need to manage something that has a queue-like behavior:

first in, first out. So what you want is something that you can put things in that will enforce that: put things at the end of the line, always return to the head of the line. And that how it managed what it did and what went on behind the scenes wasn't something we had to worry about or even concern ourselves with.

And in fact, even if we wanted to, we couldn't muck around in there. Like it wasn't our business, it was maintained privately. And that is a real advantage for both sides, right? That the client doesn't have to worry about it and can't even get in the way of us, that we can work independently and get our things done.

And so one sort of piece of terminology we often use here, we talk about this wall of abstraction. That there is kind of a real block that prevents the two of us from interfering with each other's process, as part of, you know, combining to build a program together. And there's a little tiny chink in that wall that we call the interface. And that's the way that you speak to the stack and ask it to do things on your behalf, and it listens to your requests and performs them.

And so if you think about the lexicon class, which we used in the Boggle and in the recursion assignment that managed a word list, the abstraction is, yeah, you say, "Is this word in there? Add this word to it. Load the words from a file." How does it store them? How does it do stuff? Did anybody go open up the lexicon.cpp file just to see? Anybody who was curious? And what did you find out?

Student:I just – I think it ended up in there [inaudible].

Instructor (Julie Zelenski):You ended up in there and when you got in there, what did you decide to do?

Student: Leave.

Instructor (Julie Zelenski):Leave. Yeah. Did anybody else open it up and have the same sort of reaction? Over here, what did you think?

Student:I didn't really understand.

Instructor (Julie Zelenski):Yeah. And what did you see?

Student: It was a mess.

Instructor (Julie Zelenski):It was a mess. Who wrote that code? My, gosh, she should be fired.

It's scary. It does something kind of scary. We'll talk about it. Actually, at the end, we'll come back here because I think it's actually a really neat class to study. But in fact, like you open it up and you're like, "I don't want to be here. I want to use a word list. Let me

close this file and let me go back to word list. Add word, contains word, okay." And you're happy about that. Right?

It does something very complex that turns out to be very efficient and optimize for the task at hand. But back to Boggle, you don't want to be worrying about that; you got other things on your plate and that's a fine place to just poke your nose right back out of. So if you haven't had a chance to look at it, because we will talk about it later, but the person who wrote it, was a crummy commoner. I'm just saying that they would definitely not be getting a check-plus on style, so.

So I drew this picture. It's like this wall. Right? So when you made a lexicon, you say, "Oh, I want a lexicon. Add this word. Add that word. Does it contain this word?" And that there is this big wall that – and you think of what's on the other side as a black box. The black box is the microwave that you push buttons and food gets hot. How does it really work? Ah, you know, who knows. You don't open up the back. You don't dig around in it. And this little chink here, that's the interface. And the lexicon provides a very small interface, if you remember, adding the words, checking contains word and prefix, reading the words from a file, really not much else beyond that.

And then now what we're starting to think about is, "Well, what's this other side look like?" What goes over here is that there is this implementer who does know the internal structure. Who does know what shenanigans are being pulled on that inside, who does have access to that private data, and who, upon request when you ask it add a word or look up a word or look up a prefix, internally trolls through that private information and returns information back to the client, having mucked around in those internals.

So when you say add a word, maybe what its got is some array with a count of how many words are used, and it sticks it in the next slot of the array and updates its counter. And we don't know what it does but it's not really our business as the client, but the implementer has kind of the full picture of that.

And then over here on this side, if the client attempts to do the kind of things that are really implementer specific, it tries to access directly the num words and the words array to go in and say, "Yeah, I'd like to put pig in the array. How about I do this? How about I change the number of words? Or how about I stick it in the slot at the beginning and overwrite whatever's there." That an attempt to do this should fail. It should not compile because these fields, that are part of the lexicon, should have been declared private to begin with, to make sure that that wall's maintained. Everything that's private is like over here on this side of the wall, inaccessible outside of that.

And so why do we think this is important? Of all the ideas that come away from 106B, I think this is one of the top three is this idea of abstraction. We actually even call the whole class Programming Abstractions. Because that the advantage of using this as a strategy for solving larger and more complicated problems is that you can divide up your tasks. That you can say this is an abstraction, like a word list, and kind of have it be as fancy as it needs to be behind the scenes but very easy to use from the outside.

So that makes it easy for me to write some piece and give it to you in a form that's easy for you to incorporate. We can work on our things without stepping on each other. As you get to larger and larger projects beyond this class, you'll need ways of making it so three people can work together without stepping on each other's toes the whole time. And classes provide a really good way of managing those boundaries to keep each other out of each other's hair.

And there's a lot of flexibility given to us by this. And we're gonna see this actually as we go forward, that we can talk about what a vector is. It keeps things in index order. Or a stack is, it does LIFO. But there is no guarantee there about how it internally is implemented, no guarantee expressed or implied, and that actually gives you a lot of flexibility as the implementer.

You can decide to do it one way today, and if upon later learning about some other new technique or some way to save memory or time, you can swap it out, replace an implementation with something better, and all the code that depends on it shouldn't require any changes. That suddenly add word just runs twice as fast or ten times as fast, would be something everybody could appreciate without having to do anything in their own code to take advantage of that. So these are good things to know.

So what I'm gonna do actually today is I'm gonna just stop doing slides, because I'm sick of doing slides. We do way too many slides; I'm bored with slides. And what I'm gonna do is I'm gonna actually go through the process of developing vector from just the ground up. So my plan here is to say our goal is to make the vector abstraction real, so to get dirty, get behind the scenes and say we know what vector is. It acts like an array. It has these indexed slots. It's this linear collection. It lets you put things anywhere you like in the vector.

We're gonna go through the process of making that whole thing. And I'm gonna start at the – with a simple form that actually is not templatized and then I'm gonna change it to one that is templatized. So we're gonna see kind of the whole process from start to end.

So all I have are empty files right now. I have myvector.h and myvector.cpp that really have sort of nothing other than sort of boilerplate stuff in them. So let me look at myvector.h to start with. So I'm calling this class myvector so that it doesn't confuse with the existing vector class, so that we have that name there. And I'm gonna start by just putting in the simple parts of the interface, and then we'll see how many other things we have time to kind of add into it. But I'm gonna get the kinda basic skeletal functions down and show how they get implemented, and then we'll see what else we'll try.

So having the size is probably pretty important, being able to say, "Well how many things will I put in there," get that back out, being able to add an element. And I'm gonna assume that right now the vector's gonna hold just scranks. Certainly that's not what we're gonna want in the long run, but I'm just gonna pick something for now so that we have something to practice with. And then I'm gonna have the get at, which give it an index and return something.

Okay, so I think these are the only ones I'm gonna have in the first version. And then the other ones, you remember, there's like a remove at, there's an insert at, there's an overloaded bracket operator, and things like that I'm not gonna show right away.

Question?

Student:[Inaudible].

Instructor (Julie Zelenski):Oh yeah, yeah. This is actually you know kind of just standard boilerplate for C or C++ header files. And you'll see this again and again and again. I should really have pointed this out at some point along the way is that the compiler does not like to see two definitions of the same thing, ever. Even if those definitions exactly match, it just gets its total knickers in a twist about having seen a class myvector, another class myvector.

And so if you include the header file, myvector, one place and you include it some other place, it's gonna end up thinking there's two myvectors classes out there and it's gonna have a problem. So this little bit of boilerplate is to tell the compiler, "If you haven't already seen this header, now's the time to go in there." So this ifindex, if not defined, and then the name there is something I made up, some symbol name that's unique to this file. When I say, "Define that symbol," so it's like saying, "I've seen it," and then down here at the very bottom, there's an end if that matches it.

And so, in the case where we have – this is the first time we've seen the file it'll say, "If not to define the symbol it's not." It'll say, "Define that symbol, see all this stuff, and then the end if." The second time it gets here it'll say, "If not define that symbol, say that symbol's already defined," so it'll skip down to the end if. And so, every subsequent attempt to look at myvector will be passed over. If you don't have it, you'll get a lot of errors about, "I've seen this thing before. And it looks like the one I saw before but I don't like it. You know smells suspicious to me." So that is sort of standard boilerplate for any header file is to have this multiple include protection on it.

Anything else you want to ask about the – way in the back?

Student: Why would it look at it multiple times, though?

Instructor (Julie Zelenski):Well because sometimes you include it and sometimes – for example, think about genlib. Like I might include genlib but then I include something else that includes genlib. So there's these ways that you could accidentally get in there more than once, just because some other things depend on it, and the next thing you know. So it's better to just have the header file never complain about that, never let that happen, than to make it somebody else's responsibility to make sure it never happened through the includes.

So I've got five member functions here that I'm gonna have to implement. And now I need to think about what the private data, this guy's gonna look like. So now, we are the

low-level implementer. We're not building on anything else right now, because myvector is kind of the bottommost piece of things here. We've got nothing out there except for the raw array to do the job for us.

So typically, the most compatible mechanism to map something like vector onto is the array. You know it has contiguous memory, it allows you to do direct access to things by index, and so that's where we're gonna start. And we'll come back and talk about that, whether that's the only option we have here. But what I'm gonna put in here is a pointer to a string, or in this case it's gonna be a pointer to a string array.

And so in C++ those both look the same, this says arr is a single pointer that points to a string in memory, which in our situation is actually gonna be a whole sequence of strings in memory kind of one after another. The array has no knowledge of its length so we're gonna build is using new and things like that. It will not know how big it is. It will be our job to manually track that.

So I'm gonna go ahead and have two fields to go with it, and I'm gonna show you why I have two in a minute. But instead of having just a length field that says the array is this long, I'm actually gonna track two integers on it because likely the thing I'm gonna do with the array is I'm gonna allocate it to a certain size and then kind of fill it up. And then when it gets totally filled, then I'm gonna do a process of resizing it and enlarging it.

And so, at any given point, there may be a little bit excess capacity, a little slack in the system. So we might start by making it ten long and then fill it up, and then as we get to size 10, make it 20 or something like that. So at any point, we'll need to know things about that array: how many of the slots are actually used, so that'll be the first five slots are in use; and then the number of allocated will tell me how many other – how many total slots do I have, so how many slots can I use before I will have run out of space.

Both of those. Okay, so there's my interface. So these things all public because anybody can use them; these things private because they're gonna be part of my implementation; I don't want anybody mucking with those or directly accessing them at all, so I put them down there.

All right, now I go to myvector.cpp. So it includes myvector.h, so that it already has seen the class interface so it knows when we're talking about when we start trying to implement methods on the myvector class. And I'm gonna start with myvector's constructor, and the goal of the constructor will be to setup the instance variables of this object into a state that makes sense.

So what I'm gonna choose to do is allocate my array to some size. I'm gonna pick ten. So I'm gonna say I want space for ten strings. I want to record that I made space for ten strings so I know the num allocated, the size of my array right now is ten, and the number used is zero. So that means that when someone makes a call, like a declaration, that says myvector V, so if I'm back over here in my main.

Say myvector V, but the process of constructing that vector V will cause the constructor to get called, will cause a ten-member string element array to get allocated out in the heap that's pointed to by arr, and then it will set those two integers to know that there's ten of them and zero of them are used. So just to kind of all as part of the machinery of declaring, the constructor is just wired into that so we get setup, ready to go, with some empty space set aside.

So to go with that, I'm gonna go ahead and add the destructor right along side of it, which is I need to be in charge of cleaning up my dynamic allocation. I allocated that with the new bracket, the new that allocates out in the heap that uses the bracket form has a matching delete bracket that says delete a whole array's worth of data, so not just one string out there but a sequence of them.

We don't have to tell it the size; it actually does – as much as I said, it doesn't its size. Well somewhere in the internals, it does know the size but it never exposes it to us. So in fact, once I delete [inaudible] array, it knows how much space is there to cleanup.

Yeah?

Student: Are you just temporarily setting it up so the vector only works on strings?

Instructor (Julie Zelenski):Yes, we are.

Student: Okay.

Instructor (Julie Zelenski):Yes. We're gonna come back and fix that, but I think it's easier maybe to see it one time on a fixed type and then say, "Well, what happens when you go to template? What things change?" And we'll see all the places that we have to make modifications.

So I have myvector size. Which variable's the one that tells us about size? I got none used. I got none allocated. Which is the right one?

Student: Num used.

Instructor (Julie Zelenski):Num used, that is exactly right. So num allocated turned out to be something we only will use internally. That's not gonna – no one's gonna see that or know about it, but it is – the num used tracks how many elements have actually been put in there.

Then I write myvector add. So I'm gonna write one line of code, then I'm gonna come back and think about it for a second. I could say arr num used ++ = S, so tight little line that has a lot of stuff wrapped up in it.

Using the brackets on the dynamic array here are saying, "Take the array and right to the num used slot, post-incrementing it so it's a slide effect." So if num used is zero to start

with, this has the effect of saying array of, and then the num used ++ returns the value before incrementing.

So it evaluates to zero, but as a slide effect the next use of num used will now be one. So that's exactly what we want, we want to write the slot zero and then have num used be one in subsequent usage. And then we assign that to be S, so it'll put it in the right spot of the array. So once num used is five, so that means the zero through four slots are used. It'll write to the fifth slot and then up the num used to be six, so we'll know our size is now up by one.

What else does add need to do? Is it good?

Student: Needs to make sure that we have that [inaudible].

Instructor (Julie Zelenski):It'd better make sure we have some space. So I'm gonna do this right now. I'm gonna say if num used is equal to num allocated, I'm gonna raise an error. I'm gonna come back to this but I'm gonna say – or for this first pass, we're gonna make it so it just doesn't grow. Picked some arbitrary size, it got that big, and then it ran out of space.

Question?

Student: So when the – for the vector zero, first time it gets called it's actually gonna be placed in index one of the array?

Instructor (Julie Zelenski):So it's in place of index zero. So num used ++, any variable ++ returns the – it evaluates to the value before it increments, and then as a side effect, kind of in the next step, it will update the value. So there is a difference between the ++ num used and the num ++ form. Sometimes it's a little bit of an obscure detail we don't spend a lot of time on, but ++ num says first increment it and then tell me what the newly incremented value is. Num ++ says tell me what the value is right now and then, as a side effect, increment it for later use.

Student: So the num used gets changed in that?

Instructor (Julie Zelenski):It does get changed in that expression, but the expression happened to evaluate to the value before it did that change.

Question?

Student: And is it really necessary to have myvector:: or is –

Instructor (Julie Zelenski):Oh yeah, yeah, yeah, yeah.

Student:– there any way for –

Instructor (Julie Zelenski):So if I take this out, and I could show you what happens if I do this, is what the compiler's gonna think that is, is it thinks, "Oh, there's just a function called size that take the arguments and returns them in." And it doesn't exist in any context; it's just a global function. And then I try to compile it, if I go ahead and do it, it's gonna say, "Num used? What's num used? Where did num used come from?" So if I click on this it'll say, "Yeah, num used was not declared in this scope." It has no idea where that came from.

So there is no mechanism in C++ to identify this as being a member function other than scoping it with the right names. You say this is the size function that is part of the myvector class. And now it has the right context to interpret that and it knows that when you talk about num used, "Oh, there's a private data member that matches that. That must be the receiver's num used."

So one other member function we got to go, which is the one that returns, of I do this, it almost but not quite, looks like what I wanted. So I say return arrays of index. They said get in the next and returned the string that was at that index. Is there anything else I might want to do here?

Student: Check and see if the index is [inaudible].

Instructor (Julie Zelenski):Well, let me check and see if that index is valid. Now, this is one of those cases where it's like you could just sort of take the stance that says, "Well, it's not my job." If you ask me to get you know something at index 25, and there are only four of them, that's your own fault and you deserve what you get.

And that is certainly the way a lot of professional libraries work because this, if I added a step here that does a check, it means that every check costs a little bit more. When you go in to get something out of the vector, it's gonna have to look and make sure you were doing it correctly. And there are people who believe that if it's like riding a motorcycle without a helmet, that's your own choice.

We're gonna actually bullet proof; we're gonna make sure that the index isn't whack. And I'm gonna go ahead and use my own size there. I'm gonna say if the index is less zero or it's greater than or equal to, and I'm gonna use size. I could use num used there but there are also reasons that if I use my own member functions and then later somehow I change that name or I change how I calculate the size, let's say I change it to where I use a linked list within an array and I'm managing the size differently, that if I have used size in this context, it will just change along with that because it's actually depending on the other part of the interface that I'll update.

And so I'm gonna go ahead and do that rather than directly asking my variable. I'll say if that happens I say error, out of bounds. And on that, that error will cause the program to halt right down there, so I don't have to worry about it, in that case, getting down to the return.

So I feel okay about this. Not too much code to get kind of a little bit of something up and running here. Let's go over and write something to test. Add Jason, and here we go. Okay, so I put some things in there and I'm gonna see if it'll let me get them back out.

And before I get any further, I might as well test the code I have. Right? This is – one of the things about designing a class is it's pretty hard to write any one piece and test it by itself because often there's a relationship: the constructor and then some adding and then some getting stuff back out. So it's a little bit more code than I typically would like to write and not have a chance to test, having all five of these member functions kind of at first.

So if I run this test and it doesn't work, it's like well which part what was wrong? Was the constructor wrong? Was the add wrong? Was the size wrong? You know there's a bunch of places to look, but unfortunately, they're kind of all interrelated in a way that makes it a little hard to have them independently worked out. But then subsequent thing, hopefully I can add the like the insert at method without having to add a lot more before I test.

Okay. So I run this guy and it says Jason, Illia, Nathan. Feel good, I feel good, I feel smart. Put them in, in that order. See me get them out in that order. I might try some things that I'm hoping will cause my thing to blow up. Why don't I get at ten? Let's see, I like to be sure, so I want you to tell me what's at the tenth slot in that vector. And it's, ooh, out of bounds, just like I was hoping for. Oh, I like to see that. Right?

What happens if I put in enough strings that I run out of memory? And we talked about a thing – it's set it to be ten right now. Why don't I just make it smaller, that'll way it'll make it easier for me. So I say, "Oh how about this?" I'm only gonna make space for two things. And it just: error, out of space. Right? That happened before it got to printing anything. It tried to add the first one. It added the second one. Then it went to add that third one and it said, "Oh okay, we run out of space. We only had space for two set aside, you asked me to put a third in. I had no room."

So at least the kind of simple behaviors that I put in here seem to kind of show evidence that we've got a little part of this up and running. What I'm gonna fix first is this out of space thing. So it would be pretty bogus and pretty unusual for a collection class like this to have some sort of fixed limit. Right? It wouldn't – you know it'd be very unusual to say well it's always gonna hold exactly 10 things or 100 things or even a 1,000 things. Right?

You know one way you might design it is you could imagine adding an argument to the constructor that said, "Well, how many things do you plan on putting in there? And then I'll allocate it to that. And then when we run out of space, you know you're hosed." But certainly a better strategy that kind of solves more general case problems would be like, "Oh let's grow. When we run out of space, let's make some more space."

Okay, let's think about what it takes to make more space in using pointers. So what a vector really looks like is it has three fields, the arr, the num used, and the num allocated. When I declare one, the way it's being setup right now, it's over here and it's allocating space for some number, let's say it's ten again, and then marking it as zero. Then as it fills these things up, it puts strings in each of these slots and it starts updating this number, eventually it gets to where all of them are filled. That when num used equals num allocated, it means that however much space I set aside, every one of those slots is now in use.

So when that happens, it's gonna be time to make a bigger array. There is not a mechanism in C++ that says take my array and just make it bigger where it is. That the way we'll have to do this is, we'll have to make a bigger array, copy over what we have, and then, you know, have it add it on by making a bigger array full of space.

So what we'll do is we'll make something that's like twice as big, I'm just gonna draw it this way since I'm running out of board space, and it's got ten slots and then ten more. And then I will copy over all these guys that I have up to the end, and then I have these new spaces at the end.

And so I will have to reset my pointer to point to there, update my num allocated, let's say to be twice as big or something, and then delete this old region that I'm no longer using. So we're gonna see an allocate, a copy, a delete, and then kind of resetting our fields to know what we just did. So I'm gonna make a private helper to do all this, and I'll just call that enlarge capacity here.

Question?

Student: Is this like [inaudible]?

Instructor (Julie Zelenski):Well it is – it can be, is the truth. So you're getting a little bit ahead of us. But in the sense that like, you know, if the array has a thousand elements and now we got to put that thousand-first thing in, it's gonna take all thousand and copy them over and enlarge the space. So in effect what is typically an O of one operation, just tacking something on the end, every now and then is gonna take a whole hit of an end operation when it does the copy.

But one way to think about that is that is every now it's really expensive but kind of if you think of it across the whole space, that you got a – let's say you started at 1,000 and then you doubled to 2,000, that the first 1,000 never paid that cost. And then all of a sudden one of them paid it but then now you don't pay it again for another 1,000. But if you kind of divided that cost, sort of amortized it across all those adds, that it was a – it didn't change the overall constant running time.

So you have to – you kind of think of it maybe in the picture. It does mean every now and then though one of them is gonna surprise you with how slow it is, but hopefully that's few and far between, if we've chosen our allocation strategy well.

So I'm gonna say – actually, I'm gonna do this. I'm gonna call this thing actually double capacity. So one strategy that often is used is you could go in little chunks. You could go like ten at a time all the time. But if the array's gonna get very big, going by tens will take you a lot of time. You might sort of use sometimes the indication of, "Well how big is it already?" Then if that's about the size it seems to be, why don't we go ahead and just double.

So if we start at 10, you know, and then we go to 20, then we go to 40, then we to 80, then as we get to bigger and bigger sizes, we'll make the kind of assumption that: well, given how big it is already, it's not – it wouldn't surprise us if it got a lot bigger. So let's just go ahead and allocate twice what we have as a way of kind of predicting that it's pretty big now, it might get even bigger in the future.

That isn't necessarily the only strategy we could use, but it's one that actually often makes pretty decent sense. And then at any given point, you'll know that the vector will be somewhere between full and half capacity, is what you're setting up for.

So let's do this. Oops, it's not ints; string bigger equals, and I make a new string array that is num allocated times two, so twice as big as the one we have. And now I go through and I copy. I'm copying from my old array to my bigger one, all of num used. In this case, num used and num allocated are exactly the same thing, so. And then I'm gonna reset my fields as needed.

I'm gonna delete my old array because I'm now done with it. I'm gonna set my pointer to this, so that's doing pointer assignment only. Right? So I deleted the old space, I've copied everything I needed out of it, I'm now resetting my pointer to the bigger one, and then I'm setting my num allocated to be twice as big, num used doesn't change. Okay, so went through the process of building all these things. And as noted, that is gonna cost us something linear in the number of elements that we currently have, so.

So this guy is intended to be a private method. I don't want people outside of the vector being able to call this, so I'm actually gonna list this in the private section. It's not as common that you'll see member functions listed down here. But in this case, it's appropriate for something that you plan to use internally as a helper but you don't want anybody outside just to be calling double capacity when they feel like it.

Question?

Student: So normally [inaudible] an array, you couldn't actually declare a size like that, though, right, with a variable?

Instructor (Julie Zelenski):You know I don't understand the question. Try that again.

Student: You know in this case to enlarge it, you use a variable as the – for the size of the array. When you normally declare an array, you couldn't do that, right?

Instructor (Julie Zelenski):Well if you did it –

Student: It has to be a constant.

Instructor (Julie Zelenski):So like if you used this form of an array, you know, where you declared it like this. Did you see what I just did?

Student: Yeah.

Instructor (Julie Zelenski):Yeah. That way won't work, right? That way is fixed size, nothing you can do about it. So I'm usually totally the dynamic array at all times, so that everything –

Student: So you do it when you're declaring it on a heap?

Instructor (Julie Zelenski):Yes.

Student: Okay.

Instructor (Julie Zelenski):Yes, exactly. All I have in the – stored in the vector itself is a pointer to an array elsewhere. And that array in the heap gives me the flexibility to make it as big as I want, as small as I want, to change its size, to change where it points to, you know all the – the dynamic arrays typically just give you a lot more flexibility than a static array.

Student: That's really stack heap.

Instructor (Julie Zelenski):It is. Array is a stack heap thing. When you put on the stack, you had to say how big it is at compile time and you can't change it. The heap let's you say, "I need it bigger, I need it smaller, I need to move it," in a way that the stack just doesn't give you that at all.

So when I go back to myvector.cpp, the place I want to put in my double capacity here is when num used is equal to num allocated, but what I want to do is call double capacity there. After I've done that, num allocated should've gone up by a factor of two. Space will be there at that point. I know that num used is a fine slot to now use to assign that next thing. So whenever we're out of space, we'll make some more space.

And so I'm gonna – right now, I think my allocated is still set at two. That's a fine place. I'd like it to be kind of small because I'd like to test kind of the – some of the initial allocations. So I'll go ahead and add a couple more people so I can see that I know that – at that point, if I've gotten to five, I'm gonna have to double once and then again to get there. And let's, I'll take out my error case here, see that I've managed to allocate and move stuff around. Got my five names back out without running into anything crazy, so that makes me feel good about what I got there.

So I could go on to show you what insert and remove do; I think I'm probably gonna skip that because I'd rather talk about the template thing. But I could just tell you – I could sketch the [inaudible]: what does insert at do, what does remove at do? Basically, that they're doing the string – the array shuffling for you. If you say insert at some position, it has to move everything down by one and then put in there. Whereas at is actually just tacking it onto the end of the existing ones. The insert and remove have to do the shuffle to either close up the space or open up a space.

They'll probably both need to look at the capacity as well. That the – if you're inserting and you're already at capacity, you better double before you start. And then the remove at, also may actually want to have a shrink capacity. Where when we realize we previously were allocated much larger and we've gotten a lot smaller, should we take away some of that space.

A lot of times the implementations don't bother with that case. They figure, "Ah, it's already allocated, just keep it around. You might need it again later." So it may be that actually we just leave it over-allocated, even when we've deleted a lot of elements, but if we were being tidy we could take an effort there.

What I want to do is switch it to a template. So if you have questions about the code I have right here, now would be a really good time to ask before I start mucking it up. Way in the back?

Student:[Inaudible].

Instructor (Julie Zelenski):I will. You know that's a really good idea because if I – at this point, I'll start to change it and then it's gonna be something else before we're all done. So let me take a snapshot of what we have here so that I – before I destroy it.

Question?

Student: When does the deconstructor get called?

Instructor (Julie Zelenski):Okay, so the destructor gets called in the most – there are two cases it gets called. One is when the – so the constructor gets called when you declare it, and then destructor gets called when it goes out of scope. So at the opening brace of the block where you declared it in, is when the constructor's happening, and then when you leave that. So in the case of this program, it's at the end of May, but and if it were in some called function or in the body of a for loop or something, it would get called when you enter the loop and then called as – destroyed as it left.

For things that were allocated out of the heap, so if I had a myvector new myvector, it would explicitly when I called delete that thing when I was done with it. So especially when the variable that holds it is going away, right? Either because you're deleting it out of the heap or because it's going out of scope on the stack.

Here?

Student: When you have string star array, arr, it's a pointer to a single string and then later you can use a new statement to link it to –

Instructor (Julie Zelenski):Yeah.

Student:– an array. So how does it know when it –

Instructor (Julie Zelenski):How does it know? It doesn't, is the truth, is that when you say something's a pointer to a string, the only guarantee is really there and says it's an address of someplace where a string lives in memory. Whether there's one string there or a whole sequence of strings is something you decide and you maintain, and so the compiler doesn't distinguish those two cases. It does not know that you made exactly 1 string at the other end of this or 500.

And so, that's why this idea of tracking the num used and num allocated becomes our job; that they look exactly the same in terms of how it interprets them. It says, "Well it's the address of where a string lives in memory, or maybe it's a sequence. I don't know." And so it lets you use the brackets and the new and stuff on a pointer without distinguishing – having any mechanism in the language to say this is a single pointer and this is an array pointer. They look the same.

Student: So you could use the bracket notion on a single pointer?

Instructor (Julie Zelenski):You certainly can.

Student: And then –

Instructor (Julie Zelenski):It's not a good idea but you can do it. So legally in the language, it just makes no distinction between those. They are really commingled in way that – and so that's one of the more surprising features of C and C++ and one's that a little bit hard to get your head around is it doesn't track that it's a pointer to a single thing versus a pointer to an array. That they're both just pointers and it's mine. And that allows opportunity for errors, when you mistreat them as the other type for example.

Question?

Student: Can you go over in your secret view file in the [inaudible] of what's going on with [inaudible] real quick because it's just –

Instructor (Julie Zelenski):Yeah, yeah, yeah, so let me – it was basically the thing I drew over here, but I'll do it again just to watch it, is that let's start – let's imagine I have a slightly smaller num allocated so it's a little less for me to write.

So let's say that I'm gonna use a num allocated of two, so this allocates two. So when I construct it, it makes a block that holds two things and num used is zero. So I do two adds: I add A, it increments num used to one; I had B, it increments num used to two. I try to add C. It says, "Oh, well num used equals num allocated. We're gonna go to double capacity now."

So double capacity has this little local variable called bigger. And it says bigger is gonna be something that is four strings worth in an array, so it gets four out there. It does a full loop to copy the contents of the old array on top of the initial part of this array; so it copies over the A and the B, into there. And then it goes, "Okay, I'm done with this old part. So let me go ahead and delete that."

And then it resets the arr to point to this new one down here, where bigger was. So now, we got to aliases of the same location. And then it sets my num allocated to say and now what you've got there is something that holds four slots. And then that used call here says, "Okay and now writer the C into the slot at three."

So the process here is the only way to enlarge an array in C++ is to make a bigger one, copy what you had, and then by virtue of you having made a bigger array to start with, you have some more slack that you didn't have before.

Daniel?

Student: How does it delete arr with a star?

Instructor (Julie Zelenski):You know it has to do with just delete takes a pointer. It does it – so a star arr is a string, arr is a pointer to a string. So both forms of delete, delete and delete bracket –

Student: So conceptually –

Instructor (Julie Zelenski):– a pointer.

Student:– there is a start there because it's delete –

Instructor (Julie Zelenski):Well effectively, yeah. It's delete the thing at the other end of the pointer, really. But it's funny. Delete says take this address and reclaim its contents. And so it doesn't really operate on a string, per se, it operates on the storage where that string is. And so I don't know whether you want to call that is there an implicit star there or not, it really is about the pointer though rather than the contents. So saying that address has some memory associated with it, reclaim that memory.

Student: If I could raise –

Instructor (Julie Zelenski):Uh-huh.

Student: So when you're first declaring or when you're making a pointer like string bigger, string star bigger, you have to declare it with the star notion. But then later on, you don't ever have to use that again?

Instructor (Julie Zelenski):You pretty much won't see that star used again. Right? It's interesting that things like bigger sub I and erase sub I implicitly have a D reference in them. And that can be misleading. You think, "Well how come I'm never actually using that star again on that thing to get back to the strings that are out there?"

And it has to do with the fact that the bracket notation kind of implicitly D references in it. If I did a star bigger, it would actually have the effect of giving me bigger sub zero, it turns out. You can use that notation but it's not that common to need to.

Student: And so down on the last, one line up from the bottom, it says array equals bigger. You don't have to –

Instructor (Julie Zelenski):Yeah, if you did that, if I did say –

Student: If you said array –

Instructor (Julie Zelenski):Star arr equals star bigger, I would not be getting what I want. Right? What it would be doing is it would say follow bigger and see what's at the other end, so that would follow bigger and get that string A. And then it would say follow ARR and overwrite it with that A, so it would actually have the effect of only copying the first string from bigger on top of the first string of array. But array would still point to where it was, bigger would still point to where it was, and they would – we would've not have updated our, the pointer we really wanted to point to the new array.

So there is a difference. Without the star, we're talking about the changing the pointers; with the star, we're talking about the strings at the other end. And so we're – this is a string assignment. It says assign one string to another. Without the star on it, it's like assign one pointer to another; make two pointers point to the same place. When you're done with this, bigger and arr will be aliases for the same location. That's a very important question though to get kind of what that star's doing for you.

Here?

Student: After arr is bigger, can you delete bigger after that?

Instructor (Julie Zelenski):If I deleted bigger, at that point arr is pointing to the same place. And so remember that having two or three or ten pointers all at the same place, if you delete one of them, they actually effectively are deleted. The delete really deletes the storage out here. And then if I did that, it would cause arr to then be pointing to this piece of memory, and not a good scene will come from that. It means that when it later goes back in there and starts trying to read and write to that contents at any moment it could kind of shift underneath you. You don't own it any more; it's not reserved for your use.

So if we did that, we'd get ourselves into trouble. All right? So there should basically be a one-to-one correspondence between things you new and things you delete. And so in the myvector case, we newed something in the constructor that we're gonna delete in the destructor. If at some point along the way we got rid of our old one and get a new one, that's the new the one that's gonna get deleted. If we deleted it midstream here, we would just be asking for havoc when we start accessing that deleted memory.

Way in the back?

Student: Is it possible to create a pointer to something – a pointer to the address that's one off the end of the original array and then just create an array just off the end there?

Instructor (Julie Zelenski):Not really. So new doesn't let you decide where you want something. So you're point being to think, "Well I can tell you what this address is, can I just make space right there and then I won't have to copy." And it turns out new just doesn't give you that kind of control. You ask it for space, it finds it wherever it has and you can't – there isn't even a mechanism where you could suggest where you'd like it to be. You could say, "Well let that place right there would be really handy. Could you please give me that one?" It just doesn't give it you.

So you're – this is the way that typically you have to manage a dynamic array. And this is actually one of the big drawbacks to continuous memory as a reason for implementing things is that the fact that it has to maintain contiguousness. Means you have to shuffle and move and copy this block without the flexibility of something like a link list where every cell is independently manipulated.

There?

Student: Why does [inaudible] the delete brackets arr as delete just arr?

Instructor (Julie Zelenski):So the difference is that if you allocated something with new string bracket, new something bracket, you need a delete bracket. If you actually use delete without the brackets, it thinks there's a single pointer and there's only one string at the other end. Where delete brackets says delete the whole gob of strings that are there.

If you don't do it, it's not – the consequence is not that big; it's like some memory gets orphaned, some things don't happen. But to be totally correct, they go hand in hand: if you use new with brackets, use delete with no brackets, if you use new with brackets, use delete with brackets. So it's either both with brackets or both without.

Student: So even though arr is just point to the first place, the brackets knows the –

Instructor (Julie Zelenski):Yeah, it does. And so that kind of makes me feel like I'm a lair because I said well the array doesn't know its length. Well it does somehow. Internally it is maintaining some housekeeping but it doesn't expose it to you. So when you say delete bracket arr it knows, "Oh, there's a bunch of strings and I got to do a

bunch of cleanup on them." But it doesn't ever expose that information back to you. It doesn't let you depend on it, so it's up to you to maintain that information redundantly with it.

All right, let me see if I can make it a template. I probably can't do this actually fast enough to get it all done today, but we can at least get started on it. So then, I introduce a template header and I make up the name that I want here, so same class header now other than typing elem type. Then I look through my interface and I see places where I previously had said it's strings, it's strings, it's storing strings. And I say it's not actually storing strings; it's gonna store elem type things, it's gonna return elem type things and it's going to have an array of elem type things.

So I think that's everything that happened to the interface. Let me see if I see any other places that I – so the interface part is kind of small. There's one other change I'm gonna have to make to it but I'm gonna come back to it. I'm gonna look at the code at the other side for a second. And I say, "Okay, well that wasn't so bad."

Now it turns out that it gets a little bit goopier over here because that template type name has to go on every one of these: introduce them to the template type and elem type. And now there's another place where it needs to show up. So the full syntax for this is now saying this is a template function, depending on elem type, and it's actually for the myvector who is being – we are writing the myvector constructor for something whose name is myvector angle bracket elem type.

So there's gonna be a lot of this goo. Every one of these is kinda change its form, from just looking like the ordinary myvector class scope doesn't really exist any more. Myvector is now a template for which there's a lot of different class scopes, one for each kind of thing being stored. So myvector int is different than myvector string. So we say, "Well, if you were building the myvector constructor for myvector string, it looks like this." Or you know having filled an elem type with those strings.

So everywhere I was using string, I got to change to elem type in the body as well. And then I kind of take this guy and use it in a bunch of places. I'm gonna use it here and then I'm gonna have to do it down here, on that side, do it here, and it's gonna return something of elem type, here. It's a little bit of a mess to do this, and the code definitely gets a little bit goopier as a result of this. It doesn't look quite as pretty as it did when it wasn't a template, but it becomes a lot more useful.

Okay. Then I need to look for places that I used a string. And every place where I was using string, assuming that's what I was storing, it now actually turns into elem type. So my pointers and the kind of array I'm allocating is actually now made into elem type. The rest of the code actually didn't say anything specific about what's its doing, just copying things from one array to another. And now, depending on what the arrays are, it's copying ints or strings or doubles.

And then other places in the interface where I'm doing add or I'm going get at, I have to be describing the things that are coming in and out as elem type so that they can be matched to whatever the client's using. I think the rest of it looks okay.

Student: Why do you have to write template type name, and elem type above every –

Instructor (Julie Zelenski):Because you just have to, because it's C++. Because the thing is, that piece of code is, itself, a template, so these are like little mini-templates. So that I had the interface, which said here's the template pattern for the interface, and each of these says when you're ready to make the size member function for a vector of int, it comes off this template. So this template describes what the size member function looks like for any of the myvectors you might instantiate. And it describes the template because, in fact, we need to build a new size for ints versus doubles versus strings.

It's even funny because you think of my size like, "Well size doesn't even use anything related to the elem type." But in fact, each of the member functions is kinda specific. It's not just a myvector size; it's the myvector int size, the myvector string size. And that for some of the member functions it's quite obvious why you need a distinct copy. Get at returns an int in some cases and a double in others; but even though ones that don't appear to have any dependence on the elem type, actually are separated into their own individual versions.

So I think I got all of that fixed, and then I'm gonna have to do one thing that's gonna seem really quirky. And it is very quirky but it is C++. Let me show you what I'm gonna do. Is I'm going [inaudible] out of the project. Okay, stop compiling that. And I'm gonna change how it is that myvector gets compiled by doing this.

Okay. Take a deep breath. This is really just an oddity of C++. So the situation is this: that templates aren't really compiled ahead of time, templates are just patterns. You know? They like describe a recipe for how you would build a myvector class. But you can't just compile myvector and be done with it because until the client uses it, you don't know what kind of myvectors you're building. Are they myvectors of ints or strings or pseudo structures?

So it turns out that the myvector really needs to get compiled at the usage, at the instantiation. When you're ready to make a myvector of students, it then needs to see all the code for myvector so it can go build you a myvector for students on the fly. In order to see that code, it actually has to be present in a different way than most code.

Most code is compiled, instead of .cpp, it just gets compiled once and once for all. The random library, random integer doesn't change for anybody usage, there's a random.cpp. It compiled the function. You're done. So the template code does not get compiled ahead of time. It doesn't get listed in the project. What happens is the .h typically has not only the interface, but actually all the code.

And so the two ways to get the code in here, one way is I could've just put all the code down here. And that's the way a lot of professional code gets written, it has the interface followed by all the template code right after it. I like to keep us thinking about the interface and the implementation being separate, so I'm actually taking the interface and keeping the .h, keeping this [inaudible] over here in a .cpp.

And then I'm using the #include mechanism in a very unusual way. That almost never would you want to, in a regular usage, to #include another .cpp file. For templates, we're making an exception. And we're saying, "Well in this case, because I really need that code there," the #include mechanism is basically saying go take the contents of this thing and just dump it in here.

It really is an include mechanism. It says, "Go get this file and take its text contents and dump it right into this file." So that when somebody's trying to import the myvector.h, they're getting both the interface plus all the code that we'll generate a pattern from.

So this is definitely just a quirk. There's no consistency between how other languages that do stuff like this expect this. This is just unique to C++ and its compilation mechanisms that require this kind of sort of slight variation in handling. So we'll see this for all the templates we'll use is that they will not be included as normal cpp files, they will get included in the .h. And there is this exact pattern, which is reproduced for every one of the ones in the text. You'll see it on stack and queue and integer. That it becomes the kind of boilerplate we'll use when making a template.

So in general, I'd say be very wary of anything that looks like this. This is not a normal thing to do and we're doing it just specifically to kind of keep up the illusion that the interface and the implementation are kept separate because there's actually some good thinking that comes from that. But the way the compiler sees it, it doesn't want them to be separate, and so we have to accommodate it with this little hack, let's say, here.

So once I've done that, I go back to lecture. If I change this to be myvector string, I'm hoping that everything will still work. Which it did, kind of amazingly.

Daniel?

Student: So where is the myvector.cpp file at?

Instructor (Julie Zelenski):So it's actually just living in the same directory with this, the way myvector.h is. So typically, like your .h files are just sitting in the directory – .cpp is sitting in the same directory. That's where it's gonna look for it when it goes #including is in the kind of local contents.

Student: But like where is that? Like is it in resources?

Instructor (Julie Zelenski):No, it's just – if you look – you know this is the directory I have, this is the contents of my, you know all my files, my project files, where the thing gets dumped.

Student: Oh, okay.

Instructor (Julie Zelenski):It's just sitting there with all the code.

And I should be able to change this now to put some numbers in and have it do both. I just did it with strings and now I'm gonna do it with ints. And voila, we now have something that holds ints. So a certain amount of goo that went from the simple form to the template form, but a lot of power gained. Suddenly we took this thing that was one purpose, that held strings only, and you just made it to where it can hold anything you can think of to stick in there by making little machinations in the syntax there.

So we'll see a lot more of this. It's not the first, nor the last, syntax that – for templates that we're gonna be playing with, but that will be next week. Come to Café with me, if you got some time. Otherwise, I will see you on Monday.

[End of Audio]

Duration: 53 minutes

ProgrammingAbstractions-Lecture19

Instructor (Julie Zelenski):I've got the big box of graded exams, which I was hoping we'd get back to you today. And we do have them back, and we have the solution and stuff to go back with it, so I'll spend a little time at the end kind of talking about it. But it's – just so you know, it's something behind us now.

Assignment 5 is due today. So we'll do a little query on how much time got spend on Assignment 5. Hopefully this was a little bit lighter than the other ones, but we never know until we actually ask. How many people managed to do that in under ten hours? Okay, that's a big chunk. All right, I'd say over half of you. How many 10 to 15? A few people, kind of on that thing. More than 15? Way in the back, all right. When you – how – was it fun, a fun time? Were you playing with an algorithm or just getting it done?

Student:[Inaudible].

Instructor (Julie Zelenski):Okay. This next week assignment, Assignment 6, is now turning our attention from client-side programming to implementation-side programming. So we've done a lot of work so far that is dependent on using vector and grid and stack and queue and lexicon, and all sort of things. Now that we're starting to kind of open up that black box and look under the hood and see what's going on, we're now gonna change roles.

And so this is the – where things actually get a little bit hairier on the implementation side, is that suddenly the management of memory, the exposed pointers, the use of link lists, the use of raw array, suddenly is on your plate. You are implementing those things to provide the convenience that you've been counting on for the many weeks prior to this. So it does require kind of, I think, a real attention to detail and kind of perseverance to get through some of these things.

So the priority queue is the assignment you'll be working on, so today we'll talk about stack and queue as implementation topics. And then you'll be taking the idea of the queue and kind of twisting it a little bit into this form called the priority queue. And there are two implementations we give you that you just get to play with, and then there are two implementations that you'll get to write.

There's not a lot of code, actually, in either of them, but the code that's there is very dense. So kind of like back to that recursion things where it's like, well, any particular piece of it is not – it's not the length that gets you, it's the intensity and the kind of management of difficult data structures that's gonna be this week. So it's due a week-and-a-half from today. And I do think of this as being one of your heftier pieces, so this is not one you want to put off to the last minute try to get that done.

What I'm gonna do today is I'm gonna finish talking about vector. We'd gotten to where we had a skeleton vector that was – mostly appeared to be working at the end of Friday's lecture, but there's actually a couple things about it that we need to fix before we can

move away from that. And then we'll look at stack and queue as kind of our next thing: how do you implement a stacked queue, what options do you have there?

So the reading to go along with this, this is pretty much all Chapter 10: Class Templates, and then the next thing we'll look at is the editor buffer case study, which is covered in Chapter 9.

Student:I have a [inaudible] that relates to sorting. Are there metrics that determine like how well something is sorted or unsorted?

Instructor (Julie Zelenski):What do you mean by "how well" I guess is –

Student: Well I don't know. I mean something like – I was just gonna imagine, for instance, that if you could define some kind of scale that like at one end it would be completely sorted [inaudible] and then on the other end it would be completely unsorted, although I mean that's a hard to define –

Instructor (Julie Zelenski):Yeah, well and – I think you get the choice of how you define that. Some people, for example, would use things like, well what are the percentage of elements that are already in the correct position, is one measure of sorted. Right? And so that would be interesting in sorts that are trying to move things to the right place if it's already in exactly the right place.

Other times what you're more interested in is things that are out of order, so it might be that you know if you had the data sorted, like the front half was sorted and the back half was sorted. A lot of things are in the wrong place but it still is pretty sorted by – and so if you were considering some other variant of it, which is what are the number of sorted runs within it, so if you look at contiguous sequences. And sometimes you count things, like what's called the number of inversions, which is the number of pairs that are out of order relative to each other.

And so depending on kind of what things you're trying to look for, and that probably has a lot to do with what algorithm you're using, is what things can it take advantage of. Does it – is it better at taking advantage of things that are already in the right place? Or better at taking of things that are already in an ordered sequence with its neighbors, as to whether that would allow the algorithm perform more efficiently on that dataset.

But there is not one metric that everyone uses when they're saying how much sorted is, you know what percentage or what – how close to sorted or unsorted it is. It's kinda a matter of how you define your term as to what you mean by close.

Student: Right. But algorithms actually use something like that sometimes?

Instructor (Julie Zelenski):Sometimes, right? I mean some algorithms tend to actually depend on those features. So for example, there's one that looks for sort of sorted runs and then merges them. It's called a natural merge, so where you're looking for existing

sorted runs and then merging them from the bottom-up. And it's like that one, depending on the length and the number of runs in there, its runtime could be expressed using those metrics, which themselves are a measure of how close it is to being in the order that merge – this natural merge sort wants it to be.

So sometimes they do use that as a way of telling you about how does this perform, given this number of inversions or this number of placements and things like that. So it is one of the factors you're kind of looking at when you're trading off algorithms that depend on the ordering being given to start with, to just say things about their performance.

Anybody have an interesting sort that they actually had a fun time with, they learned something cool that they would love to have an audience to talk about? Way in the back?

Student: Heap sort is kind of cool.

Instructor (Julie Zelenski):Heap sort is very cool. Heap sort's a great algorithm, so anybody who – and how many other people actually ended up using heap sort as the one they did? You guys are in luck because it turns out the – part of the assignment review this week involves using a heap, so you guys are like already one step ahead of the game.

But heap is a great data structure and it actually has an awesome sort that actually has properties of N log N, so and actually it is a reliable N log N, doesn't have this kind of worst case degenerate thing hiding in there the way quick sort does, doesn't use any extra storage, auxiliary storage outside of it. It has a pretty neat algorithm that's kind of based on using the array itself as this notion of a heap, kind of a tree. So that's a great sort actually, it's kind of – I think one's that actually underappreciated for what it does.

Anybody else have a cool sort? Did anybody write a sort they thought was particularly fast when they were done? Sorts that beat our sorts that were really – you'd use in practice for fun? Did you guys all just write – what's the one that's alphabetically first on the list? This usually happens, whatever one I use first, it turns out half the class does. What was the one I listed first?

Student: Bingo.

Instructor (Julie Zelenski):Bingo. Did I put bingo first? Oh, that's a dumb sort. I think the last time I put comb sort first. And then it turned out everybody did comb sort because it was like the one that was in the front or something.

Student: So I found you could do the sorting algorithm called flash sort, which only looks for integers that uses an equation, basically to determine about where each integer should be in the final sorted array. And I wrote an implementation of it. I'm not sort how well it worked out, but –

Instructor (Julie Zelenski):That's cool.

Student:– it was actually really close to operating in [inaudible] space, even up to something like 10 million elements.

Instructor (Julie Zelenski):Oh, that's awesome.

Student: And it was more than twice as fast as the quick sort provided in the – [Crosstalk]

Instructor (Julie Zelenski):Excellent. Excellent. That's cool.

Okay. All right. So where did we leave off last time? I'm gonna reiterate these things that are quirky about the last changes I was making at the end, was taking an existing container class that held only strings and turning it into this template form of it that would now hold any kind of thing I wanted to stick into it. And along the way, I had to make a couple changes. And each of them actually kind of has a little – some pitfalls in getting this a little bit wrong, and so I just want reiterate them.

There's a lot of examples of these in the textbook. So you will see all of the templates that are implemented in the Chapters 9, 10, 11, and so on, do have examples of this. And for this first assignment, we won't be building it as a template, so this is actually kind of more like a future thing to kind of get prepared for when you're writing a template later.

But just a reminder about the things that have to happen is this template type name elem type, the template header we'll call that, gets placed sort of all over the place when you make that change. The first place it appears is in the .h on the class interface. You'll say template type name elem type class vector, class stack, class whatever it is you're building. That now means that the class is a template type whose name will only really exist in the form vector angle bracket string close angle bracket, from that point on.

In the .cpp, it gets repeated on every single member function definition. There's not one overarching kind of scope around all of those. Each of those has its own little independent setup for the scope, and then it has this additional sort of goo that needs to go on it about, oh, it's the template form of the member function size that is based on the vector class template. So you'll have to kind of get this all over the place.

And then the scope of the class, its new name is now whatever your original class name was but with the angle bracket elem type colon colon; there is no longer a vector unadorned once you've made that change. And so, when the client uses it, you'll always see that. And even on the scope resolution that's being used, even in the member functions themselves have this same adornment.

And then elem type gets used everywhere you have otherwise said it's storing strings. This is a string argument, this is a string return type, this is a local variable that's string, this is a data member that is a, you know, array of strings, whatever. All those places where you were saying – you were earlier committing to a precise type, you're gonna now use elem type everywhere.

It's pretty easy to do in like nine out of ten places you need to do it and leave one of them out. And the funny effect of that will be that sometimes it'll – you'll almost – it'll go unnoticed for a while because you – let's say you wrote it as a vector of strings originally. You change it to be a vector template, you left one of the places where there should've elem type still says string. But then you keep testing on vectors of string because you happen to have a lot of vector string code lying around that you were earlier using.

And what happens is you are instantiating the vector with string and it's kind of – the mistake is actually being hidden because the one place where it's wrong, it happens to be exactly string, which is what you're testing against. And it was only when you went out of way to then put vectors of int that you would suddenly get this type mismatch in the middle of the code, where it says, oh, here's this place where you were declaring a variable of string and you're trying to put an integer into it or something, or you're trying to have an array that's declared as integers and it really needs to hold – it's declared to hold strings and it really needs to hold ints.

And so you will – one way to shake that out early is actually to be sure that whatever testing code you're doing on one type, you also do it again on a different type, to make sure that you haven't accidentally got some subtle thing hiding in there.

And then this last thing is about how the templates are compiled, which is truthfully just a quirky behavior of the way C++ requires things to be done based on how the compiler's implemented, that the template is not compiled normally. So all normal files, you've got random.cpp and name.cpp and boggle.cpp, you always put those in the project, and that tells the IDE you want this one compiled. And it says, "Please compile this and link this in with all the code."

Once you have code that is templatized in here, you don't want it to be compiled in advance. It can't be compiled in advance, is really the problem. That looking at that code only describes the pattern from which to build vectors, not how to build one vector and compile it for all intents and purposes. It's on the fly gonna make new vector, vector, vector. So we pull it out of the project. We remove it, so we don't want the compiler trying to do something with it in advance.

For some compilers, actually, you can leave it there and it'll be ignored. The better rule to get used to though is just take it out because some compilers can't cope with it. You don't want to actually get into any bad habits. Pull it out; it no longer is compiled like an ordinary implementation file. You then change your header file to bring in that code, into the visible interface space here, at the very end by doing this #include of a .cpp file.

In no other situation would you ever want to #include a .cpp file. So as a habit, I'm almost giving you a little bit of a bad one here. Do not assume this means that in any other situation you do this. It is only in this one situation. I have a template class, this is the header for it, it needs to see the implementation body as part of the pattern to be able, to generate on the fly the client's particular types, and this is the way we're getting the code in there.

The way I would say most library implementations, sort of standard implementations in C++ do this, is they just don't bother with having a separation in the .h and the .cpp at all. They just put everything into the .h to begin with. And they don't even pretend that there is an interface separated from the implementation.

That's sort of sad because and sometimes that interface implementation split is an important psychologically of how you're thinking about the code and what you're doing. And I think jamming them together in the same file clouds that, that separation that we consider so important. But that is probably the most practical way to handle this because then it just doesn't – you don't have problems with thinking you can compile this and having to muck with this #include of cpp which is weird. So you'll see it done the other way in other situations, too.

All right, so that was just like I wanted to reiterate those because those are all little quirky things that are worth having somebody tell you, than having to find them out the hard way by trial and error. Any questions about kind of template goo? This is like the biggest class I've had in years. What do you guys – like is this midterm day? Was that why you guys are here or just because you heard I was gonna dress in a skirt and you wanted to see it? There's like three times as many people as we always have.

I'm gonna code. I'm gonna code, I'm gonna go to my new computer. You see my new fancy computer? It's like – I like it when we just code. I feel – like when I do it on the slides, I always feel like it's so dry, it's so boring, and it looks like it just you know came out of nowhere. It's good to kind of see it in real time and watch the mistakes be made live.

So what we're gonna do over here is we've got the myvector that we were working on last time. So it's got a skeletal interface. It has size add get set at, and some of the other functions are missing. The ones like clear and insert at and remove at, and the overloaded bracket operator and things like that are not there. Okay. But as it is, it does work.

In our simple test right here, we put nine, ten, one, and then we printed them back out. So let's just go ahead and run that to see that if I put a nine, ten, and a one and I iterate over what I got there using the get at and size, it seems like it has put ten – these three things in and can get them back out. Okay.

Now, I'm gonna do – I'm gonna show you something that's gonna make us a little bit sad and then we're gonna have to figure out how to fix it. That there is behavior in C++ for any class that you declare, that unless you state otherwise, that it's capable of assigning from one to another, making a copy. So all I added here at the bottom was another vector I called W, who – I should make this size a little bigger. This is screen is, I think, a little different than my other resolution, so let's crank that number up; make it a little easier to see what we're doing.

I declared a new myvector also of int type. It's name is W, and I said W=V. So this is actually true of all types that you declare in C++ that by default they have a default

version of what we call the assignment operator, that does kind of what's – you can imagine just a memberwise copy of V onto W.

So that works for structs, think about it in the simple case of a struct. If you have a struct with X and Y fields and you set this one to zero zero. And then you have point one has – set the X and Y to zero zero, and you say Point 2 = Point 1, it copies over the X and Y values. So you end up with two points who have the same field values, whatever they were, the same numbers. The same thing is true of classes, that unless you state otherwise, if you copy one instance, one object on top of another, it copies the values of the fields.

Okay, let's look at the fields to see how this is gonna work for us. The fields in myvector right now are a pointer to a dynamic array out here, two integers that are recording how many slots are used in that array and what its capacity is, and then there's no other data members, so we have three data members. So when I have made my V and my W, each of them has D space for one pointer here at the top and then these two integers.

And in the first one, if you remember a little about how the code works, it allocated it to some default size. I don't remember what it was, maybe it's, you know, ten or something. And then it fills it up from left to right.

So the first one I had, I'm gonna put in the numbers, you know, one, two, three, let's say. So I add one, I add a two, I had a three, then the number used will be three and let's say the number allocated is ten. So this is num allocated, this is num used. Okay. So subsequent things being added to that array will get added at the end. We know what our bounds are and all this sort of stuff. Okay.

Now I say W in my code, I say, "Oh yeah, just declare W." Well, W's constructor builds it a ten member array, sets the num allocated to ten, sets the num used to zero, and it's got kind of empty ready to go vector to put things into. When I say W=V, it's just gonna take these fields kinda one by one and overwrite the ones that were in W. Let's watch how that works out.

So we copy the num allocated over the num allocated, ten got written on ten, okay. We write the num used on top of the num used, okay. And then we copy the pointer on top of the pointer. This is pointer assignment, this is not doing any what we think of as a deep copy, it's a shallow copy or an alias. That means that W no longer points to here, its arr points to there.

All right. This can't be good, right? If you look at this picture, you've already got to be worried about where this is heading. You know just immediately you will notice that this thing: orphaned. Right? No one's holding onto this, this thing's hanging out in the heap, this guy's dead in the water. Okay, so we've lost some memory.

That, in itself, doesn't sound terrible, but now we have V and W both pointing to the same array. They have the same values for these two things but they're actually

independent. And now it is likely that if we were to continue on and start using V and W, we're gonna see some really strange effects of the two of them interfering with each other.

So for example, if I do a V.SetAt – well, let's do W for that matter, so I say W.SetAt at index zero, the number 100. So W says, you know, check and make sure that's in balance. It says, "Oh yeah, the index zero is within – it's greater than or equal to zero, it's less than my size. Okay." And it says, "Go in and write 100 in there."

As a result, it turns out V.GetAt also changed. And that would be pretty surprising that these happen. So now, they're just colliding. They have – there's one piece of memory that's being shared between the two of them, and when I change one of them, I'm changing both of them. They are not independent. They now have a relationship based on this assignment that's gonna kinda track together.

There are actually worse consequences of that than that. So this looks kind of innocent so far, not – "innocent" isn't the word but at least it's the opportunity for malicious error still seems like, well okay, you have these values that are changing. The really terrible situation would happen when let's say V keeps growing, they keep adding more things.

So they add the numbers four, five, six, seven, eight, nine, and then it fills up. So it has num used as ten, num allocated as ten, and it goes through and it says, "Oh, it's time to grow. I want to add an 11." And the process of growing, if you remember, was to build an array that was twice as long, copy over the front values, and so copy up to here, have this back half-uninitialized and then deallocate the other one. So delete this piece of memory, and then update your pointer to point to this new one, that now is allocated to a length of 20.

Okay, when you did that, W just got screwed. W points to this piece of memory that you just took away. And then you will be much more sad than just getting a junk value out or a surprisingly changed value out. Now if you ask W to go get the value at position zero, all bets are off. Right? It might happen to work, it probably will work for a little while, but at some point this memory will get reclaimed and reused and be in process for something else, and you'll just have completely very random looking undetermined results from access to that W.

So this really just can't exist. We do not want it to be the case that this default memberwise assignment goes through. It will do us no good. So in the case for objects where memberwise copying is not what you want, you have to go out of your way to do something about it.

The two main strategies for this is to really implement a version of assignment operator that will do a keep copy. That's actually what our ADTs do. So the vector and the map and stack and queue are setup to where if you say V=W, it makes a full copy. And that full copy means go take the piece of memory, get another new piece of memory that big, copy over all the contents.

And so then, you end up with two things that have the same number of elements in the same order, but in new places in memory. They're actually fully duplicated from here to there. And at the point, V and W don't have any relationship that is gonna be surprising or quirky in the future. They just happen to get cloned and then move from there.

That's the way kind of a professional sort of style library would do it. What I'm gonna show you instead, because I don't really want you to learn all the goopy things about how to make that work, is I'm gonna show you how to just disallow that copying. To make it to where that it's an error for you to attempt to copy V onto W or W onto V, by basically taking the assignment operator and making it inaccessible, making it private.

So I'm gonna show you how I'm gonna do that. Well actually, first I'll just show that like the truth about these kind of errors. And these errors can be really frustrating and hard to track down, so you just don't want to actually mess with this. If I have W=V here, and then I say W.SetAt zero is 100. So I was supposed to put in the numbers – let me go – numbers I know where to look for.

I'm gonna put in one, two, and three. I change it in W and then I write another for loop here that should print out the values again. This is where I'm gonna see. So like 1, 2, 3 now have 100, 2, 3. And then actually I'm even seeing an error here at the end where it's saying that there was a double free at the end. The free is kind of the internal name for the deallocation function.

And in this case, what happened is that the destructor for V and W were both being called. As I exited scope, they both tried to delete their pointer. So one of them deleted it and then the second one went to delete it, and the libraries were complaining and said, "You've already deleted this piece of memory. You can't delete it twice. It doesn't make sense. You must have some error." So in fact, we're even getting a little bit of helpful runtime information from what happened that might help us point out to what we need to do.

So let me go and make it stop compiling. So there is a header file in our set of libraries that's called disallow copy. And the only thing that is in disallow copy is this one thing that's called a macro. It's kind of unfortunate that it has to be done that way. Well, I can't find the header file, so I'll just tell you what it is. And it looks like this. And I say disallow [inaudible] copying, in all capital letters, and then I put in parentheses the name of the class that I'm attempting to disallow copying on.

You can have a semicolon at the end of this or not, it doesn't matter actually. I'll put one just because maybe it looks like more normal to see it that way. And by doing this in conjunction with that header file it's bringing in the macro that says put into the private – so we always put this in the private section. So we go to the private section of our class, we put this disallow copying macro in here. And what this will expand to is the right mojo.

There's sort of a little bit a magic incantation that needs to be generated, and this is gonna do it for you. That will make it such that the – any attempt to clone a vector using that memberwise copy, and so that would happen both in direct assignment from V to W, or in the cases where copies are made, like in passing by value or returning by value, those actually are copy situations as well, that it will make all of those things illegal. It will take the standard default behaviors for that, and make them private and inaccessible and throw errors basically, so that you just can't use them.

And if I go back to – have made this change, I go back to the code that was trying to do this, and I'll take out the rest of the code just so we don't have to look at it, that it's gonna give me this error. And it's gonna say that in this context, the errors over here, that the const myvector, and there's just a bunch of goo there. But it's saying that the operator equals, is the key part of that, is private in this context.

And so, it's telling you that there is no assignment operator that allows one myvector to be assigned to another myvector. And so any attempt by the client to make one of those copies, either from passing, returning, assigning, will just balk right then and there and not let the kind of bad thing that then will just have all sorts of other following errors that we would not want to have to debug by hand.

So this will become your mantra for any class that has memberwise variables to where copying them in a naive way would produce unintended effects. So if all the variables in here were all integers or strings or, for that matter, vectors or other things that you know have true deep copying behavior, then you don't need to go out of your way to disallow copying.

As long as each of the members that's here, that if you were to do an assignment from one variable of this type to the other, it would have the right effect, then you don't need to disallow. But as soon as you have one or more variables where making a simple assignment of it to another variable of that type would create some sort of long-term problem between those two objects, you want to disallow that copying and not let that bad thing happen. So things that have a link list in them, things that have pointers in them, dynamic arrays in them, will all need this protection to avoid getting into that situation.

Question?

Student: Could you somehow overload the assignment [inaudible]?

Instructor (Julie Zelenski):Well, you certainly can so that's kind of the first strategy I said, which is like, well, you can make them deep copy is the alternative. So one way is to say, "Well the shallow copy is wrong. What are you gonna do about it?" So I'm saying don't let shallow copy happen. The other alternative is to replace shallow copy with a real deep copy. That's what ours do.

If you're interested to look at that, you can look at our code and see what we do. It just goes through – you can imagine what it would take. It's like, okay, get some information from here, make a new array, copy the things over, and then at that point you will be able to continue on with two things that are cloned from each other, but no longer have any relationship that would cause problems.

It's not that it's that hard but the syntax for it is a little bit goopy, and ours in C++ we're not gonna see. So you can look at it; if you want to as Keith at 106L, he will tell you everything you want to know.

Any questions about – over here?

Student: So the things inside of this copy, can be anything? It can be one of the variables you declared?

Instructor (Julie Zelenski):So typically it'll be the name of a class.

Student: But could be like it'd just have a variable inside your class, you don't want people really to copy [inaudible]?

Instructor (Julie Zelenski):Well it doesn't really work that way. The disallow copying is saying I'm taking this type and making it not able to be copied, and so it is an operation that applies to a type, not to a variable. And so, in this case, the name here is the name of the class who we are restricting assignment between things of myvector type. And so, it really is a type name not a variable name.

If I said disallow copying of like arr or num used, it actually won't make sense. It'll expand to something that the compiler won't even accept. It says, "I don't know what you're talking about." It expects a type name in that capacity, so what thing cannot be copied. It is things that are of myvector type.

So there's not a way to say you can't copy this field by itself or something like that. It's really it's all or nothing. You can copy one of these objects or you can not copy it. And once you have some field that won't copy correctly without help, you're gonna wanna probably disallow the copying to avoid getting into trouble.

All right, so you got vector; vector's a good thing. I'm gonna add one more member function of vector before I go away, which is insert at. I'll call it E. The insert at one is the – allows you to place something in an index in the – which one I just copy? There I go. Insert at index in an element type. So if the index is before the beginning or off the end, I'll raise the same out of bounds error, so that's I just picked up that piece of code from there.

I also need to have this little line, which is if the number of elements used is equal to the capacity, then we need to make more space. So it – just like the case where we're adding to the end, if we're already at capacity, no matter where we're inserting, we have to make

some more space. So we go ahead and do that, the initial step of checking to see if we're out of capacity and enlarging in place.

Once we've got – we're sure we have at least enough room, we're gonna move everybody over. So I'm gonna run a for loop that goes from size minus one to – whoops, I need an I on that. Is greater than the index – actually, I want to do greater than or equal to index. And I'm gonna move everything from index over in my array, and then array of index equals E.

Let me think about this and make sure I got the right bounds on this, right? This is taking the last element in the array, it's at position size minus one; and then it's moving it over by one, so it's reading from the left side and moving it over by one. And it should do that all the way down until the last iteration should be that I is exactly equal to index. And so, it should move the thing out of the index slot to the index plus one slot, so along the way moving everybody else there.

Why did I run that loop backwards?

Student:[Inaudible].

Instructor (Julie Zelenski):Yeah, it overran the other way. Right? If I try to do it the other way, I try to copy, you know I have the numbers four, five, and six in here and I'm planning on inserting at zero, I can't start from the beginning and four on top of the five and then on top of that, without making kind of a mess of things. So it's actually – I can make that work but it's definitely sort of messier. It's sort of easier to think about it and say, "Oh, well just move the six over, then move the five over, then move the four over, and then write your new element in the front." And before I'm done, I do that.

And so, I have insert at; and I could probably test it to make sure that it does what it's supposed to do: V.InsertAt position zero, put a four in the front, and then move this loop down here. So I have one, two, three, and then I put a four in the front. Oh, look at that, something didn't work. Want to take a look at that? Four, one, two, what happened to my three? Where did my three go? Why did I lose my three?

How does it know how many elements are in the vector? It's keeping track of that with num used. If you look down here, for example, at add, when it sticks it in the last slot, it updates the num used and increments it by one. I wasn't doing that over here at insert, right? And a result, like it didn't – you know it's admirable job. It moved all the numbers over, but then it never incremented its internal counter, so it thinks actually there's just exactly still three elements in there. So I'd better put in my num used ++ in there too. Ah, four, one, two, three. So my number was there, I just wasn't looking at it. It was just a little further in there. Okay.

All right, so with that piece of code in, I think we're in a position to kinda judge the entire strategy of using a dynamic array to back the vector. The remove at operation is similar, in terms of needing to do that shuffle; it just does it in the other direction. And

then we have seen the operations, like set at and add and get at, and how they're able to directly index into the vector and kind of overwrite some contents, return some contents. And so you kinda get a feel for what it is that an array is good at, what things it does well, and what things it does poorly.

So a vector is an abstraction. You offer up to somebody a vector it's a list of things. That the tradeoffs that this implementation of vector is making, is it's assuming that what you most care about is that direct access to anything in the vector because that's what arrays are good at. It is a trivial operation to say start at the beginning and access the Nth member, because it just does a little bit of math. It takes the starting address in memory and it says, "Oh, where's the tenth member? It's ten positions over in memory." And so, it can do that calculation in no time, and then just directly access that piece of memory.

So the – if what you are really concerned about is this ability to retrieve things or update things in that vector, no matter where you're talking about, the front, the back, the middle, in constant or one time, this design is very good for that. What are the operations it's gonna actually bog down on? What is it bad at? When do you see sort of the consequences, let's say, of it being in contiguous memory, being a disadvantage rather than advantage?

Student: Adding something at the [inaudible].

Instructor (Julie Zelenski):Adding something where?

Student: At the beginning.

Instructor (Julie Zelenski):At the beginning. Certainly any operation like this one, the insert at or the remove at, that's operating in the front of the array is doing a big shuffle, things forward, things back, to make that space, to close down that gap. And so, for an array of large enough size, you'd start to notice that.

You have an array of 1,000, 10,000, 100,000, you start inserting at the beginning, you're gonna see this cost of the insert shuffling everything down, taking time, relative, in this case, linear, in the number of elements that are being moved. Which on average, you figure you're inserting kind of randomly in positions, it could be half of the elements or more are gonna need to move.

It also actually pays a little cost in the resizing operation. That happens more infrequently. The idea that as you get to capacity, you're growing, in this case we're doubling, so we're only seeing that kind of at infrequent intervals, especially as that size gets large. Once you get to 1,000, it'll grow once to be 2,000. Well then, it'll grow once and be 4,000. So you'll have a lot of inserts before you'll see that subsequent resize operation.

But every now and then, there'll be a little bit a glitch in the resizing where you would say, "I'm adding, adding –." If you're adding at the end, adding at the end is easy, it

increments the num used and sticks it at the end. But once every blue moon, it'll be a long time as it copies. And then you'll be go back to being fast, fast, fast, fast, fast, till you exhaust that capacity. And then again, every now and then, a little bit a hiccup when it takes the time to do the resizing.

Overall, the number of inserts, that cost can kind of be averaged out to be considered small enough that you'd say, "Well, amortized over all 1,000 inserts it took before I had to copy the 1,000 when I grew to 2,000, each of them paid 1/1000th of that cost, which ended up making it come out to, on average, still a constant cost," is one way that we often look at those analyses.

So this tells you that like there's certain things that the array is very good at. It has very little additional memory overhead. Other than the additional space we're kind of storing in the capacity when we enlarge, there really isn't any per element storage that's used in addition to the number or the string or whatever it is we're storing itself. So it has very low housekeeping associated with it. And it does some things very well but other things a little less efficiently because of having to stick it in memory meant we had to kind of do this shuffling and rearranging.

So the other main alternative you could use for a vector is a link list. I'm actually not gonna go through the implementation of it because I'm actually gonna do stack as a link list instead. But it is interesting to think about, if you just sort of imagine. If I instead backed vector with a link list, first of all could I implement all the same operations? Like is it possible to use a link list, can it do the job if I'm determined?

Like what will it take, for example, to get the element at the nth position from a link list design? How do you know where the nth element of a linked list is? It's [inaudible] list, you know. Help me out. [Inaudible].

Student: You have to actually dereference the pointers N times.

Instructor (Julie Zelenski):That's exactly what you got to do; you got to walk, right? You got to start the beginning and say, "Okay you're zero. And after you is one, and after that's two." And so you're gonna walk, you're gonna do N arrow next, to work your way down. You will start at the beginning and – and we're gonna learn some new vocabulary while we're here. And so that means that accessing, for example, at the end of the list is gonna be an expensive proposition.

In fact, every operation that accesses anything past the beginning is doing a walk all the way down. And so if you planned on, for example, just printing the list or averaging the list or searching the list, each of those things is going, "Give me the zero, give me the next, give me the third." You know it's gonna just keep walking from the beginning each time. Like it's gonna turn what could've been a linear operation into N squared because of all those start at the beginning and walk your back down.

So for most common uses of a vector, that would be so devastating that you just wouldn't even really take it seriously. That every time you went to get something out of it or set something into it, you had to make this enormous traversal from the front to the back would just be debilitating, in terms of performance implications.

The thing that the link list is supposed to be good at is that manipulation of the memory. That the operation, like insert at or remove at, that's doing the shuffle, the link list doesn't have to do. So for example, the worst place you can insert at in a vector is zero, if it's implemented using an array because you have to shuffle everybody over. The best place you can insert at in a link list is actually at zero because then you just tack a new cell on the front.

And then sort of further down the list, right? It's not the act of splicing the new cell in that's expensive; it was finding the position at which you needed to do the splice. So if you want to insert at halfway down the list, you have to walk down the list, and then you can do a very quick splice but you had to get there.

So it makes this – it's kind of an inverted set of tradeoffs, relative to the array form, but adds a bunch memory overhead, adds a bunch of pointers, adds a bunch of allocation and deallocation. So there's a bunch of code complexity, and in the end you don't really get anywhere I think you'd want to be. So it's very unusual to imagine that someone would implement something like the vector using a link list strategy, although you could. So.

Let's talk about stack, instead. I have a little stack template that I started: mystack. So I push in pop and size on. Okay. I'm lazy, so I'm going to do the easiest thing in terms of implementing stack: that is to layer.

Okay. So once you've built a building block as an implementer, there's nothing that stops you from then turning around in your next job and being a client of that building block, when implementing the next thing you have to do. It may very well be that the piece that you just built is a great help to writing the next thing, and you can build what's called a layered abstraction. I'm ready to build stack and I've already gone to all this trouble to build vector, it's like, well. It turns out one of the ways I could implement a stack is to use something like a vector or an array.

Now I could build it on a raw array, but I happen to build something that kind of has the behaviors of a raw array: the tradeoffs of a raw array, the Big-O of raw array, and it actually just manages convenience, it has error checking and stuff in it. It's like why not just use that? I'll pay a little bit of cost in terms of taking its level of indirection onto things, but it's actually gonna be worth it for saving me sort of the grungier aspects of the code.

So if I'm gonna put stack contents into an array, and if I were going to push A then B then C, right? So then, the stack that I want to have would look like this. There's the top of the stack up here, where C is the last thing on. The bottom of the stack is down here; with the first thing I pushed on, which is A. I could put this in an array. Seems like the

two obvious ways to do this would be, well, to put them in this way: A, B, C, or to put them in this way. Okay. So this would be the top of the stack is over here, the bottom of the stack is over there, and then down here it's inverted. This is the top of the stack; this is the bottom of the stack.

Okay, they seem symmetric, you know at the very least, but one of those is a lot better for performance reasons. Which one is it? Do you want to go with Strategy 1 or Strategy 2?

Student: One.

Instructor (Julie Zelenski):One? Why do I want one?

Student: What?

Instructor (Julie Zelenski):Why?

Student: Because you don't have to move all the –

Instructor (Julie Zelenski):Exactly. So if I'm ready to put my next element in, I'm ready to put D in, that in the Strategy 1 here, D gets added to the end of the vector. Right? Adding to the end of vector: easy, doesn't require any shuffling, updates the number. Sometimes it has to grow but easiest thing to do, add to the end. In this version, adding to the top would be moving everybody over, so that I have to update this to D, C, B, A, and slide everybody down. So I had to do insert and a shuffle: bad news, right?

Similarly, when I'm ready to pop something, I want to get the topmost element and I want to remove it that the topmost element here being at the end, remove at is also fast at the very end. Right? Taking the last element off, there's no shuffling required; I just [inaudible] my num used. If I need to do it from here, I have to shuffle everybody down. So it seems that they were not totally symmetric, right? There was a reason, right? And I had to think it out and kind of do it and say, "Oh yeah. Okay, put them at the end."

So if I do that and I go over here, I'm gonna finish doing the things that make it a proper template. I just want to look at mystack, ooh, and I have the things here, is that I don't have anything to do with my constructor or destructor. I'm just gonna let the automatic construction and destruction of my members work, which is to say give me a vector, delete my vector. All that happens without me doing anything.

Then I want to know my size. I don't know my size but the vector does for me, so I tell the vector to do it for me. When I want to push something, I tell my vector to add it for me. I'm telling you, this is like a piece of cake. And so then I say elem type top equals elems.GetAt. I can actually use the – I'm back to using the real vector, so I can use anything that has at the last position. Elems.RemoveAt, elems.size [inaudible], then I return it. Almost but not quite what I want to do, but I'll leave that there for now.

And then that's defining the function I wanted, right? So they're all just leveraging what the vector already does for me, in terms of doing the remove. I guess RemoveAt actually, the name of that member function. The add that does the growing and the shifting and all this other stuff happened. Oh good, I have a stack and I didn't have to do any work. I love that. So let's see if it works.

Mystack.s, push. Pop one thing off of it just for – even though I actually got what I was supposed to get. Oh, I'd better include the right header file; I'm gonna do that. Doesn't like size; let's see what I did with size that I got wrong. Okay, should be size with arguments, there we go. What did I put? One, two, three, and then I popped and I got a three. Hey, not bad, not bad for five minutes of work.

There's something about this pop method though that I do want to get back. So push actually is totally fine, that's just delegating the authority to kind of stash the thing in the vector. Pop right now, what's gonna happen if there isn't anything in the stack when you pop? Something good? Something bad? Something helpful? Wanna find out? Why don't we find out? Never hurts to just try it.

So like right now, it seems like it just you know digs into the elem. So if the elem size is zero, there are no elements in the stack, it'll say, "Oh, give me the elements of negative one," and then it'll try to remove that negative one. I got to figure those things aren't good things. I'm hoping that we'll push one thing and then we'll pop two things. See how it goes.

Hey, look at that. [Inaudible] access index negative one and the vector size zero, so that was good. It was good that we got an error from it. We didn't actually like just bludgeon our way through some piece of memory that we didn't know it or anything crazy like that. But the error that we got probably wasn't the most helpful. That upon seeing that error, it might cause you to go – kind of get a little bit misled. You might start looking around for where am I using a vector? Where am I making a call to vectors bracket?

Like I look at my client code and all I see is use of stack. The fact that stack is implemented on vector, is something I'm not supposed to know or not even supposed to be worrying about. And so having it poke its head up there, might be a little less clear than if, instead, it had its own error message.

And so I can actually just go in here to mystack and say, "Yeah, you know, if write my size is zero, error, pop empty stack." That you know it doesn't really change the behavior in any meaningful way. It's like it still gets an error, but it gets an error in this case that's more likely to tell somebody where the trouble is and where to look for it: so start looking for pop on a stack, instead of expecting to go look for a vector access. So just being a little bit more bulletproof, a little bit more friendly to the client by doing that, that's what.

And so all the operations on this stack are O of one right now, the add, the pop are – push and pop are both adding to the end of the array, which is easy to do. And so the – this is a pretty efficient implementation and pretty easy to do because we're leveraging vector.

What we're gonna look at next time is what can a link list do for us or not, does it provide anything that might be interesting to look at. What I'm gonna talk about instead is I'm gonna give you the two-minute summary of what the midterms looked like. And then we will give them back to you, which is, I'm sure, what you just wanted on your nice Monday morning.

So the histogram, can I have one of the solutions here? So you've got your really just rock solid normal distribution but with a kind of very heavy weighting sort of toward the end, right? The median was I think in the mid-70s: 74, 75; the mean was a little bit lower than that, and so a very strong performance overall. I'd say that probably the most difficult on the exam was clearly the recursion question, whereas I'd say the average on that was about eight or nine points down from perfect, so.

But there was actually – it's kind of a very interesting clumping. A bunch of people who were totally perfect, a big bunch that kinda got something right in the middle, and then some smaller groups on the other end. So they're very kind of separated into three groups.

The solution has some information, has the grading stuff, stuff like that, kind of gives the thing. The most important thing though is if you are at all concerned about how to interpret your grade, so if you feel like you want some more information, the check, check-plus stuff on the assignments kind of confuses you, you just want to have an idea, the right person to talk to is me. So you can certainly come to my office or send me an e-mail, if you just kind of where you're at and how to make sure you're at the place you want to be going forward, I'm happy to talk that through with you.

So in general, I thought the exam was very, very good. It was a little bit long, I think. Even though people did manage to answer all the questions, I think had it been a little shorter we might've have gotten a little bit higher scores overall. But in general, I thought the scores kind of right in the target range we wanted to have, so I'm very pleased.

If you would like to come up, I'm gonna have to figure out how to do this in a way that does not actually make a huge – I think what I will do is I'm gonna do the alphabet. I have them alphabetized by last name, and I'm gonna start by putting kind of As, you know A through C in a little pile, and so on, and sort of across the room. So depending on where your last name is, you might want to aim yourself toward the right part of the room. Okay?

So I've got a little stack of like As and Bs, let's say; this is kinda Cs and Ds and something else.

[End of Audio]

Duration: 51 minutes

ProgrammingAbstractions-Lecture21

Instructor (Julie Zelenski):Hey there! Oh, we're back. We're back and now we're getting down and dirty. We'll do more pointers, and link list, and implementations. It's actually going to be like our life for the next week and half. There's just even more implementation, trying to think about how those things work on the backside. So I will do another alternate implementation of the stack today. So we did one based on vector and we're going to do one based on a link list. And then I'm going to do a cube based on link list, as well, to kind of just mix and match a little bit. And then, we're still at the end of the case study, which the material is covered in Chapter 9. I probably won't get thorough all of that today. What we don't finish today we'll be talking about on Friday, and then we'll go on to start talking about map implementation and look at binary search trees. How many people started [inaudible]? How many of you got somewhere feeling good? Not just yet. One thing I will tell you is that there are two implementations you're working on and I listed them in the order of the one that the material we've seen the most, right. We've talked about link list more than we have talked about things that are tree and heat based. So I listed that one first. But, in fact, actually, in terms of difficulty of coding, I think, the second one is little bit more attractable than the first.

So there's no reason you can't do them in either order if it feels good to you to start on the second one before you come back to the first one. Or just trade off between working on them if one of them is giving you fits just look at the other one to clear your head, so just a thought about how to approach it. Okay. So I am going to do some coding. I'm going to come back to my slides in a little bit here. But to pick up where we left off last time, was we had written a vector based implementation of the stack attraction and so looking at the main operations here for a stack or push and pop array that its mechanism here is just to add to the end of the vector that we've got. So I'm letting the vector handle all the growing, and resizing, and memory, and manipulations that are needed for that. And then, when asked to pop one, we just take the one that was last most added, which will be at the far end of the array. And so given the way vector behaves, knowing that we know it's a continuous array based backing behind it, then we're getting this O of one access to that last element to either add a new one down there or to retrieve one and remove it from the end so that push and pop operations will both being in constant time no matter how many elements are in the stack, 10, 15, 6 million, the little bit of work we're doing is affecting only the far end of the array and not causing the rest of the configured storage to be jostled whatsoever so pretty easy to get up and running.

So in general, once you have one abstraction you can start leveraging it into building other attractions and the vector manages a lot of the things that any array based implementation would need. So rather than using a raw array, there's very little reason to kind of, you know, ditch vector and go the straight raw array in this case because all it basically does it open up opportunities for error and kind of management and the kind of efficiency you can gain back by removing vectors intermediate stuff is really not very profound so not worth, probably, taking on. What we do is consider the only link limitation. We had talked about a link list for a vector and we kind of discarded it as likely not to lead anywhere interesting. I actually, am going to fully pursue it for this

stack because it actually turns out that a link list is a good fit for what the behavior that a stack needs in terms of the manipulations and flexibility. So if I have a stack of enterers where I push 10, 20, and 30. I'm going to do that same diagram we did for the vector one as I did for the stack is one possibility is that I put them in the list in this order. Another possibility is that I put them in, in this order. We'll call this Strategy A; we'll call this Strategy B. They seem, you know, fairly symmetric, right. If they're going to be a strong reason that leaves us to prefer Strategy A over B, or are they just equally easy to get the things done that we need to do? Anybody want to make an argument for me. Help me out.

Student: Last in, first out.

Instructor (Julie Zelenski):Last in, first out.

Student:[Inaudible].

Instructor (Julie Zelenski):Be careful here. So 10, 20, 30 means the way the stack works is like this, 10, 20, 30 and so it's at the top that we're interested in, right?

Student: Um hm.

Instructor (Julie Zelenski):That the top is where activity is taking place.

Student: Oh.

Instructor (Julie Zelenski):So we want to be adding things and losing from the top. So do we want to put the top at the end of our list or at the front of our list? Which is the easier to access in the link list design?

Student:[Inaudible].

Instructor (Julie Zelenski):The front, right? So in the array format we prefer this, putting the top of the vector on the far end because then there is the only jostling of that. But needing access to the top here in the front is actually going to be the better way to get the link list strategy you work in.

Let me actually go through both the push and the pop to convince you why this is true. So if I have access in the 10, 20, 30 case I could actually keep a separate additional pointer, one to the front and one to the back. So it could actually become maintaining at all times as pointing to the back. If I did that, right, then subsequent push up to a 40, it's pretty easy to tack on a 40. So it turns out it would be that the adding it's the push operation that actually isn't really a problem on this Strategy A. We can still kind of easily, and over time, if we have that tail pointer just tack one on to the end.

As for the pop operation, if I have to pop that 30, I need to adjust that tail pointer and move it back a cell. And that's where we start to get into trouble. If I have a pointer to the

tail cell there is no mechanism in the singly linked list to back up. If I say it's time to pop 30, then I would actually need to delete the 30, get the value out, but then I need to update this tail pointer to point to 20, and 30 doesn't know where 20 is, 30's oblivious about these things. And so the only way to find 20 in singly linked list would be go back to the very beginning, walk your way down, and find the one that pointed to the cell you just took out of the list and that's going to be an operation we want to avoid. In the Strategy B case, right, adding a cell, this means putting a 40 in allocating a new cell and then attaching it, splicing it in, right between wiring in two pointers, and we can do these links in constant time, and then popping one is taking that front row cell off. Easy to get to, easy to update, and splice out so no traversal, no extra pointers needed. The idea that there are, you know, 100 or 1,000 elements following the one that's on top is irrelevant. It doesn't have any impact, whatsoever, on the running time to access the front, or stuff another front, or taking it on or off the front. So it is Strategy B that gives us the best setup when it's in the back of the linked list.

So let me go ahead and do it. I think it's kind of good to see. Oh, like, what is it like to just write code and make stuff work. And so I'm kind of fond of this and you can tell me. I make a cell C it has an L type value and it has a soft T next right there. And then I have a pointer to the front linked list cell. So in addition, you know, it might that in order to make something like size operate efficiently, I might also want to catch the number of cells that update, or added or renew. I can also decide to just leave it out for now. Maybe I'll actually even change this to be the easier function, right, which is empty form. That way I don't even have to go down that road for now. Go back over here to this side and I've got to make everything now consistent with what's going on there. Okay. Let's start off with our constructor, which is a good place to make sure you get into a valid state. In this case, we're going to use the empty list so the head pointer points and tells us we have no cells whatsoever in the list. So we don't pre allocate anything, we don't get ready. What's kind of nice with this linked list is to allocate on demand each individual cell that's asked to be added to the list. The delete operation actually does show we need to do some work. I'm actually not going to implement it but I'm going to mention here that I would need to delete the entire list, which would involve, either iterates or recursion my way down to kind of do all the work. And then I changed this from size to iterate. I'd like to know if my list is empty. So I can just test for head is equal, equal to null. If it's null we have no list.

Let me work on push before I work on pop. I'll turn them around. The push operation is, make myself a new cell. All right, let's go through the steps for that. New cell equals new cell T. [Inaudible] the size to hold a value and an X link. I'm going to set the new styles val to be the perimeter that came in. And now, I'm going to splice this guy onto the front of my list. So the outgoing pointer I'm doing first to the new cell is allocated and leads to what was previously the front row cell. And then it is now updated to point to this one. We've got pointer wired, one value assigned, and we've got a new cell on the front. The inverse of that, taking something off my list, I'm going to need to kind of detach that front row cell, delete its memory, get its value, things like that. So the error checking is still the same at the very beginning. Like, if I've got an empty staff I want to be sure I don't just kind of do reference no and get into some bad situations. So I report that error

back using error, which will halt the program there. Otherwise, right, the element on top is the one that's in the value field of the head cell. So knowing that head's null is a safety reference to do there. And then I'm going to go ahead and get a hold of what that thing is and I'm going update the head to point to the cell after it. So splicing it out and then deleting the old cell. Just take a look at these. Once we start doing pointers, right, there's just opportunities for error. I feel kind of good about what I did here but I am going to just do a little bit of tracing to kind of confirm for myself that the most likely situations that get you in trouble with link list is you'll handle the mainstream case but somehow forget one of the edge cases. Such as, what if the list was totally empty or just had a single cell? Is there some special case handling that needs to be dealt with? So if I think about it in terms of the push case, you know, I might say, well, if there's an existing set of cells, so we're using Strategy B here, let me erase A so I know what I'm looking at, that its head is currently pointing to a set of cells, right, that my allocation of the new cell out here, right, assigning its value, assigning its next field to be where head points two, and then sending head to this new guy, it looks to be good. If I also do that tracing on the – what if the list that we were looking at was just null to begin with. So there were no cells in there. This is the very first cell. It isn't going to have any kind of trouble stumbling over this case. So I would allocate that new cell 40, assign its value. Set the next field to be what head is, so that just basically sets the trail coming out of 40 to be null and then updates the head there. So we've produced a single linked list, right, with one cell. It looks like we're doing okay on the push behaviors.

Now, the pop behavior, in this sort of case may be relevant to think about, if it's totally empty. All right, we come in and we've got an empty cell, then the empty test should cause it to error and get down to the rest of the code. Let's say that B points to some sequence of things, 30, 20, 10, that taking the head value off and kind of writing it down somewhere, saying, okay, 30 was the top value. The old cell is currently the head so we're keeping a temporary on this thing. And then head equals head error next, which splices out the cell around so we no longer have a pointer into this 30. The only place we can get back to it is with this temporary variable we assigned right here of old and then we delete that old to reclaim that storage and then our list now is the values 20 and 10, which followed it later in the list. To, also, do a little check what if it was the very last cell in the list that we're trying to pop? Does that have any special handling that we have overlooked if we just have a ten here with a null after it? I'll go through the process of writing down the ten. Assigning old to where head is, and assigning head-to-head error next, where head error of next cell is null but pointing to that, and then we set our list to null, and then we delete this cell. And so after we're done, right, we have the empty list again, the head pointer back to null. So it seems like we're doing pretty good job. Have a bunch of cells. Have one cell. No cells. Different things kind of seem to be going through here doing the right thing. Let's do it. A little bit of code on this side.

Student:[Inaudible].

Instructor (Julie Zelenski):Okay. Pop empty cell. I've got that. Oh, I think I may have proved, oh, it deliberately makes the error, in fact. That it was designed to test on that.

[Inaudible] up to the end. So let's see if we get back a three, two, one out of this guy. Okay. So we manage to do okay even on that.

So looking at this piece of code, all right, the two operations we're most interested in, push and pop, right, are each oh of one, right. The number of elements out there, right, aren't changing anything about its performance. It does a little bit of imagination and a little bit of pointer arrangement up here to the front to the end and so it has a very nice tight allocation strategy, too, which is appealing relative to what the vector-based strategy was doing. It's like, now, it's doing things in advance on our behalf like extending our capacity so that every now and then we had some little glitch right when you were doing that big resize operation that happened infrequently, you won't have that same concern with this one because it does exactly what it needs at any given time, which is allocate a single cell, deletes a single cell, and also keeping the allocation very tidy in that way. So they'll both end of being oh of one. And both of these are commonly used viable strategies for how a stack is implemented. Right. Depending on the, you know, just the inclination of the program or they may have decided one way was the way to go versus another that you will see both used in common practice because there really isn't one of them that's obviously better or obviously worse, right. Now, this uses a little bit more memory per cell, but then you have excess capacity and vector has excess capacity but no overhead per cell. There's a few other things to kind of think about trading off. You say, well, the code, itself, is a little harder to write if you're using only [inaudible]. That means it's a little bit more error prone and more likely, you'll make a mistake and have to do debug your way through it and the way with the vector it was pretty easy to get without tearing out your hair.

Now, if we can do stack and queue that fast – I mean, stack that fast, then you figure queue must be not so crazy either. Let me draw you a picture of queue. Okay. I've got my queue and I can use the End queue operation on it to put in a 10, to put in a 20, to put in a 30. And I'm going to try and get back out what was front most. Okay. So queue is FIFO, first in first out, and waiting in line, whoever's been in the queue longest is the first one to come out, very fair handling of things. And so if were to think about this in terms of let's go straight across the linked list. We just got our head around the link list with the stack. Let's see if there's a link list for the queue provide the same kind of easy access to the things we need to do the work. So it seems like the two strategies that we looked at for stack are probably the same two to look at for this one, right, which we go in the front ways orders. I mean the front of the queue accessible in there and reversing our way down to the tail or the other way around. Right. I mean, the tail of the queue up here in the front leading the way to the head of the queue. So this is head – I'll call it front of queue, and this is back, this is the back and that's the front. Okay. So that will be GA that will be GB. But first off, ask yourself, where does the queue do its work? Does it do it at the front or does it do it at the back? It does it at both. But when you're end queuing, right, you're manipulating the back of the queue, adding something to the tail end of the line. When you are de queuing, you're adding something to the front of the queue.

So like unlike the stack where both the actions were happening in the same place, and that kind of unified your thinking about where to allow for that, you know, easy access to

the top, the one thing the stack cared about, but the bottom of the stack, right, was, you know, buried and it didn't matter. The queue actually needs accesses to both. So that sort of sounds like that both these strategies, right, have the potential to be trouble for us. Because it's like in the link list form, right, you know the idea that access to the head is easy, access to the tail that is a little bit more tricky. Now, I said we could add a tail pointer. And maybe that's actually going to be part of the answer to this, right, but as it is, with no head pointers in here, that adding an item to A would involve traversing that queue to find the end to put a new one on. Similarly, if indeed, the inverse operation is going to be our trouble, which is de queuing, we have to kind of walk its way down to find the last element in the link list ordering, which is [inaudible] queue. So what say we add a tail pointer in both cases? So we have a head that points to here and a tail that points to there, it's like that will make our problem a little bit easier to deal with. So let me look at B first. That adding to the back of the queue is easy. We saw how we did that with the stack, right, so that the End queue operation doesn't need that tail pointer and it already kind of wasn't a problem for us. Now, the D queue operation would mean, okay, using our tail pointer to know where the last close value is tells us it's ten.

But we have the same problem I had pointed out with the stack, which is, we can get to this value, we can return this value, you know, we can delete this cell, but what we need to do, also, in the operation, update our tail pointer so a subsequent D queue can do its work efficiently. And it's that backing up of the tail pointer that is the sticky point that given that ten doesn't know who points into it, the single link list if very asymmetric that way. But you only know what's follows you, not what proceeds, but backing up this pointer is not easy. It involves the reversal of going back to the beginning, walking your way down to find somebody who points to the last [inaudible]. So it seems like this tail pointer didn't buy us a lot. It helped a little bit, you know, to get some of the operation up and running, but in the end, keeping and maintaining that tail pointer sort of came back to bite us. In the case of the Strategy A, the tail pointer gives us immediate access to the back, which we're going to need for the end queue. If I go to end queue of 40, in this case, then what I'm going to need to do is make a new cell, attach it off of the current cell, and then update the tail, right, to point to that cell while that update's moving forward down the list, right. If you're on the third cell and you add a fourth cell, you need to move the tail from the third to the fourth. Moving that direction's easy. It's the backing up where we got into trouble. So in fact, this suggests that Strategy A is the one we can make both these operations manipulate in constant time in a way that this one got us into trouble. I think we can do it. Let's switch it over. I've got an empty queue here. Also, it is empty in size. This is the same problem with size, which is either you have to go count them or you have to cache the size. I will build the same exactly structure, in fact.

Now, I'll type next. It's going to have both a head and a tail pointer so we can keep track of the two ends we've got going. I think I'm missing my [inaudible]. Slip over. Okay. And then I will make the same comment here about, yeah, you need to delete all cells. So I set the head and the tail to null. I'm now in my empty state that tells me I've got nothing in the queue, nothing at all, so far. And my Y is empty. Oh. We'll check that return equals, equals null, just like it did. In fact, it looks a lot like the stack, actually. And here I'm going to show you something kind of funny. If I go take my stack CCPs pop. So if

you remember what pop did over here, is it took the front most cell off the list. And in the case of the queue that was the popping operation. And it turns out the D queue operation is exactly the same. If we're empty, right, then we want to raise there. Otherwise, we take the head's value. We move around the head, we delete the old memory. So it's not actually the top element. I should, actually, be a little bit more careful about my copying and pasting here. I could really call that front. It's the front-most element on the top. But it's like the exact same mechanics work to take a cell off the front in that way. Well, that's sort of nice. And it's like, yeah, well, since I have some code around I want to go try it. My stack, also, kind of does some things useful in push that I might be able to use. It's not exactly the same because it's adding it to the front, but it is setting up a new cell. So I'm going to make a new cell and I set its value. Then, it's attaching looks a little different. Let's take a look. We omitted a cell, copied the value in, now the goal is giving our pointer to the tail, we want to attach onto the tail. So we know that it's always going to have a next of null, the new cell. So no matter what, it will be the new last cell and it defiantly needs that value that tells us we're at the end of the list. And then we have to attach it to our tail. I'm going to write this code first and it's going to be wrong. So just go along with me in this case. If the tail's next is the new cell. So if I'm wiring in the pointer from the tail onto the cell we have there, and then we want to update the tail to point to the cell. So almost, but not quite, everything I need to do. Someone want to point out to me what it is I have failed to consider in this situation.

Student:[Inaudible].

Instructor (Julie Zelenski):It's the what?

Student:[Inaudible].

Instructor (Julie Zelenski):Yeah.

Student:– trying to make null point to something [inaudible].

Instructor (Julie Zelenski):Exactly. So if my queue is totally empty. So in these cases like – one thing you can often think about whenever you see yourself with this error, and you're dereferencing something, you have to considered, well, is there a possibility that that value is null. And if so I'd better do something to protect against it. So in the case of the new cell here, I'm setting aside and not clearing that cell. We know it's valid, [inaudible], we've got good memory. That part's good but the access to the tail, and that's assuming there is a tail, that there is at least one cell already in the queue that the tail is pointing to.

When that's not true, this is going to blow up. It's going to try to dereference it. So what we can do here is we can just check for that. We can say if the queue is empty then what we want to do is say that head equals tail equals new cell. Otherwise, we've got some existing tail and we just need to attach to it.

Now, this is what I meant about that often there are little bit of special cases for certain different inputs. In particular, the most common ones are an empty list or a single link list being distinguished from a list that has two or more elements. So sometimes it's all about – a single link list has problems and so does the empty list. In this case, it's exactly the empty list. Like a single cell is actually fine, you know, one, ten, 6,000 all work equally well. It's that empty case that we have to work on setting our head and tail to that.

So with this plan we go back to my use of this. And stop talking about my stack and change into my queue, end queuing instead of pushing. [Inaudible]. Oh, 591 errors. Well, that's really – 591, what is that? [Inaudible]. Oh, yeah, this is the year I was going to talk about but I failed to do.

So this is the kind of thing you'll get from failing to protect your [inaudible] from being revisited. And so I'll be – if I haven't deft this funny little symbol I just made up then I want to find it in the end deft. So if it came back around and it saw this same header file, it was supposed to say I already seen that symbol and I won't do it. But, in fact, I had accidentally named one of them with a capital H and one with a lowercase H. It says if this symbol's not been identified then define this other symbol and keep going. And so the next time it saw the header files it said if that this lowercase H hasn't been defined, well it hadn't so it made it again and then it kept doing that until it got really totally twisted up about it. So I will make them match, with any luck, without the caps lock on. Three times is a charm. And then one, two, three pop out the other side of queue, as I was hoping. Okay. So you know what I'm going to do, actually? I'm actually not going to go through the alternate implementation of queue. I'm just saying like given that it works so well for stack, right, to have this kind of single link list, you know, efficient allocation, it's not surprising that queue, like, given to us. And what this is, like I said earlier, I said, well, you know you can do things with vector, you can do things with stack, anything you do with stack you can do with vector if you were just being careful. Right. The pushing, and popping, and the queue D queue, are operations that you could express in other terms of vector adds and removes and inserts that things.

But one of the nice things about providing a structure like a stack or queue is it says these are the only things you're allowed to add at this end, remove from that end, it gives you options as a implementer that don't make sense for the general purpose case. That the vector as a link list had some compromises that probably weren't worth making. But in the case of stack and queue it actually came out very cleanly and very nicely, like this tidy little access to one end, or the other end, or both, right, was easily managed with a link list and then it didn't have any of the constraints of the continuous memory to fight with. And so often having a structure that actually offers less gives you some different options as an implementer because you know you don't have to support access into the middle in an efficient way because the stack and queue don't let you. They don't let you riffle through the middle of the stack or the queue to do things and that gives you some freedom as an implementer, which is neat to take advantage of. Yes, sir.

Student:[Inaudible] in Java that, you know, we supposed to use an Array whenever we could because Array uses more memory –

Instructor (Julie Zelenski):Um hm.

Student: Does the queue and stack use more memory than an Array, for example?

Instructor (Julie Zelenski):So typically they'll be about the same, because of exactly this one thing, which is it allocates tightly so it asks for one cell per entry that's in there, right. And that cell has this overhead in the form of this extra pointer. Right. So in effect it means that everything got a little bit bigger because everything's carrying a little overhead. Okay. So there's a little bit more memory.

The thing the Array is doing is it tends to be allocating extra slots that aren't being used. So it's not usually tightly allocated either. So in any given situation it could be as much as twice as big because it had that extra space. So kind of in the tradeoff they tend to be in the same general range. This one has about twice the space, I think it's about four bites it has a four-byte pointer. The thing's much bigger and it turns out actually this four bite pointer might be a smaller percentage of the overhead and will be a larger amount of the capacity in the excess case.

So for a lot of things they're going to be about the same range. And then there's a few isolated situations where, yeah, if you have very big things being stored having a lot of excess capacity is likely to be a more expensive part of the vector strategy than the link list. But the link list does have this built in overhead for every element. And so it's not the case where you would say right off the bat, well, definitely an array over link list because that the allocation space. It's like, hey, you have to think about the whole picture. That is going to be the whole next lectures. Well, you know, it's about tradeoffs. It's not that one strategy's always going to be the clearly better one. If it is, then we don't need to think about anything else. There are times when you know that one is just great and there's no reason to think about anything else. And then there's other times when there are reasons to think about different ways to solve the same problem. So let me propose a case study that has some meat. You may think along the way, but you're going to really have to pretend that this is going to be relevant at first because it's going to seem like you've been transferred back to 1970, which was a very bad year, long before you born.

I want you to think about the idea of a text setter. So Microsoft Word, or you know your email send window, or BB edit, or X code. All these things, right, have some backing of what usually is called a buffer, a buffer, sort of a funny word, right, but a buffer of characters behind it. Okay. And that buffer is used as the kind of document storage for all the characters that you're editing and moving around. As you kind of bop around and move around there's something often called the cursor where you type characters and characters get moved as you insert. You can select things, and delete them, and cut, copy, and paste, and all that sort of stuff. So what does that data structure look like? What is the kind of thing that wants to back a buffer? What kinds of things are important in that context to be able to support and what operations needs to be optimized for that tool to field [inaudible]? So what we're going to look at is a real simplification kind of taking that problem, kind of distilling it down to its essences and looking at just six operations that are kind of at the core of any text editor about moving the cursor forward and

backwards with these commands. Jumping to the front and the back of the entire buffer and then inserting and deleting in relative to the cursor within that buffer. And we're going to do this with an extremely old school interface that's actually command base. It's like VI comes back from the dead for those of you who have ever heard of such things as VI and E Max. And so we're going to look at this because there's actually a lot of ways you can approach that, that take advantage of some of the things that we've already built like the vector, and stacking queue, and the link list. And sort of think about what kind of options play out well.

What I'm going to show you first off is just what the tool looks like so you can really feel schooled in the old way of doing things. So I insert the characters A, B, C, D, E, F, G into the editor. And then I can use the command B to backup. I can use the command J to jump to the beginning, E to jump to the end, and so F and B, B moving me backwards, F moving me forwards, and then once I'm at a place, let's say I jump to the end, I can insert Z, Z, Y, X here at the beginning and it'll slide things over, and then I can delete characters shuffling things down. So that's what we've got. Imagine now, it's like building Microsoft Word on top of this. There's a lot of stuff that goes, you know, between here and there. but it does give you an idea that the kind of core data requirements every word processor needs some tool like this some abstraction underneath it to manage the character data. Okay. Where do we start? Where do we start? Anyone want to propose an implementation. What's the easiest thing to do?

Student:[Inaudible].

Instructor (Julie Zelenski):In a way. Something like that. You know, and if you think Array, then you should immediately think after it, no, I don't want to deal with Array, I want a vector. I'd like a vector. Vector please. Right. So I'm going to show you what the interface looks like, here, that we have a buffer in these six operations. We're going to think about what the private part looks like, and we say, what about vector. Vector handles big, you know, chunky things indexed by slots. That might be a good idea. So if I had the buffer contents A, B, C, D, E, right, and if I could slam those into a vector, and then I would need one other little bit of information here, which is where is the cursor in any given point because the cursor is actually where the uncertain lead operations take place relative to the cursor. The buffer's also going to manage that, knowing where the insertion point is and then deleting it and inserting relative to that. So I say, okay. I need some index. The cursor actually is between two character positions. This is like slot zero, one, and two. And the cursor is between, actually, one and two. And there's just a minor detail to kind of have worked out before we start trying to write this code, which is do we want the cursor to hold the index on the character that's to the left of it or to the right of it. It's totally symmetric, right.

If I have these five characters, it could be that my cursor then goes from zero to four, it could be that it goes from zero to five, or it could be it goes from negative one to four, depending on which way I'm doing it. I'm going to happen to use the one that does zero to five where it's actually recorded as the character that's after it. Okay. So that's just kind of the staring point. I'm going to write some code and make it happen. So I'm going

to remove this. Reference it, that's okay. And then I'm going to add the things I want to replace it with. So I put in the editor code and to put in the buffer code. Was buffer already in here? I think buffer's already here. Okay. Let's take a look. I get my buffer. So right now, let's take a look what's in buffered. I think I have the starting set information. So it has move cursor over, you know move the cursor around, it has the delete, and it has – let me make it have some variables that I want. I've got my cursor. I've got my vector of characters.

Right now, it has a lot of copying, I think, just in anticipation of going places where copying is going to be required. Right now, because it's using vector and vector has deep copying behavior, I'm actually okay if I let windows copy and go through. But I'm actually going to not do that right away. All right. So then let's look at the implementation side of this and I have an implementation in here I want to get rid of. Sorry about this but I left the old one in here. That will do all the things that will need to get done. Okay. So let's deal with at the beginning. Starting off, right, we will set the cursor to zero. So that's the vector gets started, initialize empty, and nothing else I need to do with the vector. A character [inaudible] cursor position in that, and then moving the cursor position forward, it's mostly just doing this, right, cursor plus, plus, right. It's just an easy index to move it down. But I do need to make sure that the cursor isn't already at the end. So if the cursor is less than the size of the vector then I will let it advance. But it never gets beyond the size itself. So like all of these are going to have same problems. Like, if I'm already at the end and somebody asks me to advance, I don't want to suddenly get my cursor in some lax stated that sort of back that.

Similarly, on this one, if the cursor is greater than zero, all right, then we can back it up and move the cursor around. Moving the cursor at the start is very easy. Moving the cursor to the end is, also, very easy, right. We have a convention about what it is. And so then inserting a character is, also, pretty easy because all the hard work is done by the vector. When I say insert this new character at the cursor position, right, then this character advance the cursor to be kind of past it, and then deleting the character, also, pretty easy. Let me show you, before I go finishing the cursor, I just want to show you this little diagram of what's happening while I'm doing stuff. So this is kind of a visualization of the things that are happening. So this is showing the vector and its length. And then the cursor position as we've done things. Inserted six characters, the cursor's currently sitting at the end, and for the length of the cursor, actually, the same value right now. If I issue a back command, right, that deducts the cursor by one, it shows another one backs up by one, and then eventually the cursor gets to the position zero and subsequent backs don't do anything to it, it just kind of bounces off the edge. And moving forward, things are easier to deal with. It will advance until the gets to the length, the size of that text, and it won't go any further. And that jumping to the front in the end are just a matter of updating that cursor position from zero to the size and whatnot.

The operation where things get a little more bogged down is on this insert where I need to insert a position. If I insert the X, Y, Z into the middle there and you can see it kind of chopping those characters down, making that space to insert those things in the middle. Similarly, doing delete operations, if I jump to the beginning I start doing deletes here. It

has to copy all those characters over and deduct that length, right, so that the vectors do that work for us on remove that. That means that if the vector's very large, which is not a typical in a word processor situation, you could have you PhD theses which has hundreds of pages, if you go back to the beginning and you start deleting characters you'd hate to think it was just taking this massive shuffle time to get the work done. So let's see my – I'm going to bring in the display that – whoops, I don't like something here. [Inaudible]. Now, what do we have? Oh, look at this. Well, well, we will continue on. This inside – okay. Hello. [Inaudible]. I have no idea. I'm not going to try to [inaudible]. Okay. It's inside. That's got me totally upset. Okay. We still have our old code somewhere. Okay. Why do I still have the wrong – what is – okay. Oh, well. Today's not my lucky day. I'm not going to worry about that too much. I'm just going to show [inaudible]. Okay.

So seeing what's actually happening, right, it's like just mostly kind of moving that stuff back and forth, and then having the vector do the big work, right, of inserting and deleting those characters, and doing all the copying to get the thing done. We'll come back over here and kind of see what good it does and what things it's not so hot at, right, is that it is easy to move the cursor wherever you want. You know, that moving it a long distance or a short distance, moving it is a little bit or to the beginning to the end, equally easy as by just resetting some variable. It's the enter and delete, right where this one actually really suffers, right, based on how many characters, right, follow that cursor, right you could potentially be moving the entire contents of the buffer. Adding at the end is actually going to be relatively fast. So if you have to type in order, like you just type you theses out and never go back to edit anything, then it would be fine, adding those characters at the end. But at any point, if you move the cursor back into the mid of the body, somewhere in the front, somewhere in the middle, right, a lot of characters get shuffled as you change, makes edits in the middle there. And so what this seems that it actually has good movement but bad editing as a buffer strategy. That's probably, you know, given that we have these six operations we'll all interested in, this is probably not the mix that seems to make the most sense for something that's called an editor, right. It is the editing operations are the ones that is weakest on, it has its most debilitating performance, it wouldn't make that good of an editor, right, if that was going to be your behavior.

It has one advantage that we're going see, actually, we're going to have to kind of make compromises on it, which it actually uses very little extra space, though, but it's using the vector, which potentially might have excess capacity up to twice that. So maybe it's about two bites per char in the very worse case. But it actually has no per character overhead that's already imbedded in the data structure from this side. Now, I'm just going to go totally somewhere else. Okay. So instead of thinking of it as vector, thinking of it as continuous block, is to kind of realize the editing operations, right, that comes back to the idea, like if I were working at the very end of my document that I can edit there efficiently. It kind of inspires me to think of, how do I think I can make it so the insertion point is somehow in the efficient place. Like the edits that are happening on the insertion point, can I somehow make it to where access to the insertion point is easy? That rather than bearing that in the middle of my vector, if I can expose that to make it accessible easily in the way the code manipulates the internal of the buffer.

And so the idea here is to break up the text into two pieces. Things that are to the left of the cursor, things that are to the right, and I'm calling this the before and the after. And then organize those so they're actually averted from each other to where all the characters that lead up to the cursor are set up to where B is very close, assessable at the top of, in this case, a stack, and that the things that are farther away are buried further down in the stack in that inaccessible region. And similarly over here, but inverted, that C is the top of this stack and D and E, and things are further down, they are buried away from me, figuring the things I really need access to are right next to the cursor that I'm willing to kind of move those other things father away to gain that access. And so what this buffer does is it uses two stacks, a before stack, an after stack, and that the operations for moving the data around become transferring things from the before to the after stack. Where's the cursor? How does the cursor represent it? Do I need some more data here? Some integer some –

It's really kind of odd to think about but there is no explicit cursor, right, being stored here but the cursor is, in this case, right, like modeled by what's on the before stack or all the things to the left of the cursor. What's on the after stack is following the cursor. So in fact, the cursor is represented as the empty space between these two where you have, you know, how many ever characters wrong before is actually the index of where the cursor is. So kind of a wacky way of thinking about it, but one that actually have some pretty neat properties in terms of making it run efficiently. So let me show you the diagram of it happening. So insert A, B, C, D, E, F, G. And so, the operation of inserting a new character into this form of the buffer is pushing something on the before stack. So if I want to move the cursor, let's say I want to move it back one, then I pop the top off of the before push it on the after. I keep doing that and it transfers the G to the H, you know, the E and so on, as I back up. Moving forward does that same operation in the inverse. If I want to move forward I take something off the top of the after and push it on before. So it's kind of shuffling it from one side to the other. Given that these push and pop operations are constant on both implementations of the stack we saw, then that means that our cursor movement is also of one so I can do this. And if I do deletes it's actually just popping from the after stack and throwing it away. So also easy to do edits because I'm talking about adding things to the before and deleting things from after, both of which can be done very efficiently.

The one operation that this guy has trouble with is big cursor movement. If I want to go all the way to the beginning or all the way to the end, then everything's got to go. No matter how many things I have to empty out the entire after stack to position the cursor at the end, or empty out the entire before stack to get it all the way over here. So I could of sort of talk you though this code, and actually, I won't type it just because I actually want to talk about its analysis more than I want to see its code. Each of these actually becomes a one liner. Insert is push on the before stack. Delete is pop from the after stack. The cursor movement is popping from one and pushing on the other depending on the direction and then that same code just for the Y loop, like wild or something on the one stack, just keep pushing and popping from one empty to the other. So very little code to write, right, depending a lot on the stacks abstractions coming through for us and then making this ultra fit that we've managed to get editing suddenly, like, a lot more efficient

but the cost of sacrificing one of our other operations. Let's take a look at what that look like, right.

Suddenly I can do all this insert and delete at the insertion point with very little trouble, and I suddenly made this other operation, oddly enough, slow. All a sudden it's like if you go back to the beginning of your document you're in the middle of editing, you're at the bottom. You say, oh, I want to go back to the beginning and you can imagine how that would feel if you were typing it would actually be the act of going up and clicking up in the top right by where you made the insertion and all a sudden you'd see the white cursor and you'd be like, why is that taking so long. How could that possibly be, you know, that it's doing this kind of rearrangement, but yet it made editing even on a million character document fast. But that movement long distances, you know, jumping a couple of pages or front to back, suddenly, having to do this big transfer behind the scenes, so different, right. So now I had six operations, I had four fast, two slow, now I have four fast, two slow. It still might be that, actually, this is an interesting improvement, though, because we have decided those are operations that we're willing to tolerate, being slower, especially, if it means some operation that we're more interested in, if performance being faster. And that likely seems like it's moving in the right direction because editing, right, being the whole purpose, right, behind what we're trying to do is a text setter, that seems like that might be a good call.

What I'm going to start looking at and it's going to be something I want to talk about next time, is what can a link list do for us? Both of those are fighting against that continuous thing, the copying and the shuffling, and the inserting is there something about that flexibility of the rewiring that can kind of get us out of this some operations having to be slow because other operations are fast. Question.

Student:[Inaudible].

Instructor (Julie Zelenski):Yeah.

Student: Why is space used for stacks when those are half?

Instructor (Julie Zelenski):Yeah. So in this case, right, depending on how you're modeling your stack, if it is with vectors, right, then you're likely to have the before and the after stack, both have capacity for everything, even when they're not used, right. And so it's very likely that either right give 100 characters they're either on one stack or the other, but the other one might be kind of harboring the 100 spots for them when they come back. Or it could just be that you have the link list, which are allocating and de allocating but has the overhead of that. So it's likely that no matter which implementation the stack you have, you probably have about twice the storage that you need because of the two sides and the overhead that's kind of coming and going there. So we will see the link list and we'll talk that guy through on Friday. I'll see you then.

[End of Audio]

Duration: 51 minutes

ProgrammingAbstractions-Lecture21

Instructor (Julie Zelenski):Hey, welcome to Friday. I've got a big voice today. We are going to pick up on the editor buffer case study where we left off on Wednesday's lecture. So talking about the link list implementation and kind of see what that gets us. And what we have left of today's lecture, we'll start talking about the map implementation. So we have two other classes to go, on which is map and set, and for those we're going to learn about the concept of binary search trees and hashing. And then that kind of completes the whole picture of all the pieces in the AT library. So we'll be in a good position from there to kind of pick up some advanced topics before we close off for the end of the quarter.

The reading is, we're finishing the last half of chapter nine and then starting on chapter 13, which is where the binary search tree stuff is covered. And then we will come back up and pick up chapter 11 after that. We will not do chapters 12 and 14. So if you are keeping track of things in the text, we will not cover those at all. I'm going to go to the Truman Café today after class, so if you have free time this afternoon and want to come and hang out and have a little snack and beverage with us. I would love to have you join me. We'll also do that next Friday, and I think that will be the last one I'll be able to do, because the very last of the quarter I'm going to be at a conference that day. So hopefully one of the last two days you can come and visit with us.

So let me ask a little bit about priority queue. How many people have started priority queue somewhere? Gotten some stuff done? Okay that's not a lot of you. What are the rest of you guys doing? How many people have finished at least one of the implementations? So people who have started. How many have finished both implementations? How many people are taking the weekend off? Hold your hand up. Any advice, from those of you who finished one or the other of the implementations, that you want to offer up to those who are a little bit farther back behind you? If you want them to struggle the way you did, you can just keep quiet. You can just smile. Did you do them in the order that they are listed? Did you link-list them in heap and that worked out okay? All good? Did anybody do them the other order just in case?

All right, that is your weekend excitement. Let us refresh about where we were at the end of Wednesday. As we had talked about the vector form of the buffer, so the editor buffer that's backing the word processing client, and the two stack version, where we're shuffling stuff, often on the before and after stacks to get things done, right? And when we were looking at the main six operations, we're looking at the four cursor movements and the two editing operations, right, that we had traded off. Where editing had been slow in the vector, to where editing was now fast, but then these long distance movements were slow, because of the shuffling. And then there was a little discussion at the very end about how this probably increases our space requirements, because we're now keeping two stacks that are each capable of holding the entire contents as opposed to one vector here with its slop.

So that was where we were at. We are still in kind of the pursuit of this Holy Grail of could we get it to where everything is fast. Is it always a matter of tradeoffs or could we

just invest some more smarts and more cleverness and hard work and squeeze it down to where we could get everything to perform efficiently? So typically when computer scientists say they want something to perform efficiently, they're hoping for log in time or faster. Constant time is the best. Log in grows so slowly that in fact, logarithm time is effectively constant for all reasonable values of N up to millions and billions. And so typically anything that is linear or higher is often a weak spot that you are trying to look at, and certainly things that are quadratic or exponential are definite opportunities for improvement.

So let's look at this idea of the link list, so that both the vector and the stack – we're relying on this idea that continuous memory is the weak link that was causing some problems. In the vector, that shuffling to move things up and down, or in the case of the stack version, one side to the other was part of what we were working up against. And the link list promises to give us this opportunity to have these each individual character being managed separately, and flexibility might help resolve some of our problems here. SO what we're going to look at is the idea of implementing the buffer using a link list.

So if I have characters A, B, C, D, E, I could have a link list here where I'm pointing to the head cell, which is A, which points to B, and C, and D, and E, and down to the null. SO what I'm modeling it as in the private data section will be a little link list node with a character in there, a pointer to the next one. And then I'm going to keep two pointers, one to the front-most cell, and then that's going to model where the cursor is within the buffer. So rather than doing this like an index, an index was handy in the case of vector where we had direct access, right? Given the way the link list works, knowing that it's at the fifth cell, the fourth cell isn't really very helpful. It would be more helpful to have the pointer kind of directly in the midst of things to give us direct access to where the cursor is.

But let's think a little bit about that cursor, because there's an important decision to be made early about helping to facilitate the later work we have to do. I have the contents A, B, C, D, E, so the cursor is actually between two characters. If the cursor right now is situated after the A and before the B, what I'm modeling is A cursor B, C, D, then it seems like the obvious two choices for where the cursor might point is to A or to B. And I had said in the case of the vector that if it were index zero or index one that there wasn't a really strong preference for one over the other. There is going to be a really good reason to pick one for the link list over the other. Would you rather point to the A for where the cursor is or would you rather point to the B?

Student:[Inaudible]

Instructor (Julie Zelenski):Exactly, so the idea is that when the cursor position is there, the next edit actually goes in between the A and the B. So if I insert the character Z, it really does go right here. And if I am going to, if effect, wire that in and splice it in then list, if I'm pointing to B then I'm kind of hosed, because I'm already one past where I need to be able slice in. Because I need to know what was behind it to bring it in. So I'm pointing to the character behind the cursor gives you access to wiring this down to here

and this into there, and then updating the cursor to point to the Z without having to do any of the difficult backward looking backward traversal.

So that strongly suggests what it is, it's going to point to A when it's between A and B; B when it's between B and C; and so on. There is another little point here, which is there are actually five letters in the cursor, but there are six cursor positions. It could be at the very beginning, in between all these, or at the very end. And the way we've chosen it right now, I can identify all of the five positions that have at least one character to the left, so being – pointing to the A means it's after A; pointing to the B it's after B; and so on, all the way down to the E. But I don't have a cursor position for representing when the cursor is at the very beginning of the buffer.

So one strategy I could use, right, is to say I'll just use the special cursor position of null. But I need one more pointer value to hold onto, and I could make a special case out of this because there's null. Once I've done that though, I've actually committed myself to this thing that's going to require all of my code to be managing a special case of, well when the cursor's null do something a little different than when the cursor is pointing to some cell. I'm going to show you a way that we can kind of avoid the need for making a special case out of it, by wasting a little memory and reorganizing our list to allow for there to be a sixth cell, and extra cell that we call the dummy cell.

So in this case that cell would go at the very front. It would have no character, just let the character field be unstatused. It's not really a character. What it is, is just a placeholder. And when I created the buffer to begin with, even when it's empty, it's always going to have that cell. I'm going to create that cell in advance, I'm going to have it sitting ready and waiting, and that cell never gets deleted or moved or changed. In fact, the list always, the head of this will always point to the same cell from the beginning of this object's lifetime to the end. So it will never change this at all, and the cells that follow are the ones that are going to be changing and growing and rearranging in response to editing commands.

What this is going to buy us is a couple of things. First off, it's going to make it so there is a sixth cursor position. So now when the cursor is at the very beginning, we can set the cursor to point to this dummy cell. Okay that's one thing that we solve by doing that. Another thing that we've solved is actually we have made all the other cells on the list, the ones that are being edited and manipulated. We've made them all totally symmetric. Previously there was always going to be a little bit of a special case about, well if you're the front of the list, we're seeing that sometimes we're doing inserts on the stack in queue link list.

That inserting later in the list involves kind of wiring in two pointers; one to command and one to go out. But there was a special case that oh, if you're the very first cell of the list, then your outpointer is the same, but your incoming pointer is resetting the head. That piece has actually gone away now, that now every cell in there always has a predecessor. It always has a previous, right? Something behind it so that actually they will always be pulling the pointer in and pulling the pointer out for every cell with

character data that's being modified. And that means some of that extra little handing of oh, if you're the very first cell, have gone away. The very first cell is always this kind of dummy cell and doesn't require us to go out of our way to do anything special for it.

So this is going to solve a bunch of problems for us, and it's kind of a neat technique. This thing is used actually fairly commonly in situations just managing a link list just to buy yourself some symmetry in the remaining cells.

So let me look at some code with you, and I'm going to actually draw some pictures as I go. I'm not going to write this code in real time, because I think it's actually more important to see it acting on things than it is to see me typing out the characters. But the mechanism, right, in the case of the buffer here is it's going to have a head point, which points to the front most cell; and a cursor, which points to the character that proceeds the cursor. So if right now the contents of the buffer are the contents A, D, C. And let's say that right now the cursor is pointing to A. Actually, I need my dummy cell. Let me move everybody down one, so you just have dummy cell and my A and my B.

And so the cursor right now is at the very front of the buffer. This is kind of what it looks like. It has two characters, the A and the B. I'm going to trace through the process of inserting a new character into the buffer. It is that my local variable CP is going to point to a new cell allocated out in the heap. Let's say the character I'm going to insert is the Z, so I'll assign CH to the character, the slot right there. And then the next for this gets to – what the cursor is – kind of following the cursor. So the cursor is pointing to the cell before it, and the next field of that cell tells us what follows it; so if I trace my cursor's next field, it points to A, and so I'm going to go ahead and set up this new cell's next field to also point to A.

So now I've got two different ways to get to the A cell, tracing off the dummy cell or tracing off this new cell. And then I update the cursor's next field to point to CP. So I just copied right there that CP's next field was pointing to A. I copied it down here to kind of do the splice out. Now I'm going to do the splice in, causing it to no longer point to A, but instead to point to my new cell down here, CP. And then I update the cursor to point to this new cell. That just means that subsequent inserts, like the C is kind of behind the cursor after I've made this edit. That goes away when that one ends, and so now the new structure that I have here is head still points to the dummy cell that never changes. Dummy cell now points to Z, Z points to A, Z points to B, and then the cursor in this case is between the Z and the A, by virtue of the fact that the cursor is pointing to Z.

A subsequent one would kind of do the same thing. If I typed out ZY after I did this. We'd make a new cell over here. It would get what was currently coming out of the cursor's field. The next field, right, we would update cursor's next field to point to this new one. And then we would update cursor to point to this guy. And so now following the list, right, dummy cell, Z, Y, A, B, cursor pointing to the Y, which is the character directly to the left of the cursor. And so inserting in any position now, front cell, middle cell, end cell, doesn't require any special case handling. So you don't see any if of things.

And you notice that you'll never see an update to the head directly here. That the head was set up to point to the dummy cell in the constructor and never needs any adjusting. It's only the cells after it that requires the new arrangement of the pointers coming in and out.

When I'm ready to delete a cell, then the mechanism for delete in the buffer is to delete the thing that follows the cursor. So if I have Z, Y, A, B, this is the character that delete is supposed to delete, the one after the cursor. And so, the first thing it looks at is to make sure that we're not already at the end. In the case of the link list form of this, if the cursor was pointing to the last most cell, and then we looked at it's next field and saw that it was null, that would be a clue. Oh, it's pointing to the very last cell, there's nothing that follows it. So we have a quick test to make sure that there's something there to delete. There is something following the cursor, and in this case since the cursor is pointing to the Y, we're good.

It says look at its next field, it points to A. It says okay, call this thing old and point to that A. Now do a wire around it. So take the cursor's next field, and so this thing is where I'm targeting my overwrite, and copy out of the old its next. It's no longer pointing to the A, but instead it's pointing to the B. And deleting the old, which causes this piece of memory to be reclaimed, and now the contents of my buffer have been shifted over to where it goes Z, Y, and straight to D. The cursor in this case doesn't move. It actually deleted to the right, and so whatever was to the left of the cursor just stays in place. So again, no need to update anything special for the first or the last or any of those cells, but the symmetry of all the cells past the dummy kind of buys us some convenience of handling it.

Student: What does a dummy cell look like?

Instructor (Julie Zelenski):The dummy cell looks like just any other cell. It's actually a cell, T, and what you do is you just don't assign the Christian field, the character field, because it has nothing in it. So it actually looks just like all the others, it's just another node in the chain. But this one, you know you put there purposefully just as the placeholder. So it doesn't contain any character data. So if you're ready to print the contents of the buffer, you have to remember the dummy cell is there, and just skip over it. You tend to start your list at heads next, the first interesting cell to look at, then work your way down to the end. So it's just like all the others, but you just know that the character data here is useless, it's not important. You didn't even write anything in there.

Student: When you delete, you just go left?

Instructor (Julie Zelenski):It turns out the delete in this case goes forward. So you can imagine that in some other world it does, but all the ones we have done so far have always deleted forward. So the delete – for example, the stack case, grabbed from the after stack – it's done that way, because it happens to simplify some other things that you can imagine that delete and reverse. What would it take? Well, you'd have to back the cursor up. And we're going to find out pretty soon that would make it challenging for the

link list. So partly we've picked a delete that made certain things work out a little better for us.

So both of these operations are O of 1; this is where the link list really shines, right, on this sort of stuff. Because if you have access to the point at which you want to make a modification, so if you are already kind of in the midst of the thing you're trying to rearrange, it's often to the link list's advantage, because it actually can do these this rewiring in the local context. And even if there's thousands of characters that follow the cursor. It doesn't matter. And there can be thousands of characters preceding the cursor for that matter, too. The idea that there's a much larger context that's working and doesn't impact its performance; it's really just doing this local little rearrangement of the pointers. And that's a real strength of the link list design is that flexibility that doesn't rely on everything living in contiguous memory.

So let's talk about the movement options for this guy. So right now I have the contents Z, Y with the cursor between that, and the B following that. So three characters worth, and then the initial two operations are pretty simple. The later two are going to be a little bit messier. So let's look at the easy ones first. Moving the cursor to the beginning, so that jump that pulls you back to the very beginning, very easy in a link list form. If I want to move the cursor to the beginning, all I need to do is take that cursor pointer, and reset it to the head, which is where the dummy cell is.

And so, by virtue of doing that, the character it points to is the one that's to the left, right of that, and to the left of that the dummy cell is one we're not counting. In fact, it's like nothingness that's there, and that means that the first character that follows the cursor is Z, which is the initial character of the buffer. So that reset, easy to do, we have easy access to the front of the list, and so no special crazy code required.

Moving the cursor forward, also an easy thing to do, link list are designed to make it easy to work your way from the front to the back. And so moving the cursor forward is advancing that cursor by one step. So in the array form of this, what was a cursor plus, plus, the comparable form of that in the link list is cursor equals cursor next. It has a little if test there, all of the versions actually have something that is comparable to this. If you're already at the end, then there's nothing to be done, so you don't advance the cursor off into space. You check to make sure that there is something for it to point to. So in this case, seeing if the cursor's next field is null, it's not, it points to Z. Then we go ahead and update the cursor to there, which has the effect of changing things to where the Z is the character to the left of the cursor, and then the YB follows it.

So moving this way, getting back to the beginning, and kind of walking our way down is kind of easy to do. Now we start to see where the link list causes us a little bit of grief, moving the cursor to the end. So now, starting at the beginning or the middle, or wherever I am, I want to advance my way to the end. I don't know where the end is. Link lists in general don't keep track of that, that in the simple form of the single link list, I'm going to have to work to find it. So I'm going to take where I am and walk my way down. And this one actually just makes use of an existing public number function to move

cursor forward. It's like while the cursor next does not equal null, which is to say while we are not pointing at the very last cell of the link list, then keep going. So advance cursor equals cursor.

Some will say, "Is the cursor's next field null?" No, then set the cursor to be the cursor's next field. Is the cursor's next field null? No it's not, so advance the cursor to the next field. Is the cursor's next field null? Yes. So that would the last thing in the buffer, and so if we have hundreds or thousands or tens of characters, whichever it is, right, and depending on how close we already were, it will walk through everything from where the current editing position was to find the end of the buffer one by one, working it's way down.

Even more painful operation is the simple one of just moving backwards. If I'm pointing right now to that B, and I'd like to get back to that Y, the link list is oblivious about how you got there. It knows where to go from here, but that backing up is not supported in the simple form. In order to find out what preceded the V, our only option right now is to go back to the beginning, and find it. So starting this pointer CP at the head, and saying, do you point to; is your next field the cursor? And this one says, no. And then it says, well okay advance. Does your next field point to the cursor? No it's not. Go to this one, does your next field point to the same place as the cursor, yes. Wise next field, points to the same place the cursor does. That must mean that Y was the one that preceded the cursor in the link list.

So after having gone back to the beginning and walked all the way down, especially if I'm near the end, that's a long traversal. And when I get there, I can say "Oh yeah that's the one, back it up to there." Walking it all the way down; some good link list coding, good practice in there, but again a funny, funny set of trade offs, right? Does anybody have any questions about any part of the code?

Student: At the end could you just save the address of the pointer, and then the person who wants to go to the end could just access that?

Instructor (Julie Zelenski):That was a great idea. It's like, so what about improving this? Right now all we have is a pointer to the front, and each pointer has only a pointer to the next. It's like, maybe we need to add some more stuff, track some more things. What if we tracked the tail? Would that help us? If we tracked the tail, we could move to the end quickly. Would that help us moving backwards? No. It solves one of our problems though, so you're on to something. Let's look at what we've got, and then we'll start thinking about ways to fix it, because that's going to be one of the pieces we're going to think about.

So if I move to link list relative to what I have, again, it's like a shell game. The old ends moved. It used to be that editing was slow and we didn't like that. Then we made big movement fast, and now we have this funny thing were moving in certain ways is easy and moving other ways is bad. So moving down the document is good, moving backwards in the document not good. Moving back to the beginning is good, but moving

to the end is bad. But then editing in all situations is good, another sort of quirky set of things. It just fells like it's like that game where the gophers pop there heads up, and you pound on them. And they just show up somewhere else. As soon as you get one thing fixed, it seems like something else had to give.

This, I think, would actually be my ideal editor, because it turns out I have exactly this working style, which is like, I'll be down at the end of the document, and I'll have to be generating new things. And I'm getting tired and I don't like writing, and what I keep wanting to do is I go back to the beginning. And I read that again, because I worked on that the most and it's nice. And so I go back to the beginning, and I'm like oh yeah, look at this. Look at how lovely this is, and then I never want to go back to the end where all the trouble is. So I just keep going back to the beginning and editing in the front, and then slowly get back to the end. And then I don't like it again, and then I go back to the beginning. But I never actually willingly go to the end to face my fears. So this is the editor I need.

Also has a bit of kicker in terms of space. So if the space for a character is 1 byte, which it typically is, and the space for a pointer is 4 bytes, then each of those link list cells is costing us 5 bytes, right? Which is sort of five times the amount of storage that the character itself would have taken alone, so in the vector it's fairly tight, a little bit more going up on that. But we've actually kind of blown it up by another factor of two even beyond the stack, which is not quite a good direction to be going.

So let's talk about what we can do to fix that. I'm not going to go through the process of this. I'm just going to kind of give you a thought exercise, so you can just kind of visualize it. What we have here, right, is an information problem. We only are keeping track of certain pathways through that link list. What if we actually increased our access to the list? We don't need to have the full access of a vector, being able to know the nth character by number is not actually as far as we need to go. But we do just need a little bit more coordination between a cell and its neighbors.

So the two things that would buy us out of a lot of the things that we're seeing here is if we added a tail pointer. So if I just add a tail pointer that points to the last cell, then I can move to the end quickly. You say cursor equals tail; the same that cursor equals head. So what was an O of N problem now is an O of 1. The backing up, not as easily solved by just having one thing that's going on; I also am going to need, on a per cell basis this ability to move backwards.

Well, it's just symmetry, right? If A knows where B is, and B knows where C is, and C knows where D is, what's to stop it from also just tracking the information going in the other way, so that D knows that it came from C, and C knows it came from B, and so on. And so having this completely symmetric parallel set of links that just run the other direction, then buys you the ability to move backwards with ease.

The other thing that this would give you is the tail pointer is almost not quite enough to give you the exact place of the end. It gives it to you, but you also have to be careful

about what is it going to take to update it? And so inserting can easily update the tail pointer. It's the deleting that kind of would get you into trouble. If you were here and deleting, you would have to kind of keep track of making sure if you were deleting the thing that was the tail pointer, so you have to be a little about making sure you don't lose track of your tail. But once I have both of these in place, I suddenly have something that can move forward easily, O of 1; move to the front easily, O of 1; move to the tail easily, O of 1; move backwards O of 1. So I can make all of my operations O of 1.

Deleting, you're still going to have to delete the individual cells, but there's one place where it suffers, but that's a much more rare occurrence. And I can get all six; it moves fast, in every dimension; inserts and deletes all good. And there are two places we paid for it. One is certainly the memory. So a 4 byte pointer one direction, a 4 byte pointer another direction, plus the 1 byte for the character means we have a 9 byte per character storage usage.

And the other way, which is a little bit hard to represent with some number in a quantitative way, but actually it's also an important factor, which is your code got a lot more complicated. If you looked at the code for the vector of the stack version, it was a few lines here a few lines there. If you looked at the code for the link list singly we looked at, it's like you've got to wire up these pointers. Now imagine you've got to wire up twice as many pointers. Not only do you have the in and out going one direction, you'll have the in and out going the other direction, and just the opportunity to make errors with pointers has now gone up by a factor of two.

The idea that you could get a vector version of this up and running in no time, compared to several days, let's say, of getting the doubly link one working. It might be a factor to keep in mind. If you're kind of investing for the long haul, sure, work hard, but know that it wasn't for free. So you've got 89 percent overhead. That's a lot of overhead for small bits of data. More likely, and this is actually the way that most modern word processors do do this, is they don't just make one or the other; it's not just a array or a stack or a link list. They actually look at ways to combine the features of both to get a hybrid, where you can take some of the advantages of one pipe of data, but also try to avoid its worst weaknesses.

So the most common way they're going to do this is some kind of chunking strategy, where you have a segment of the buffer that kind of is moved aside and worked on in one little array. It might be that those arrays are ten or 20 or a sentence long, or paragraph long. It might be that they are actually dictated by when you change styles. All of the code that's in one font kind of gets moved into one chunk, and as soon as you change styles, it breaks into a new chunk that follows to kind of keep track of the formatting information. It depends on some of the other features that are present in the processor here, but it's likely that rather than having all 1,000 or all 10 million characters in one data structure, they're actually kind of separated to where you have a little bit of both.

So the most likely thing is that there is some kind of array strategy on a chunk basis. There's a linking between those chunks. And what this allows you do is to share that

overhead cost. So in this case, if I had every sentence in a chunk by itself, and then if I had next and previous pointer, which pointed to the next sentence and the previous sentence, when I need to move the cursor forward or backwards within a chunk, it's just in changing this index. Oh, it's in index two, but now it's moving to three, or four, or five. And then when it gets to the end of that chunk, to the end of that sentence, and you move forward, well then it follows the next link forward.

Similarly moving backwards, right, so within a chunk doing array-like manipulations, and then as it crosses the boundary using a link list steps to get further down. So when it needs to move to the front or the back, those things are easily accessible using head and tail pointers. When I need to do an edit, it operates in this array-like kind of way when there's space within that chunk. So you start editing in a sentence, then it would kind of do some shifting on that array. But the array itself being pretty small means that although you are paying that cost of shuffling, you are doing it on a smaller chunk. It's not 1,000 characters that have to move, it's 20 or 40 that have to move within that sentence.

And that when you need to insert a new sentence or a new chunk in the middle, you are doing link list-like manipulations to build a new chunk and insert and splice, so you have that flexibility, where the many characters aren't all affected by it. So you have kind of local array editing and global link list editing that gives you kind of the best of both worlds. But again, the factor is cost showing up in code complexity. That what you are looking at is kind of space/time tradeoff. Well, I can throw space at this, and most people will say that in computer science, "Well I can throw space and this and get better times." So the double link list is a good example of something that throws space at a problem to get better time. There's a lot of overhead.

Moving to the chunk list says I want to get back some of that space. I'm not willing to invest 90 percent overhead to get this. Can I find a way to take a whole sentence worth of things and add a few pointers here? That means that there's only two 4 byte pointers per each sentence as opposed to each character, and that really way reduces the overhead. But then now, the problem is still back on your shoulders in terms of effort, now I have both an array, and a link list, and double links, and a lot of complicated stuff flying around to make it work. But it does work out.

It's kind of the process that designers go through when they're building these kinds of key structures. It like, what are the options, and do any of the basic things we know about work? What about building even fancier things that kind of take the best of both world and mix them together and have a little bit of the weaknesses of this a little of that, but none of the really awful worst cases present in the end result. So in this case you'd have a lot of things that were constant time where constant was defined by the chunk size. So O of 100, let's say, if that was the maximum chunk size to do these things, as opposed to O of N for the whole block net, so kind of a neat little thing to think through.

There was a time, just so you can feel gratified about how I've worked you very hard, but actually in the past I've been even harsher. We actually had them build the doubly linked chunked list editor buffer, and now we have you build the singly linked chunk list PQ,

and it's actually still very complicated, as you'll find out, but not nearly quite as hairy as the full in both directions every way craziness. And you can kind of imagine in your mind. Any questions about edited buffer?

We've got two more data structures to implement, two more things that we've been using, and happy to have in our arsenal that we need to know how they work. Let's talk about map first, and in fact the strategy that I'm going to end up showing you for map is actually going to be the same one we use for set, then we'll come back and think of an even more clever way to do map, too. So we've got two things to talk about, which are binary search trees and hashing. We'll do binary search trees first, and then we'll get back to the hashing the second time around.

So map is, next to vector, probably the most useful thing you've got going on in your class library, all kinds of things that do look up based activities; looking up students by ID number, phone numbers, by name, any of these kinds of things where you need to have this key at associated satellite information. And you'd like to be able to do that retrieval and update quickly. The map is the one for you. So if you remember it's key value fears, it's all about key being the identifying mark that allows you to find the associated value. And then the value is actually just being stored and kind of retrieved without using it or examining it in any kind of interesting way. And what we're really interested in here is how to make this guy work really efficiently.

So efficiently being hopefully log in or better, if we could get that on both the update operations that add and remove things from the map, as well as the look up operation, then that would please probably all our clients immensely, if we could get both of those going well. So I'm going to build you on that. I probably have enough time to do that. That attacks it from using simple tools to start with, just to kind of get a base line for what we can do.

Vector, our old friend vector, can't get too much mileage out of vector, apparently, because you can always find a way to make it do something useful for you. It gives us the convenience of managing that thing with low overhead, direct access is good. We make a parastruct that has the key and the value together. We store it into a vector, and then we may or may not sort that vector. But if it's sorted, there's probably some good reasons. Think about what order to sort it in, what information to be using to sort that, and then we'll get value to add.

We're just going to go through doing it, rather than talk about it. We'll just go over and make a vector happen. Let's get over here in X-code. Let's take a look at my map dot H, and all I'm going to put in there is actually add and get value. And the other one contains key and remove." But these are these two operations that really matter to us, getting something in there, and getting something back out. And so I'm going to start by building this thing on vector. So let me go ahead and indicate that I plan to use vector as part of what I'm doing. So I defined a little structure that's the pair. I'm going to call that "pair" even, that's a nice word for it. So the pair of the key and the value that go together, and then what I will store here is a vector of those pairs. They'll give me a key to value; I'll

stick them together in that little struct, stick them into my vector, and hopefully be able to retrieve them later.

So given that I'm depending only on vector, I don’t have any other information, I don't technically need to go out of my way to disallow then memory-wise copying, because vector does a deep copy. But I'm going to leave it in there, because I plan on going some place that is eventually going to need it. I might as well be safe from the get go. The vector will be set up and destroyed automatically as part of management of my data member. So I actually don't have any explicit allocation/deallocation, so if I look at my constructor and destructor they'll be totally empty. And then add and get value are where I'm going to actually get to do some work.

Before I implement them, I'm just going to think for a second about how this is going to play out. Let's say I'm doing a vector that's storing strings and their length. Let's say, it's just a map of string, and if I do M dot add car. I'm going to store a three with it, so on the other side of the wall, what we're going to do is make a little struct that has the word car and the number three with it, and we package it into our ray. And so then we get a second call to put something in like dog, also with a three, then we'll make a second one of these and stick it in our ray. And so on, so the idea is to have the vector really kind of manage where the storage is going. And then what we'll just do is package it up and stick it in there.

The one thing, though, that I have to think a little bit about, because it sounds like add should be pretty easy. It almost seems like what I need to do is something as simple as this. And I'll start to write the code, and then you can tell me why this isn't what I want to do. I say pair TP, and I say P dot key equals key P dot value equals val, and then I say entries dot add P. Almost, but not quite what I want, what have I forgotten to take care of? If it's not already in there; this has to do with just what's the interface for nap. What does nap say about it if you try to add for a key a different value.

So in the case of this one, it didn't quite make sense that I would add something, but let's say I accidentally put car in there with a link to the first time that I went to go fix it, right, that my subsequent to ask it store car with the proper value, three, doesn't create a totally new entry. It has to find the existing entry that already has car and overwrite it. So the behavior of mad was it could add when it was not previously existing. It also could be an overwrite of an existing value. So we do need to find if there is an entry that already has that key, and just replace its value rather than add a second entry with the same key. So at any given point, right, in this whole vector, there should be one entry for each unique key.

So I have to do a search. And let me write this code, and then I'm going to mention why I'm going to decompose it in a second. If I do entries dot size, I plus plus then if entries of I dot key equals key. Then let's break. Let me pull out I so I can get access to it when I'm done. So after this loop exits, if I is less than entries dot size, then I know that it found a match. In the case where it went through all of them, and it never hit the break. Then I will have gone all the way to where it is exactly equal to entries dot size; if it didn't then I

just have a place to overwrite. And I can say entries of I dot val equals val. Otherwise I go through this process of adding a new one. I had to go hunt that thing down and replace it.

That code, when you look at it, you can say, "Gosh I feel like that code's going to be familiar." Because that idea of searching to find a match to the key probably is going to come in really handy when I need to do get value, but it has the same exact problem of oh, I've got to go find that match. So in fact, that kind of motivates the idea of why don't I just take this little piece of code and separate it out and do a helper that they can both use. I may not have planned for this from the beginning, but once I see it happening, I might as well fix it. So I can say find index for key that given a key will return to you the vector index at which that key is or a negative one. I don't know if I need to keep that one anymore. If it ever found it, it returns I otherwise it returned negative one. That can be our not found.

I can actually use that up here, and then I can say, if found does not equal negative one. Then we do that, and then that tells us that this little piece of code is going to come in handy right up here, get value if found does not equal negative one, then we can return that and what is the behavior of get value if it didn't find it? Does anybody remember? I ask it to get the value in this case for lollipop and it's not there, does it turn zero, what does it do? Untracked forget value, remember? This is an error.

There is not sort of a general case return value you can produce here that will really sort of sufficiently describe to someone who's made this call in error what happened, right? I can't return zero, because it happens to be that sometimes I'm putting in, and sometimes I'm putting strings or structs kinds of things. There's no general zero or negative one to use, so the behavior of get value is if you ask it to retrieve a value for a key, it ought to be there. So the corresponding implementation would likely also just go through the process of contains key, which we call find index for key and check. It could be used by the client if they need to know in advance that it's really there.

So let's see how I'm doing. Let's go back to my code here, and this being not my specialty, and just see that I can put something in and get it back out would be a good first test to see that things are going as they should. Oh, no it didn't like that. Oh yeah, find index for key. Find index for key was not populated clear in my dot H, so let me go move it there. I'll put it in the private section, because I don't want this to be something that is exposed outside of the implementation. The idea that there is an index and there is a vector is not really something that I would want to make part of my public interface. It's really just an internal detail of how I'm managing stuff.

Something about what I did, oh yeah, I used the right name for that. One more case let me just take a look at them both before I let the compiler tell me what is and what isn't right about them. Their both doing find index for key using linear search. If it comes back at not negative one pulling out the matching value and overriding it, and then in the case of adding or replacing the error – so the previously stored value for that is three, and if I go and I do one of those tests it's always good to know what happens if I ask it for

something that doesn't exist. Making sure that the error handling that I put in there does what it was supposed to do, right? Are there any questions about the code that I wrote here?

The vector is always kind of a good starting place for these things, because actually it just tends to use very simple things, and it tends to lend itself to easy code. But because of vector constraints being at a contiguous back structure, right, there's likely to be some performance implication. And in particular, in the unsorted case here, both add and get value involve doing a linear search. In some cases it will find it quickly at the very beginning or even half way through. In other cases it won't find it at all and have to search through the whole thing. So on average, if you figure it's equally likely to be in any position or not there at all, it's at going to at least have to look at half of them or more. And so we can say they're both linear.

If we change it to be sorted, which is the first obvious improvement to make here, then get value gets a lot faster, because we can take advantage of binary search. I sort it by key, so ignoring the value, values are just satellite data. But stored in order of key, then all the A words, B words, C words, D words, what not, so walking it down the middle and seeing which half you look at, then looking at the quarter of there and the eight of there. It's going to very quickly narrow down on where it had to be if it was there or if it wasn't there at all. So we can implement that in logarithmic time, meaning that if you 1,000 entries, it takes ten comparisons to find it or agree that it's not there. If you double that it will take one more comparison. If you go to 2,000, negligibly faster, even numbers in the millions are still just a handful of comparisons to work it out, and say, to the twentieth for example is roughly a million; so 20 comparisons in or out, and so it didn't look at a million things a million times over.

However, that didn't improve add, why not? Why does keeping an assorted order not have the same boost for add as it did for get value? Can I do the same logarithmic search?

Student: You have to move all the elements –

Instructor (Julie Zelenski):Yeah, you can find the place fast, right, you can say returning the word apple it's like oh, okay. I can very quickly narrow in on where it needed to be. If it was there I could override it quickly, but in the case where it really needed to do an add, it's got to move them down. So the old add did all its work in the search and then just tacked it on the end if it didn't find it, or overrode the middle. But the new add also has to modify the array to put it in the right place, and putting it in the right place means shuffling to make space. So even though it found it quickly, it was unable to update that quickly, so it still ends up being linear in that case.

Now this actually not a terrible result, as it stands it's probably much more likely that you're going to load up the data structure once and then do a lot of searches on it, then infrequent additions. That's a pretty common usage pattern for a map. It's kind of like you build a dictionary, right, loading the dictionary once could be expensive, but then people get tons and tons of look ups of words later. But you don't change the definitions a lot

later or add a lot of new words. So this might actually be quite tolerable in many situations that the building of it was expensive, but it gave you fast search access. But really what you want to do is say, can we get both of those? So driving to say what we do with the editor buffer, you say, "Well, what if I just want both of those operations to be logarithmic or better, what can I do?"

So I'll leave you with kind of just a little bit of a taste of where we're heading. Just to think a little bit about what the link list does and doesn't buy us, right, that the link list gave us flexibility at the editing, the local editing site, but it doesn't give us fast searching. Well maybe we can take this link list and try to add searching to the idea of this flexibility to kind of get the best of both worlds. So that's what we're going to start on Monday, building a tree. So if you have time today, I'd love to have you come to Truman, and otherwise have a good weekend, and get your PQ work in.

[End of Audio]

Duration: 48 minutes

ProgrammingAbstractions-Lecture22

Instructor (Julie Zelenski):How are we doing? Spring apparently is here; just like California, one day it's winter, one day it's spring. Hopefully it's here to say. Hopefully thought it's not interfering with you getting all of your work done? How many people have a PQ implementation working, at least one of them? That's a whole bunch. How many have two of them? Even better. Any advice from those of you who have gotten or two of those things doing their job that you want to offer anybody who's yet to reach that state of enlightenment? Okay, so that's coming in on Wednesday and then your final assignment will go out on Wednesday. It will be due on Friday of Dead Week, so a little bit over a week to do that.

In term of late day planning, if you're the type of person who just feels like you would be hopefully cheated if you didn't get to use your late days, your plan was to make sure you used them. Let it be known that you will be able to use at least one on the last one. So it will be due on that Friday. If you are taking a late day, it will be due on Monday of exam week, probably not a good thing to actually slip into exam week, but if you're really in a bind you can use one, but no more than one. So if you want to make sure you use yours up, you may have to use some now. Not that I am recommending it. I think that there's a good point of pride to get done and have both of your late days in the bag. It means that you didn't hit any crisis, which is good.

We're going to talk about binary search trees and implementing the map abstraction using a binary search tree today. And the reading that goes along with that is chapter 13, all sorts of tree stuff. Anything administratively? No? So we had talked about doing it, we did this last time, we did an unsorted vector implementation of the map, we just did a little struct of the key value pair. And then we went through that both add and get value are running at linear time in that form because of the need to search; search to find a value to override it, and search to find a value to pull out the existing match to it. And if we went to the sorted form of that, we could get get value to be able to capitalize on that binary search property and be able to quickly find the key if it exists in there or to determine it doesn't exist.

But the add still suffered from the shuffling problem. We can quickly find, using the binary search to find where the new element would need to go. But if it doesn't already go there, it doesn't already exist. We're going to have to shuffle to make that space. So we have still a linear factor in there that we're working on getting rid of. And then I know at the bottom that none of them have any built in overhead per element. There may be some capacity, excess capacity in the vector that we're absorbing, but there's no four, ten, eight-byte chunk that every element is taking as a cost.

So we started to talk about this at the end on Friday, and we're going to go with this and see where this takes us. We had the idea of the vector, and we know what vector constraints are, right? This idea of the contiguous memory, and we were going to think about the link list for a minute to see if it was going to buy us anything. It tends to have an inverse relationship, that once you know where you want to insert something it's easy

to do that rearrangement because it's just a little bit of pointer rewiring that needs to happen in that situation. But it's hard to find a position insert, even if the link list is in sorted order, knowing where the S's are, or the M's are, or the B's are is still a matter of starting from the beginning; that your only access in the singly linked list is to start at the beginning and work your way down.

And so I'm going to take you through a little bit of a thought exercise about well, what if we try to do something about that? Right now having a pointer coming into Bashful and then subsequent ones going in sorted order isn't really very helpful. But if we were willing to try to rearrange our pointers to give us the access that we have in the sorted array form; for example, if I had a pointer to the middlemost element, in this case the Grumpy that's midway down the list, if I had that pointer and from there I could say, is the element I'm looking for ahead of Grumpy or behind Grumpy. So having Grumpy split my input in half. If I had that, right, then that would actually get me sort of moving towards that direction that I wanted to get, which is using the binary search property to facilitate throwing away half the data and look into just the half remaining that's interesting.

Well, if I had a pointer to Grumpy, that doesn't quite solve my problem. A pointer to Grumpy could get me halfway down the list, but then the pointer's coming in this way. If I inverted the pointers leading away from it, that's also kind of helping to facilitate what I'm trying to do. So if I could point toward the middle and then kind of move away, and then if I actually kind of applied the same strategy not just at the topmost level, but at every subsequent division down, from the half, to the quarter to the eight, to the sixteenth. And instead of having a pointer to the front of the list, what I really maintained was a pointer to the middlemost element, and then that element maintained pointers to the middle of the upper quarter and the middle of the lower quarter, and all the way down. Then I could pull up this list into something that we call a binary search tree.

So the way I would be storing this is having a pointer to a node, which is expected to the median or the middle, or somewhere right along the values being stored. And that it has two pointers, not just a next or previous or something like that, that are related to all the elements that precede this one in the ordering, and all the elements that follow it are in these two pointers. They call these subtrees coming off the smaller trees over here with three elements on this side, and three elements on that side. And that each of those nodes has recursively the same formulation; that it has two subtree hanging off of it that contain, given any particular node, that there's this binary search ordering here that says everything that's in that left subtree precedes this node in ordering. And everything that's in that right subtree follows it. And that's true for every node throughout the entire arrangement here.

So this is called a tree, and this particular form is actually called the binary search tree. I'm going to throw a little bit of vocabulary at you, because I'm going to start using these words, and I just want to get them out there early so that you understand what those words mean when I start using them. Is that each of the cells in here, like the cells that are allocated for a link list, are individually heap allocated, we're going to have pointers that

allow us to get back to them. We typically call those nodes; in terms of tree I'll call it a cell or node in the list kind of interchangeably. I'm more likely to use node when I'm talking about tree.

And a tree is really just a pointer to a node. The empty tree would be a pointer whose value is null, so pointing at nothing. A singleton tree would be a pointer to a single node that has left and right subtrees that are both empty or both null. So Bashful by itself is a singleton, a pointer to Bashful.

When we talk about subtrees, it just kind of means looking at the top; Grumpy has two subtrees, a left and a right, that themselves are just smaller forms of the same recursive structure. And then we would say that Grumpy is the parent of Doc and Sleepy. So those are the two children nodes, and we'll call them the left child and the right child. And the node that's at the very top, the one that our external pointer is coming into and then kind of leads the way, is called the root node. The first node of the tree where we first start our work is going to be called the root. And these nodes down here at the bottom that have empty subtrees are called leaf nodes. So a node that has no further subtrees, so by itself has null left and right trees, is called a leaf.

So that actually means the tree is kind of upside down if you look at it. That Grumpy's the root and these things are the leaves shows you that apparently most computer scientists don't get out very much, because they haven't noticed that trees actually grow the other direction. But just kind of go with it, and you'll be speaking tree-speak in no time. Anybody have any questions about what we've got basically set up here?

So there is a whole family of things that come under the heading of tree in computer science. We're going to look in particular at this one embodiment, the binary search tree. But before I get deep down in the details of that, I just want to back up for a second and just talk about what trees in general mean, and what kind of things in general are represented in manipulative trees. So that you have a sense of what is the broad notion of tree as a computer scientist would see it, is just that a tree is a recursive branching structure where there are nodes, which have pointers to subtrees. There may be one of those subtrees. There may be ten of those subtrees. There may be two; depending on the nature of the tree you're working on there may be a constraint on exactly how many children they have or some flexibility in the number of children they have depending on what's being modeled here.

There's always a single root node. There's a node that kind of sits at the top of this hierarchy that leads down to these things, and that to form a tree there has to be a reachable path from the root to every single node. So there aren't some nodes that exist way out here, and there aren't two different ways to get to the same node in a tree; that if the midterm is kept in the folder called exams, which is in the folder called cs106, which is in my home folder. There's not some other way that you can get to the midterm; the midterm lives exactly that part in the tree, and there's no circularity or loops in the structure.

And so a lot of things do get modeled tree, the one I've drawn here is just the file system. Think about how your files are maintained on a computer, there tends to be a structure of an outermost folder, which has inner folders, which have inner folders. And along the way there are these files, which you can think of as leaf nodes that have no further subtrees underneath them. And you can talk about, say, what is the size of all the folders in my home directory? Well it would kind of be recursively defined as summing over all the folders within it, how much storage is being used by those; which when it gets down to a leaf node can say, how much space does this file take up and kind of add those together.

Things like your family trees, right, sort of modeling genealogy. Things like the decomposition tree in your program; main calling, read file calling, set up boggle board calling, give instructions. There is a tree structure to that, about what pieces use what pieces and descendants from there. The ones that we're going to look at in particularly are the ones that are in the family of the binary trees.

So a binary tree has two children, exactly two children. There's a left and a right pointer coming out of each node. One or both of those can be null, but there can never be more than two children. And there is sort of space for recording two children, as opposed to recording things which are trinary, which have three possibilities for children, or where there may be some four or eight or some other number there. And in particular it's the search tree that's going to help us get the work done that we want, which is there's not just any old arrangement of what's on the left, and what's on the right side. That the nodes are going to be organized in here to facilitate binary search by keeping one half over on the left and one half over on the right, and that we'll know which half something goes in by virtue of looking at the key.

So let's take a look at a simple example of a binary search tree with numbers. So if I have a struct node that keeps track of an integer value and has two pointers, the left and the right, that point away from it. And in this case my root node is 57, and looking at 57 I can tell you that every value in this tree that is less than 57 will be in the left subtree. There will be no values that are 56 or lower that are on the right side. And that will be true at every step down the tree. So if I'm looking for something that actually gives me that immediate access we were using the vector's sorting property for. Which is if I'm looking for the number 70, and I start at the root node of 57, then I know it's not in the left subtree, so assuming that about half the nodes are pointed to over on that side, I've now discarded half of the input even from consideration.

If I start at the 70, and I say okay well if 57 must be over here, then again recursively apply the same search property here. It's like if I'm looking for 70, is it going to be before or after 79? It's going to be before, so in fact I throw away anything that leads away on the right subtree of 79, and work my way one level down. So each step in the recursive search is represented by sort of taking a step down in that search tree; working my way down from the root to the leaf nodes. And I either will find the value I'm looking for along the way, or I will fall off the bottom of that tree when I haven't been able to find it.

And that will tell me; oh it couldn't have been in the tree, because the only place it could have been is the one that followed the binary search property.

And so if I look for 70, I say is it greater or less than 62? It's greater. Is it greater or less than 71? It's less then. And then I work off there I get to the empty tree. Then I say well if there was a 60, that's where it was going to be, and it wasn't there so there much not be a 70 in this tree. So it is exactly designed to support one kind of search, and one kind of search only, which is this efficient divide and conquer binary search, work its way down.

The other kinds of things, let's say, that trees are used for may not need this kind of support, so they may not go out of their way to do this. But the one we're actually very interested in is because of maps doing a lot of additions into a sorted facility and searches in that sort of facility. If we can bottle it as a binary search tree, we're able to get the best of both worlds. To have that super fast logarithmic search, as well as have the flexibility of the pointer based structure to help us do an insert.

So if I'm ready to put a 70 into this tree, the same path that led to me to where it should be tells us where to put that new node in place. And if all I need to do is to wire that in place; and if all I need to do to wire that in place is to kind of create a new node in the heap and put the pointers coming in and out of it the way that I would in a link list, then I should be able to do both searches and inserts in time proportional to the height of the tree, which for a relatively balanced tree would be logarithmic. So it sounds like we can get everything we wanted here with a little bit of work.

So let me tell you – just do a little bit of playing with trees, just to kind of get used to thinking about how we do stuff on them, and then I'm going to go forward with implementing map using a binary search tree. Trees are very, very recursive. So for those of you who still feel like recursion is a little bit hazy, this is another great area to strengthen your thinking about recursive. In fact, I think a lot of people find operating on trees very, very natural. Because trees are so recursive, the idea that you write recursive algorithms to them is almost like you do it without even thinking, you don't have to think hard. You're like, you typically need to do something to your current node, and then recursively operate on your left and your right. And it's so natural that you don't even have stop and think hard about what that means.

So your base case tends to be getting to an empty tree and having a pointer that points to null, and in the other cases have some node operate on your children. Some of the simplest things are just traversals. If I tree structure and I'd like to do something to every node, go through it and print them all, go through it and write them to a file, go through and add them all up to determine the average score, I need these things. I'm going to need to do some processing that visits every node of the tree once and exactly once, and recursion is exactly the way to do that.

And there's a little bit of some choices, and when you're handling a particular part of a tree, are you handling the current node and then recurring on the left and right subtrees? Are you recurring first and then handling the nodes. So some of those things that we saw

in terms of the link list that change the order of the processing, but they don't change whether you visit everybody. It's just a matter of which one you want to do before.

I'm going to look at a couple of the simple traversals here, and I'll point out why that mattered. If we look at in order, so we'll consider that an in order traversal is for you to operate on your left subtree recursively, handle the current node right here, and then operate on your right subtree recursively. The base case, which is kind of hidden in here, is if T is null, we get to an empty subtree we don't do anything. So only in the nonempty case are we actually processing and making recursive calls. So if I start with my node 57 being the root of my tree, and I say print that tree starting from the root. It will print everything out of the left subtree, then print the 57, and then print everything out of the right subtree.

And given that gets applied everywhere down, the effect will be 57 says first print my left subtree. And 37 says, well print my left subtree. And 15 says, well first print my left subtree. And then seven says, first print my left subtree. Well, that ones empty, and so nothing gets printed. As the recursion unwinds, it comes back to the last most call. The last most call was seven. It says, okay, print seven and now print seven's right subtree. Again that's an empty, so nothing gets printed there. It will come back and then do the same thing with 15. I've done everything to the left of 15; let's print 15 and its right. And doing that kind of all the way throughout the tree means that we will get the numbers out in sorted order; all right, five, 17, seven, 15, 32, 57, 59, 62, 71, 79, 93. An in order traversal in a binary search tree rediscovers, visits all the nodes from smallest to largest.

Student:[Inaudible]

Instructor (Julie Zelenski):So duplicates, you just have to have a strategy. It turns out in the map, the duplicates overwrite. So any time you get a new value you print there. Typically you just need to decide, do they go to the left or do they go to the right? So that you know when you're equal to if you wanted to find another one of these, where would it go? In a lot of situations you just don't have duplicates, so it's not actually a big deal. You can't throw them arbitrarily on either side because you would end up having to search both sides to find them. So you want to just know that's it's less than or equal to on the left and greater than on the right.

So that's pretty neat, right, that that means that this for purposes of being able to iterate for something, if you were to walk the structure in order, then you can produce them back out in sorted order, which tends to be a convenient way to print the output. There are a lot of situations where you do want that browsable list in alphanumerical order. And the tree makes that easy to do.

When it comes time to delete the tree, I'm going to need to go through and delete each of those nodes individually, the same way I would on a link list to delete the whole structure. The order of traversal that's going to be most convenient for doing that is going to be to do it post order. Which is to say, when I'm looking at the root node 57, I want to delete my entire left subtree recursively, delete my entire right subtree recursively, and

only after both of those pieces have been cleaned up, delete the node we're on; so deleting down below to the left, down below to the right and then delete this one. If I did it in the other order, if I tried to delete in between in the in order or in the pre order case where I did my node and then to my subtrees, I'd be reaching into parts of my memory I've deleted. So we do want to do it in that order, so recursion coming back to visit you again.

Let's talk about how to make map work as a tree. So if you think about what maps hold, maps hold a key, which is string type; a value, which is client some sort of template parameter there. Now that I've built a structure that raps those two things with a pair and adds to it, right these two additional pointers to the left and to the right subtrees that point back to a recursive struct, the one data member that the map will have at the top is just the root pointer. The pointer to the node at the top of the tree, and then from there we'll traverse pointers to find our way to other ones. And we will organize that tree for the binary search property.

So using the key as the discriminate of what goes left and right, we'll compare each new key to the key at the root and decide is it going to go in the left subtree root or the right subtree root? And so on down to the matching position when we're doing a find, or to the place to insert if we're ready to put a new node in. So get value and add will have very similar structures, in terms of searching that tree from the root down to the leaf to find that match along the way, or to report where to put a new one in.

So let's look at the actual code; it has my struct, it has the strike key, it has the value type node, two pointers there, and then the one data member for the map itself here is the pointer to the root node. I also put a couple of helper number functions that I'm going to need. Searching the tree and entering a new node in the tree are going to require a little helper, and we're going to see why that is. That basically that the add and get value public number functions are going to be just wrappers that turn and pass the code off to these recursive helpers. And if these cases, the additional information that each of these is tracking is what node we're at as we work our way down the tree.

But the initial call to add is just going to say, well here's a key to value go put it in the right place. And that while we're doing the recursion we need to know where we are in the tree. So we're adding that extra parameter. What a lot of recursive code needs is just a little more housekeeping, and the housekeeping is what part of the tree we're currently searching or trying to enter into.

So this is the outer call for get value. So get value is really just a wrapper, it's going to call tree search. It's going to ask the starting value for root. It's the beginning point for the search, looking for a particular key. And what tree searches are going to return is a pointer to a node. If that node was not found, if there was no node that has a string key that matches the one that we're looking for, then we'll return null. Otherwise, we'll return the node. And so it either will error when it didn't find it or pull out the value that was out of that node structure.

Well, this part looks pretty okay. The template, right, there's always just a little bit of goo up there, right, seeing kind of the template header on the top and the use of the val type, and all this other stuff. It's going to get a little bit worse here. So just kind of hopefully take a deep breath and say, okay that part actually doesn't scare me too much. We're going to look at what happens when I write the tree search code, where it really actually goes off and does the search down the tree. And there's going to be a little bit more machinations required on this.

So let's just talk about it first, just what the code does. And then I'm going to come back and talk about some of the syntax that's required to make this work correctly. So the idea of tree search is, given a pointer to a particular subtree, it could be the root of the original tree or some smaller subtree further down, and a key that we're looking for, we're going to return a match when we find that node. So looking at the step here, where if the key that we have matches the key of the node we're looking at, then this is the node. This is the one we wanted.

Otherwise, right, we look at two cases. If the key is less than this node's key, then we're just going to return the result from searching in the left subtree. Otherwise it must be greater than, and we're going to search in the right subtree. So this case, right, duplicates aren't allowed, so there will be exactly one match, if at all. And when we find it we return. The other case that needs to be handled as part of base case is if what if we ever get to an empty tree. So we're looking for 32 and the last node we just passed was 35. We needed to be the left of it, and it turns out there was nothing down there, we have an empty tree. We'll hit this case where T is null, and we'll return null. That means there was never a match to that. There's no other place in the tree where that 32 could have been.

The binary search property completely dictates the series of left and right turns. There's nowhere else to look. So that's actually, really solving a lot of our – cleaning up our work for us, streamlining where we need to go; because the tree is organized through the mid point and the quarter point and then the eighth point to narrow down to where we could go. So as it is, this code works correctly. So do you have any questions about how it's thinking about doing its work? And then I'm going to show you why the syntax is a little bit wrong though.

Okay, so let's talk about this return type coming out of the function. So let's go back to this for a second. That node is defined as a private member of the map class. So node is actually not a global type. It does not exist outside the map class. Its real name is map val type:: node. That's its full name, is that. Now, inside the implementation of map, which is to say inside the class map, or inside member functions that we're defining later, you'll not that we can just kind of drop the big full name. The kind of my name is mister such and such, and such and such; it's like okay, you can just call it node, the short name internal to the class because it's inside that scope. And it's able to say, okay in this scope is there something called node? Okay there is. Okay, that's the one.

Outside of that scope, though, if you were trying to use that outside of the class or some other places, it's going to have a little bit more trouble with this idea of node by itself. It's

going to need to see that full name. Let me show you on this piece of code, that node here happens to be in the slight quirk of the way sequel plus defines things, that that use of node is not considered in class scope yet. That the compiler leads from left to write, so we'll look at the first one. It says, template type name val type, and so what the compiler sees so far is, I'm about to see something that's templated on val type. So it's a pattern for which val type is a placeholder in.

And then it sees val type as the return value, and it says, okay, that's the placeholder. And then it sees this map {val type}:: and it says, okay, this is within the – the thing that I'm defining here is within scope of the map class. And so it sees the map {val type}:: and that as soon as it crosses those double colons, from that point forward it's inside the map scope. And so, when I use things like tree search, or I use the name node, it knows what I'm talking about. It says, oh there's a tree search in map class. There's a node struct in map class, it's all good.

In the case of this one I've seen template type and val type, and at this point it doesn't know anything yet. And so it sees me using node, and it say, node I've never heard of node. I'm in the middle of making some template based on val type, but I haven't heard anything about node. Node isn't something that exists in the global name space. You must have made some tragic error. What it wants you to say is to give it the full name of map {value type}:: node. That the real full this is my name, and otherwise it will be quite confused.

If that weren't enough, we'd like to think that we were done with placating the C++ compiler. But it turns out there's one other little thing that has to happen here, which is there is an obscure reason why, even though I've given it the full name, and I feel like I've done my duty in describing this thing, that the compiler still is a little bit confused by this usage. And because the C++ language is just immensely complicated, there are a lot of funny interactions, there are a couple of situations it can get in to, where because it's trying hard to read left to right and see the things. Sometimes it needs to know some things in advance that it doesn't have all the information for. And it turns out in this case there's actually some obscure situation where it might not realize this is really a type. It might think it was more of a global constant that we actually have to use the type name key word again here on this one construction.

It is basically a hint to the compiler that says, the next thing coming up really is a type name. So given that you can't tell that, you'd think it would be obvious; in this case it doesn't seem like there's a lot for it to be confused at, but there's actually some other constructions that we can't rule out. So the compiler is a little bit confused in this situation, and it wants you to tell it in advance, and so I'll tell you that the basic rules for when you're going to need to use this is exactly in this and only this situation. And I'll tell you what the things are that are required for type name to become part of your repertoire.

You have a return type on a member function that is defined as part of a class that is a template, and that type that you are using is defined within the scope of that template. So it's trying to use something that's within a template class scope outside of that scope that

requires this type name. And the only place where we see things used out of their scope is on the return value. So this is kind of the one configuration that causes you to have to do this. It's really just, you know, a little bit of a C++ quirk. It doesn't change anything about what the code's doing or anything interesting about binary search trees, but it will come up in any situation where you have this helper member function that wants to return something that's templated. You have to placate the compiler there.

The error message when you don't do it, actually turns out to be very good in GCC and very terrible in visual studio. GCC says type name required here, and visual studio says something completely mystical. So it is something to be just a little bit careful about when you're trying to write that kind of code. Question?

Student: The asterisk?

Instructor (Julie Zelenski):The asterisk is just because it returns a pointer. In all the situations, I'm always returning a pointer to the node back there. So here's the pointer to the match I found or node when I didn't find one. All right, that's a little bit of grimy code. Let's keep going.

So how is it that add works? Well, it starts like get value, that the strategy we're going to use for inserting into the tree is the really simple one that say, well don't rearrange anything that's there. For example, if I were trying to put the number 54 into this tree, you could say – well you could imagine putting it at the root and moving things around. Imagine that your goal is to put it in with a minimal amount of rearrangement. So leaving all the existing nodes connected the way they are, we're going to follow the binary search path of the get value code to find the place where 54 would have been if we left everything else in tact.

Looking at 57 you could say, well it has to be to the left of 57. Looking to the 32 you say, it has to be to the right of 32. Here's where we get some empty tree, so we say, well that's the place where we'll put it. We're going to tack it off, kind of hang it off one of the empty trees along the front tier, the bottom of that tree. And so, we'll just walk our way down to where we fall off the tree, and put the new node in place there. So if I want the node 58 in, I'll say it has to go to the right; it has to go to the left; it has to go to the left; it has to go to the left; it will go right down there.

If I want the node 100 in, it has to go right, right, right, right, right. That just kind of following the path, leaving all the current pointers and nodes in place, kind of finding a new place. And there's exactly one that given any number in a current configuration and the goal of not rearranging anything, there's exactly one place to put that new node in there that will keep the binary search property in tact. So it actually does a lot of decision making for us. We don't have to make any hard decisions about where to go. It just all this left, right, left, right based on less than and greater than.

Student: What if the values are equal?

Instructor (Julie Zelenski):If the values are equal you have to have a plan, and it could be that you're using all the less than or equal to or greater than or equal to. You just have to have a plan. In terms of our map it turns out to equal to all the rights. So we handle it differently, which we don't every put a duplicate into the tree. So the add call is going to basically just be a wrapper that calls tree enter, starting from the root with the key and the value that then goes and does that work, walks its way down to the bottom to find the right place to stick the new one in.

Here is a very dense little piece of code that will put a node into a tree using the strategy that we've just talked about of, don't rearrange any of the pointers, just walk your way down to where you find an empty tree. And once you find that empty tree, replace the empty tree with a singleton node with the new key and value that you have. So let's not look at the base case first. Let's actually look at the recursive cases, because I think that they're a little bit easier to think about. Is if we every find a match, so we have a key coming in, and it matches the current node, then we just overwrite. So that's the way the map handles insertion of a duplicate value.

Now, in some other situation you might actually insert one. But in our case it turns out we never want a duplicate. We always want to take the existing value and replace it with a new one. If it matches, we're done. Otherwise, we need to decide whether we go left or right. Based on the key we're attempting to insert or the current node value, it either will go in the left of the subtree, left of the right, in which case we make exactly one recursive call either to the left or to the right depending on the relationship of key to the current key's value.

And then the one base case that everything eventually gets down to, that's going to create a new node, is when you have an empty tree, then that says that you have followed the path that has lead off the tree and fallen off the bottom. It says replace that with this new singleton node. So we make a new node out in the heap. We're assigning the key based on this. We're assigning the value based on the parameter coming in. And then we set its left and right null to do a new leaf node, right, it has left and right null subtrees. And the way this got wired in the tree, you don't see who is it that's now pointing to this tree, this new singleton node that I placed into here? You don't typically see that wire that's coming into it. Who is pointing to it is actually being done by T being passed by reference.

The node right above it, the left or the right of what's going to become the parent of the new node being entered, was passed by reference. It previously was null, and it got reassigned to point to this new node. So the kind of wiring in of that node is happening by the pass by reference of the pointer, which is a tricky thing to kind of watch and get your head around. Let's look at the code operating. Do you have a question?

Student: Yes. [Inaudible]. Can you explain that little step? Instructor:

Well this step is just something in the case of the map, if we every find an existing entity that has that key, and we overwrite. So in the map you said previously, Bob's phone number is this, and then you install a new phone number on Bob. If it finds a map to Bob

saying Bob was previously entered, it just replaces his value. So no nodes are created, no pointers are rearranged; it just commandeers that node and changes its value. That's the behaviors specified by map.

So if we have this right now, we've got some handful of fruits that are installed in our tree with some value. It's doesn't matter what the value is, so I didn't even bother to fill it in to confuse you. So this is what I have, papaya is my root node, and then the next level down has banana and peach, and the pear and what not. So root is pointing to papaya right now. When I make my call to insert a new value, let's say that the one that I want to put in the tree now is orange, the very first call to tree enter looks like this, T is the reference parameter that's coming in, and it refers to root. So the very first call to tree enter says please insert orange and its value into the tree who is pointed to by root.

So it actually has that by reference. It's going to need that by reference because it's actually planning on updating that when it finally installs the new node. So the first call to tree enter says, okay well is orange less than or greater than papaya. Orange precedes papaya in the alphabet, so okay, well call tree enter the second time passing the left subtree of the current value of tee. And so T was a reference to root right then, now T becomes a reference to the left subtree coming out of papaya. So it says, now okay, for the tree that we're talking about here, which is the one that points to the banana and below. That's the new root of that subtree is banana, it says, does orange go to the left or right of the root node banana?

And it goes to the right. So the third stack frame of tree enter now says, well the reference pointer is now looking at the right subtree coming out of banana. And it says, well does melon go to the left or the right of that? It goes to the right, so now on this call, right, the current value of T is going to be null. It is a reference, so a synonym for what was the right field coming out of the melon node, and in this is the case where it says, if orange was in the tree, then this is where it would be hanging off of. It would be coming off of the right subtree of melon. That part is null, and so that's our place. That's the place where we're going to take out the existing null, wipe over it, and put in a new pointer to the node orange that's out here.

And so that's how the thing got wired in. It's actually kind of subtle to see how that happened, because you don’t ever see a direct descendant, like T is left equals this, equals that new node; or T is right equals that new node. It happened by virtue of passing T right by reference into the calls further down the stack. That eventually, right when it hit the base case, it took what was previously a pointer to null and overwrote it with a pointer to this new cell, that then got attached into the tree.

Let's take a look at this piece of code. That by reference there turns out to be really, really important. If that T was just being passed as an ordinary node star, nodes would just get lost, dropped on the floor like crazy. For example, consider the very first time when you're passing in root and root is empty. If root is null, it would say, well if root is null and then it would change this local parameter T; T is some new node of these things. But if it really wasn't making permanent changes to the root, really wasn't making root pointer

of that new node, root would continue pointing to null, and that node would just get orphaned. And every subsequent one, the same thing would happen.

So in order to have this be a persistent change, the pointer that's coming in, either the root in the first case or T left or T right in subsequent calls, needed to have the ability to be permanently modified when we attach that new subtree.

That's really very tricky code, so I would be happy to try to answer questions or go through something to try to get your head around what it's supposed to be doing. If you could ask a question I could help you. Or you could just nod your head and say, I'm just totally fine.

Student: Could you use the tree search algorithm that you programmed before?

Instructor (Julie Zelenski):They're very close, right, and where did I put it? It's on this slide. The problem with this one right now is it does really stop and return the null. So I could actually unify them, but it would take a little more work, and at some point you say, wow I could unify this at the expense of making this other piece of code a little more complicated. So in this case I chose to keep them separate. It's not impossible to unify them, but you end up having to point to where the node is or a null where you would insert a new node. Tree search doesn't care about, but tree enter does. It sort of would modify it to fit the needs of both at the expense of making it a little more awkward on both sides. Question here.

Student:[Inaudible] the number example. If you were inserting 54 where you said, if you called print, wouldn't it print out of order?

Instructor (Julie Zelenski):So 54 would go left of 57, right of 32, so it'd go over here. But the way in order traversals work is says, print everything in my left subtree, print everything in my left subtree, left subtree. So it would kind of print seven as it came back five, 32, and then it would print 54 before it got to the 57. So an in order traversal of a binary search tree, as long as the binary search tree is properly maintained, will print them in sorted order. It's almost like, don't even think too hard about it, if everything in my left subtree is smaller, and everything in my right subtree is greater than me, if I print all of those I have to come out in sorted order. If the property is correctly maintained in the tree, then that by definition is a sorted way to pass through all the numbers.

So if at any point an in order traversal is hitting numbers out of order, it would mean that the tree wasn't properly organized. So let's talk about how that works, there are a couple of other operations on the tree, things like doing the size, which would count the number of members. Which you could do by just doing a traversal, or you might do by just caching a number that you updated as you added and moved nodes. And the contains keys, that's basically just a tree search that determines whether it found a node or not. The rest of the implementation is not interest. There's also deleting, but I didn't show you that.

The space use, how does this relate to the other known implementations that we have for math? It adds certainly some space overhead. We're going to have two pointers, a left and a right off every entry. So every entry is guaranteed to have an excess 8-byte capacity relative to the data being stored. It does actually add those tightly to the allocation. So like a link list, it allocates nodes on a per entry basis, so there's actually not a lot of reserve capacity that's waiting around unused the way it might been in a vector situation. And the performance of it is that it's based on the height of the tree, that both the add and the get value are doing a traversal starting from the root and working their way down to the bottom of the tree, either finding what it wanted along the way or falling off the bottom of the tree when it wasn't found or when it needs to be inserting a new node.

That height of the tree, you're expecting it to be logarithmic. And so if I look at, for example, the values, I've got seven values here, two, eight, 14, 15, 20, and 21. Those seven, eight values can be inserted in a tree to get very different results. There's not one binary search tree that represents those values uniquely. Depending on what order you manage to insert them you'll have different things. For example, given the way we're doing it, whatever node is inserted first will be the root, and we never move the root. We leave the root along. So if you look at this as a historical artifact and say, well they entered a bunch of numbers, and this is the tree I got. I can tell you for sure that 15 was the very first number that they entered. There's no other possibility, 15 got entered first.

The next number that got inserted was one of eight or 20, I don't know which. I can tell you, for example, that eight got inserted before 14. In terms of a level-by-level arrangement, the ones on level one were always inserted before the ones on level two. But one possibility I said here was well maybe 15 went in first, then eight, and then two, and then 20, and then 21, and then 14, and then 18. There are some other rearrangements that are also possible that would create the exact same tree. And there are other variations that would produce a different, but still valid, binary search tree.

So with these seven nodes, right, this tree is height three, and is what's considered perfectly balanced. Half the nodes are on the left and the right of each tree all the way through it, leading to that just beautifully balanced result we're hoping for where both our operations, both add and get value, will be logarithmic. So if I have 1,000 nodes inserted, that height tree should be about ten. And if everything kind of went well and we got kind of went well, and we got an even split all the way around, then it would take ten comparisons to find something in that tree or determine it wasn't there' and also to insert, which was the thing we were having trouble with at the vector case, that because that search leads us right to the place where it needs to go, and the inserting because of the flexibility of the pointer base structure is very easy, we can make both that insert and add operate logarithmically.

Let's think about some cases that aren't so good. I take those same seven numbers and I insert them not quite so perfectly, and I get a not quite so even result. So on this side I can tell you that 20 was the first one inserted. It was the root and the root doesn't move, but then subsequent ones was inserted an eight, and then a 21, and then an 18, and then a 14, and then a 15, and then it went back and got that two, the one over there. Another

possibility for how it got inserted 18, 14, 15, and so on. These are not quite as balanced as the last one, but they're not terribly unbalanced. They're like one or two more than the perfectly balanced case. So at logarithmic plus a little constant still doesn't look that bad.

Okay, how bad can it really get? It can get really bad. I can take those seven numbers and insert them in the worst case, and produce a completely degenerate tree where, for example, if I insert them in sorted order or in reverse sorted order, I really have a link list. I don't have a tree at all. It goes two, 18, 14, 15, 20, 21, that's still a valid binary search tree. If you print it in order, it still comes out in sorted order. It still can be searched using a binary search strategy. It's just because the way it's been divided up here, it's not buying us a lot. So if I'm looking for the number 22, I'll say, well is it to the left or the right of two? It's to the right. Is it to the left or right of eight? Oh, it's to the right. Is it to the left or right, you know like all the way down, so finding 22 looking at every single one of the current entries in the tree, before I could conclude, as I fell off the bottom, that it wasn't there.

Similarly kind of over there; it didn't buy us a lot. It has a lot to do with how we inserted it as to how good a balance we're going to get. And there are some cases that will produce just drastically unbalanced things that require linear searches, totally linear to walk that way down to find something.

There's a couple of other degenerate trees, just to mention it's not only the sorted or reverse sorted. There's also these other kind of saw tooth inputs, they're called, where you just happen to at any given point to have inserted the max or the min of the elements that remain, so it keeps kind of dividing the input into one and N-1. So having the largest and the smallest, and the largest and the smallest, and then subsequent smallest down there can also produce a linear line. It's not quite all just down the left or down the right, but it does mean that every single node has an empty left or right subtree is what's considered the degenerate case here. And both have the same property where half of the pointers off of each cell are going nowhere.

There is an interesting relationship here between these worst-case inputs for transversion and quick sort, and they're actually kind of exactly one to one. That what made quick sort deteriorate was the element of the input, the max or the min of what remain. And that's exactly the same case for transversion, max or min of what remains means that you are not dividing your input into two partitions of roughly equal size. You're dividing them into the one, N-1, which is getting you nowhere.

What do you get to do about it? You have to decide first of all if it's a problem worth solving. So that's what the first one say. Sometimes you just say that it will sometimes happen in some inputs that I consider rare enough that it will degenerate. I don't think they're important enough or common enough to make a big deal out of. I don't think that applies here, because it turns out that getting your data in sorted order is probably not unlikely. That somebody might be refilling a map with data they read from a file that they wrote out in sorted order last time. So saying that if it is coming in sorted, then I'm probably going to tolerate that, is probably not going to work.

And there's two main strategies then, if you've agreed to take action about it, about what you're going to do. The first one is to wait until the problem gets bad enough that you decide to do something about it. So you don't do any rebalancing as you go. You just let stuff go, left, right, left, right, whatever's happening. And you try to keep track of the height of your tree, and you realize when the height of your tree seems sufficiently out of whack with your expected logarithmic height, to do something about it. Then you just fix the whole tree. You can clean the whole thing up. Pull them all out into a vector, rearrange them, put them back in, so the middlemost node is the root all the way down. And kind of create the perfectly balanced tree, and then keep going and wait for it to get out of whack, and fix it again; kind of wait and see.

The other strategy, and this happens to be a more common one in practice, is you just constantly do a little bit of fixing up all the time. Whenever you notice that the tree is looking just a little bit lopsided, you let it get a little bit; usually you'll let it be out of balance by one. So you might have height four on that side and height three on that side. But as soon as it starts to get to be three and five, you do a little shuffle to bring them back to both four. And it involves just a little bit of work all the time to where you just don't let the tree turn into a real mess. But there's a little bit of extra fixing up being done all the time.

And so the particular kind of tree that's discussed in the text, which you can read about, but we won't cover as part of our core material, is called the ADL tree. It's interesting to read about it to see what does it take, what kind of process is required to monitor this and do something about it. And so if we have that in place, we can get it to where the binary search tree part of this is logarithmic on both inserting new values and searching for values, which is that Holy Grail of efficiency boundary. You can say, yeah logarithmic is something we can tolerate by virtue of adding 8 bytes of pointer, potentially some balance factor, and some fanciness to guarantee that performance all over, but gets us somewhere that will scale very, very well for thousands and millions of entries, right, logarithmic is still a very, very fast function.

So it creates a map that could really actually go the distance in scale. So, that's your talk about binary search trees. We'll come back and talk about hashing in graphs and stuff. I will see you on Wednesday.

[End of Audio]

Duration: 51 minutes

ProgrammingAbstractions-Lecture23

Instructor (Julie Zelenski):Okay, we got volume and everything. All right, here we go. Lots of you think your learning things that aren't gonna actually someday help you in your future career. Let's say you plan to be president. You think, "Should I take 106b; should I not take 106b?" Watch this interview and you will see. [Video]

We have questions, and we ask our candidates questions, and this one is from Larry Schwimmer. What – you guys think I'm kidding; it's right here. What is the most efficient way to sort a million 32-bit integers?

Well – I'm sorry. Maybe we should – that's not a – I think the bubble sort would be the wrong way to go. Come on; who told him this?

Instructor (Julie Zelenski):There you go. So, apparently – the question was actually by a former student of mine, in fact, Larry Schwimmer. So apparently, he's been prepped well. As part of going on the campaign trail, not only do you have to know all your policy statements, you also have to have your N2 and login sorts kept straight in your mind, so – just so you know.

Okay. So let's talk about where we are administratively. PQ coming in, big milestone, right? So PQ, one of the rights of passage in terms of getting the pointers, and the link lists, and all that goopy, low level stuff, sort of, straight in your head. I expect this was, sort of, a bigger investment on your part, but let's do a little poll, and we'll find out how much joy there was in Mudville on this. How many people finished in under ten hours? I don't think anybody could/should. My gosh – well, okay. There you go – a few, a few. Ten to fifteen? Still, I think a very impressive, kind of, efficiency to getting that done in that. Fifteen to twenty? Clinton, you're not raising your hand; what are you telling me? Uh, oh.

Student:I had a late day.

Instructor (Julie Zelenski):Late day – oh, okay. All right. More than 20? Okay, and this more than 20 people may still be in this late day camp. It is one of the heftier pieces of work that we've given you, so we, kind of, are expecting that, but hopefully, at this point, on of the things that should've been the mid-quarter eval is, right, was people saying, "Yeah, this pointer – " and a couple things they said pointers, still mystical, link lists feeling a little shaky on that, as well as, sort of, Big O not totally solid in their head, and I do think that with the sorting lab and the PQ work on the Big O, hopefully that last category has come together but also the pointers and link lists, right, you're starting to get real hands-on practice which helps to make the concepts, kind of, make more sense than they did when they're just me yakking about it. Your last assignment is pathfinder, which is gonna go out today. I'm gonna actually demo it just a little bit because I think this is an awesome, kind of, capstone to finish off the quarter with, and I just love playing with it. So I want to show it to you just to get you psyched for what's going on here. So what pathfinder is doing is a graph search problem, but it's, kind of, just a version of

something that MapQuest, and Google Maps, and all those little GPS devices are doing is path finding, right? Given you're at this position, and you want to get at this address, what are the routes that take you from here to there, and which of those is the best on given the constraints you're working under? So maybe it's taken advantage of faster roads, freeway roads, or roads that are shorter, roads that are in better repair, roads where there's less traffic. In this case, we can, kind of, simplify this problem down to, well, if we just had the distance information and all this connecting data about which places you can be at – you can be at the golf course, and you can bike over to FroSoCo, or you could walk Robley Field, or over to the Gates Center, and things like that. It's, like, if you were at one of these places, what path could you take that would get you from one place to the other, in this case, minimizing overall distance. So it seems like a little bit of magic, right? Like, you go to those sites, you type in these things, and it finds you things, but, in fact, actually, at this point, you're capable of writing a program that can do exactly what those things do, and the key thing is having an efficient algorithm and a big pile of data to work on. We have some smaller datas that make it easier to test on but the same approach applies in the large as well as the small. And so, in the case of the program, what it does is I can pick a place that I am at. I can say, oh, I'm over here at Escondido at my kid's elementary school, and I need to get to Gates, and it'll tell me that I need to cut through campus going down by the east residences and then across through the quad, cutting off the corner and heading to Gates is my shortest path that connects up those two things. And so all the other things, kind of, out there, you know, it considered as part of its strategy here but was able to zero in on a path, in this case, is about a third of a mile between here and there, and then it actually tells me a little bit about how it did its work. It actually looked at 112 different paths before it concluded that was the best one, which is a fairly small number given all the paths that are out there. So it's actually doing some very efficient pruning to decide which ones to look at first. And so I can do that again with some more things, doing other ones. I'm over here at the stadium, and I'd like to get to Robley Field. It turns out, given the paths that it has, it doesn't have a direct path, so it has to, kind of, cut around and come back in that case, but kind of, cutting through the front of the Quad, and coming around by Lagunita. So a very neat, neat program, right, that makes use of, kind of, everything we've seen this quarter. So lots of uses for the ADTs, the set, and the vector, and stack, and queue have some roles to play. We'll see the priority queue come back. So the priority queue that you've just finished, right, is gonna have a role to play here. We're gonna take it and turn it into a template form as part of the algorithms that you're implementing need a priority queue, so you'll turn that integer-based priority queue into a template one, and then draw and manage this grafting. A little bit of graphics work, some of the stuff you saw in the chaos and the maze assignment's come back, a little bit of drawing to do here to get the visual results up. The other thing it does, actually it does another secondary graph calculation, which is the computation of what's called the Minimal Spanning Tree, which is if you took apart all the roads, you know, or paths that exist on campus, and you were interested in reinstalling them but choosing the minimal set that will connect up all the locations you have on campus so that you can get somewhere from everywhere, but without any redundancies or cycles, and choosing the smallest of your options. So if you were trying to wire up, let's say, internet across the campus, and you needed to hit each of those nodes, where are the places you could lay those cables to connect everyone onto the campus network using the

minimal amount of wire? And that's the second algorithm that is what you're implementing in this project, so a nice thing to know how to do. So we're gonna talk about why does that – what does that work; what is it doing? Let's talk about graphs. Okay. So graphs and graphs algorithms are the topics for today which relate to that final assignment, and that final assignment's coming in on Friday of dead week, so that's a week and a few days from today. You can use one late day on it if you really need to which extends it into Monday of exam week, probably not a good choice though if you have early in the week exams. So finishing it on time, kind of, gets it behind you and lets you focus on the exam studying. Our exam is at the very end of exam week, right, so that Friday afternoon 12:00 to 3:00. I know being at the end, kind of, interferes with the rushing off to have a good time over spring break, but it also gives you extra time to study. So maybe you can think of it as, kind of, a tradeoff, and then I will put out a practice exam either later this week or next week that gives you and idea of, kind of, what to expect in terms of going into that exam. The last section of the reader that we're gonna be looking at on Friday is about hashing and hash tables as an alternate implementation for the map, and that's covered in Chapter 11, and that's actually, kind of, the final material coming out of the reader. What I'll be doing next week is just looking at some advanced topics, trying to, kind of, pull together some compare and contrast in the data structures and algorithms we've seen. So I'll talk a little bit about, kind of, the big picture view of how we make choices among these things, and where we go from here. All right, any administrative questions? So let me give you some examples of things that are graphs, just to get you thinking about the data structure that we're gonna be working with and playing around with today. Sometimes you think of graphs of being those, like, bar graphs, right, your graphing histograms. This is actually a different kind of graph. The computer scientist's form of graph is just a generalized, recursive data structure that, kind of, starts with something that looks a little bit like a tree and extends it to this notion that there are any number of nodes, in this case, represented as the cities in this airline route map, and then there are connections between them, sometimes called edges or arcs that wire up the connections that in the case, for example, of this graph are showing you which cities have connecting flights. You have a direct flight that takes you from Las Vegas to Phoenix, right, one from Phoenix to Oklahoma City. There is not a direct flight from Phoenix to Kansas City, but there's a path by which you could get there by taking a sequence of connecting flights, right, going from Phoenix, to Oklahoma City, to Kansas City is one way to get there. And so the entire, kind of, map of airline routes forms a graph of the city, and there are a lot of interesting questions that having represented that data you might want to answer about finding flights that are cheap, or meet your time specification, or get you where you want to go, and that this data would be, kind of, primary in solving that problem. Another one, right, another idea of something that's represented as a graph is something that would support the notion of word ladders, those little puzzles where you're given the word "chord," and you want to change it into the word "worm," and you have these rules that say, well, you can change one letter at a time. And so the connections in the case of the word letter have to do with words that are one character different from their neighbors, have direct connections or arcs between them. Each of the nodes themselves represents a word, and then paths between that represent word ladders. That if you can go from here through a sequence of steps directly connecting your way, each of those stepping stones is a word in the chain that makes a

word ladder. So things you might want to solve there is what's the shortest word ladder? How many different word ladders – how many different ways can I go from this word to that one could be answered by searching around in a space like this. Any kind of prerequisite structure, such as the one for the major that might say you need to take this class before you take that class, and this class before that class, right, forms a structure that also fits into the main graph. In this case, there's a connection to those arcs. That the prerequisite is such that you take 106a, and it leads into 106b, and that connection doesn't really go in the reverse, right? You don't start in 106b and move backwards. Whereas in the case, for example, of the word ladder, all of these arcs are what we called undirected, right? You can traverse from wood to word and back again, right? They're equally visitable or neighbors, in this case. So there may be a situation where you have directed arcs where they really imply a one-way directionality through the paths. Another one, lots and lots of arrows that tell you about the lifecycle of Stanford relationships all captured on one slide in, sort of, the flowchart. So flowcharts have a lot of things about directionality. Like, you're in this state, and you can move to this state, so that's your neighboring state, and the arcs in there connect those things. And so, as you will see, like, you get to start over here and, like, be single and love it, and then you can, kind of, flirt with various degrees of seriousness, and then eventually there evolves a trombone and serenade, which, you know, you can't go back to flirting aimlessly after the trombone and serenade; there's no arc that direction. When somebody gets the trombone out, you know, you take them seriously, or you completely blow them off. There's the [inaudible] there. And then there's dating, and then there's two-phase commit, which is a very important part of any computer scientist's education, and then, potentially, other tracks of children and what not. But it does show you, kind of, places you can go, right? It turns out you actually don't get to go from flirt aimlessly to have baby, so write that down – not allowed in this graph. And then it turns out some things that you've already seen are actually graphs in disguise. I didn't tell you when I gave you the maze problem and its solution, the solving and building of a maze, that, in fact, what you're working on though is something that is a graph. That a perfect maze, right, so we started with this fully disconnected maze. If you remember how the Aldis/Broder algorithm worked, it's like you have all these nodes that have no connections between them, and then you went bopping around breaking down walls, which is effectively connecting up the nodes to each other, and we kept doing that until actually all of the nodes were connected. So, in effect, we were building a graph out of this big grid of cells by knocking down those walls to build the connections, and when we were done, then we made it so that it was possible to trace paths starting from that lower corner and working our way through the neighbors all the way over to there. So in the case of this, if you imagine that each of these cells was broken out in this little thing, there'd be these neighboring connections where this guy has this neighbor but no other ones because actually it has no other directions you can move from there, but some of them, for example like this one, has a full set of four neighbors you can reach depending on which walls are intact or not. And so the creation of that was creating a graph out of a set of disconnected nodes, and then solving it is searching that graph for a path. Here's one that is a little horrifying that came from the newspaper, which is social networking, kind of, a very big [inaudible] of, like, on the net who do you know, and who do they know, and how do those connections, like, linked in maybe help you get a job. This is one that was actually based on the liaisons,

let's say, of a group of high school students, right, and it turns out there was a lot of more interconnected in that graph than I would expect, or appreciate, or approve of but interesting to, kind of, look at. So let's talk, like, concretely. How are we gonna make stuff work on graphs? Graphs turn out to actually – by showing you that, I'm trying to get you this idea that a lot of things are really just graphs in disguise. That a lot of the problems you may want to solve, if you can represent it as a graph, then things you learn about how to manipulate graphs will help you to solve that kind of problem. So the basic idea of a graph, it is a recursive data structure based on the node where nodes have connections to other nodes. Okay. So that's, kind of, like a tree, but it actually is more freeform than a tree, right? In a tree, right, there's that single root node that has, kind of, a special place, and there's this restriction about the connections that there's a single path from every other node in the tree starting from the root that leads to it. So you never can get there by multiple ways, and there's no cycles, and it's all connected, and things like that. So, in terms of a graph, it just doesn't place those restrictions. It says there's a node. It has any number of pointers to other nodes. It potentially could even have zero pointers to other's nodes, so it could be on a little island of its own out here that has no incoming or outgoing flights, right? You have to swim if you want to get there.

And then there's also not a rule that says that you can't have multiple ways to get to the same node. So if you're at B, and you're interested in getting to C, you could take the direction connection here, but you could also take a longer path that bops through A and hits to C, and so there's more than one way to get there, which would not be true in a tree structure where it has that constraint, but that adds some interesting challenges to solving problems in the graphs because there actually are so many different ways you can get to the same place. We have to be careful avoiding getting into circularities where we're doing that, and also not redundantly doing work we've already done before because we've visited it previously. There can be cycles where you can completely come around. I this one, actually, doesn't have a cycle in it. Nope, it doesn't, but if I add it – I could add a link, for example, from C back to B, and then there would be this place you could just, kind of, keep rolling around in the B, A, C area. So we call each of those guys, the circles here, the nodes, right? The connection between them arcs. We will talk about the arcs as being directed and undirected in different situations where we imply that there's a directionality; you can travel the arc only one-way, or in cases where it's undirected, it means it goes both ways. We'll talk about two nodes being connected, meaning there is a direct connection, an arc, between them, and if they are not directly connected, there may possibly be a path between them. A sequence of arcs forms a path that you can follow to get from one node to another, and then cycles just meaning a path that revisits one of the nodes that was previously traveled on that path. So I've got a little terminology. Let's talk a little bit about how we're gonna represent it.

So before we start making stuff happen on them, it's, like, well, how does this – in C++, right, what's the best way, or what are the reasonable ways you might represent this kind of connected structure? So if I have this four-node graph, A, B, C, D over where with the connections that are shown with direction, in this case, I have directed arcs. That one way to think of it is that what a graph really is is a set of nodes and a set of arcs, and by set, I mean, sort of, just the generalization. There might be a vector of arcs; there might be a

vector of nodes, whatever, but some collection of arcs, some collection of nodes, and that the nodes might have information about what location they represent, or what word, or what entity, you know, in a social network that they are, and then the arcs would show who is directly connected to whom. And so one way to manage that, right, is just to have two independent collections – a collection of nodes and a collection of arcs, and that when I need to know things about connections, I'll have to, kind of, use both in tandem. So if I want to know if A is connected to B, I might actually have to just walk through the entire set of arcs trying to find one that connects A and B as its endpoints. If I want to see if C is connected to B, same thing, just walk down the whole set. That's probably the simplest thing to do, right, is to just, kind of, have them just be maintained as the two things that are used in conjunction. More likely, what you're gonna want to do is provide some access that when at a current node, you're currently trying to do some processing from the Node B, it would be convenient if you could easily access those nodes that emanate or outgo from the B node. That's typically the thing you're trying to do is at B try to visit its neighbors, and rather that having to, kind of, pluck them out of the entire collection, it might be handy if they were already, kind of, stored in such a way to make it easy for you to get to that. So the two ways that try to represent adjacency in a more efficient way are the adjacency list and the adjacency matrix. So in an adjacency list representation, we have a way of associating with each node, probably just by storing it in the node structure itself, those arcs that outgo from there. So, in this case, A has as direct connection to the B and C Nodes, and so I would have that information stored with the A Node, which is my vector or set of outgoing connections. Similarly, B has one outgoing connection, which goes to D. C has an outgoing connection to D, and then D has outgoing connections that lead back to A and to B. So at a particular node I could say, well, who's my neighbors? It would be easily accessible and not have to be retrieved, or searched, or filtered.

Another way of doing that is to realize that, in a sense, what we have is this, kind of, end by end grid where each node is potentially connected to the other N -1 nodes in that graph. And so if I just build a 2x2 matrix that's labeled with all the node names on this side and all the node names across the top, that the intersection of this says is there an arc that leads away from A and ends at B? Yes, there is. Is there one from A to C? And in the places where there is not an existing connection, a self loop or just a connection that doesn't exist, right, I have an empty slot. So maybe these would be Booleans true and false or something, or maybe there'd be some more information here about the distance or other associated arc properties for that particular connection.

In this case, it's actually very easy then to actually do the really quick search of I'm at A and I want to know if I can get to B, then it's really just a matter of reaching right into the slot in constant time in that matrix to pull it out, and so this one involves a lot of searching to find things. This one involves, perhaps, a little bit of searching. If I want to know if A is connected to B, I might have to look through all its outgoing connections, right? This one gives me that direct access, but the tradeoffs here has to do with where the space starts building up, right? A full NxN matrix could be very big, and sometimes we're allocating space as though there are a full set of connections between all the nodes,

every node connected to every other, and in the cases where a lot of those things are false, there's a lot of capacity that we have set aside that we're not really tapping into.

And so in the case of a graph that we call dense, where there's a lot of connections, it will use a lot of that capacity, and it might make sense in a graph that's more sparse, where there's a few connections. So you have thousands of nodes, but maybe only three or four on average are connected to one. Then having allocated 1,000 slots of which to only fill in three might become rather space inefficient. You might prefer an adjacency list representation that can get you to the ones you're actually using versus the other, but in terms of the adjacency matrices is very fast, right? A lot of space thrown at this gives you this immediate 0,1 access to know who your neighbors are. So we're gonna use the adjacency list, kind of, strategy here for the code we're gonna do today, and it's, kind of, a good compromise between the two alternatives that we've seen there. So I have a struct node. It has some information for the node. Given that I'm not being very specific about what the graph is, I'm just gonna, kind of, leave that unspecified. It might be that it's the city name. It might be that it's a word. It might be that it's a person in a social network, you know, some information that represents what this node is an entity, and then there's gonna be a set of arcs or a set of nodes it's connected to, and so I'm using vector here. I could be using set. I could be using a raw array, or a link list, or any of these things. I need to use some unbounded collection though because there's no guarantee that they will be 0, or 1, or 2 the way there is in a link list or a tree where there's a specified number of outgoing links. There can be any number of them, so I'm just leaving a variable size collection here to do that work for me. The graph itself, so I have this idea of all the nodes in the graph, right, would each get a new node structure and then be wired up to the other nodes that they are connected to, but then when I'm trying to operate on that graph, I can't just take one pointer and say here's a pointer to some node in the graph and say this is – and from here you have accessed everything. In the way that a tree, the only thing we need to keep track of is the pointer to the root, or a link list, the only thing we keep a pointer to, typically, is that frontmost cell, and from there, we can reach everything else. There is no special head or root cell in a graph. A graph is this being, kind of, a crazy collection without a lot of rules about how they are connected. In fact, it's not even guaranteed that they totally are connected. That I can't guarantee that if I had a pointer, for example, to C, right, I may or may not be able to reach all the nodes from there. If I had a pointer to A, right, A can get to C and B, and then also down to D, but, for example, there could just be an E node over here that was only connected to C or not connected to anything at all for that matter, connected to F in their own little island.

That there is no way to identify some special node from which you could reach all the nodes. There isn't a special root node, and you can't even just arbitrarily pick one to be the root because actually there's no guarantee that it will be connected to all of the others. So it would not give you access to the full entirety of your graph if you just picked on and said I want anything reachable from here, it might not get you the whole thing. So, typically, what you need to really operate on is the entire collection of node stars. You'll have a set or a vector that contains all of the nodes that are in the graph, and then within that, there's a bunch of other connections that are being made on the graph connectivity that you're also exploring.

So let's make it a little bit more concrete and talk a little bit about what really is a node; what really is an arc? There may be, actually, be some more information I need to store than just a node is connected to other nodes. So in the case of a list or a tree, the next field and the left and the right field are really just pointers for the purpose of organization. That's the pointer to the next cell. There's no data that is really associated with that link itself.

That's not true in the case of a graph. That often what the links that connect up the nodes actually do have a real role to play in terms of being data. It may be that what they tell you is what road you're taking here, or how long this path is, or how much this flight costs, or what time this flight leaves, or how full this flight already is, or who knows, you know, just other information that is more than just this nodes connected to that one. It's, like, how is it connected; what is it connected by, and what are the details of how that connection is being represented?

So it's likely that what you really want is not just pointers to other nodes but information about that actual link stored with an arc structure. So, typically, we're gonna have an arc which has information about that arc, how long, how much it costs, you know, what flight number it is, that will have pointers to the start and end node that it is connecting. The node then has a collection of arcs, and those arcs are expected to, in the adjacency list form, will all be arcs that start at this node. So their start node is equal to the one that's holding onto to them, and then they have an end pointer that points to the node at the other end of the arc.

Well, this gets us into a little C++ bind because I've described these data structures in a way that they both depend on each other; that an arc has pointers to the start and the end node. A node has a collection of arcs, which is probably a vector or a set of arc pointers, and so I'm starting to define the structure for arc, and then I want to talk about node in it, and I think, oh, okay, well, then I better define node first because C++ always likes to see things in order. Remember, it always wants to see A, and then B if B uses A, and so in terms of how your functions and data structures work, you've always gotten this idea like, well, go do the one first.

Well, if I say, okay, arc's gonna need node. I start to write arc, and I say, oh, I need to talk about node. Well, I better put node up here, right? But then when I'm starting to write node, I start talking about arc, and they have a circular reference to each other, right? They both want to depend on the other one before we've gotten around to telling the compiler about it. One of them has to go first, right? What are we gonna do?

What we need to do is use the C++ forward reference as a little hint to the compiler that we are gonna be defining a Node T structure in a little bit; we're gonna get to it, but first we're gonna define the Arc T structure that uses that Node T, and then we're gonna get around to it. So we're gonna give it, like, a little heads up that says there later will be an Act 2 of this play. We're gonna introduce a character called Node T. So you can now start talking about Node T in some simple ways based on your agreement to the compiler that you plan on telling it more about Node T later.

So the struct Node T says there will be a struct called Node T. Okay, the compiler says, okay, I'll write that down. You start defining struct Arc T, and it says, oh, I see you have these pointers to the node T. Okay. Well, you told me there'd be a struct like that; I guess that'll work out. And now you told it what struct Node T is. It's, like, oh, I have a vector with these pointers to arc, and then it says, okay, now I see the whole picture, and it all worked out. So it's just a little bit of a requirement of how the C++ compiler likes to see stuff in order without ever having to go backwards to check anything out again.

Okay. So I got myself some node structures, got some arc structures; they're all working together. I'm gonna try to do some traversals. Try to do some operations that work their way around the graph. So tree traversals, right, the pre, and post, and in order are ways that you start at the root, and you visit all the nodes on the tree, a very common need to, kind of, process, to delete nodes, to print the nodes, to whatnot. So we might want to do the same thing on graphs. I've got a graph out there, and I'd like to go exploring it. Maybe I just want to print, and if I'm in the Gates building, what are all the places that I can reach from the Gates building? Where can I go to; what options do I have?

What I'm gonna do is a traversal starting at a node and, kind of, working my way outward to the visible, reachable nodes that I can follow those paths to get there. We're gonna look at two different ways to do this. One is Depth-first and one is Breadth-first, and I'll show you both algorithms, and they will visit the same nodes; when all is done and said, they will visit all the reachable nodes starting from a position, but they will visit them in different order. So just like the pre/post in order tree traversals, they visit all the same notes. It's just a matter of when they get around to doing it is how we distinguish depth and breadth-first traversals. Because we have this graph structure that has this loose connectivity and the possibility of cycles and multiple paths to the same node, we are gonna have to be a little bit careful at how we do our work to make sure that we don't end up getting stuck in some infinite loop when we keep going around the ABC cycle in a particular graph. That we need to realize when we're revisiting something we've seen before, and then not trigger, kind of, an exploration we've already done. So let's look at depth-first first. This is probably the simpler of the two. They're both, actually, pretty simple. Depth-first traversal uses recursion to do its work, and the idea is that you pick a starting node – let's say I want to start at Gates, and that maybe what I'm trying to find is all the reachable nodes from Gates. If I don't have a starting node, then I can just pick one arbitrarily from the graph because there is actually no root node of special status, and the idea is to go deep. That in the case of, for example, a maze, it might be that I choose to go north, and then I just keep going north. I just go north, north, north, north, north until I run into a dead wall, and then I say, oh, I have to go east, and I go east, east, east, east, east. The idea is to just go as far away from the original state as I can, just keep going outward, outward, outward, outward, outward, outward, outward, and eventually I'm gonna run into some dead end, and that dead end could be I get to a node that has no outgoing arcs, or it could be that I get to a node that whose only arcs come back to places I've already been, so it cycles back on itself, in which case, that also is really a dead end.

So you go deep. You go north, and you see as far as you can get, and only after you've, kind of, fully explored everything reachable from starting in the north direction do you

backtrack, unmake that decision, and say, well, how about not north; how about east, right? And then find everything you can get to from the east, and after, kind of, going all through that, come back and try south, and so on. So all of the options you have exhaustively getting through all your neighbors in this, kind of, go-deep strategy. Well, the important thing is you realize if you've got this recursion going, so what you're actually doing, basically, is saying I'm gonna step to my neighbor to the right, and I'm gonna explore all the things, so do a depth-first search from there to find all the places you can reach, and that one says, well, I'm gonna step through this spot and go to one of my neighbors. And so we can't do that infinitely without getting into trouble. We need to have a base case that stops that recursion, and the two cases that I've talked about – one is just when you have no neighbors left to go. So when you get to a node that actually, kind of, is a dead end, or that neighbor only has visited neighbors that surround it.

So a very simple little piece of code that depends on recursion doing its job. In this case, we need to know which nodes we have visited, right? We're gonna have to have some kind of strategy for saying I have been here before. In this case, I'm using just a set because that's an easy thing to do. I make a set of node pointers, and I update it each time I see a new node; I add it to that set, and if I ever get to a node who is already previously visited from that, there's no reason to do any work from there; we can immediately stop.

So starting that step would be empty. If the visit already contains the node that we're currently trying to visit, then there's nothing to do. Otherwise, we go ahead and add it, and there's a little node here that says we'll do something would occur. What am I trying to do here? Am I trying to print those nodes? Am I trying to draw pictures on those nodes, highlight those nodes, you know, something I'm doing with them. I don't know what it is, but this is the structure for the depth-first search here, and then for all of the neighbors, using the vector of outgoing connections, then I run a depth-first search on each of those neighbors passing that visited set by reference through the whole operation. So I will always be adding to and modifying this one set, so I will be able to know where I am and where I still need to go.

So if we trace this guy in action – if I start arbitrarily from A, I say I'd like to begin with A. Then the depth-first search is gonna pick a neighbor, and let's say I happen to just work over my neighbors in alphabetical order. It would say okay, well, I've picked B. Let's go to everywhere we can get to from B. So A gets marked as visited. I move onto B; I say, well, B, search everywhere you can to from B, and B says okay, well, I need to pick a neighbor, how about I pick C? And says and now, having visited C, go everywhere you can get to from C. Well, C has no neighbors, right, no outgoing connections whatsoever. So C says, well, okay, everywhere you can get to from C is C, you know, I'm done.

That will cause it to backtrack to this place, to B, and B says okay, I explored everywhere I can get to from C; what other neighbors do I have? B has no other neighbors. So, in fact, B backtracks all the way back to A. So now A says okay, well, I went everywhere I can get to from B, right? Let me look at my next neighbor. My next neighbor is C. Ah, but C, already visited, so there's no reason to go crazy on that thing. So, in fact, I can

immediately see that I've visited that, so I don't have any further path to look at there. I'll hit D then as the next one in sequence.

So at this point, I've done everything reachable from the B arm, everything reachable from the C arm, and now I'm starting on the D arm. I said okay, where can I get to from D? Well, I can get to E. All right, where can I get to from E? E goes to F, so the idea is you think of it going deep. It's going as far away as it can, as deep as possible, before it, kind of, unwinds getting to the F that has no neighbors and saying okay. Well, how about – where can I get to also from E? I can get to G. From G where can I get to? Nowhere, so we, kind of, unwind the D arm, and then we will eventually get back to and hit the H.

So the order they actually were hit in this case happened to be alphabetical. I assigned them such that that would come out that way. That's not really a property of what depth-first search does, but think of it as like it's going as deep as possible, as far away as possible, and so it makes a choice – it's pretty much like the recursive backtrackers that we've seen. In fact, they make a choice, commit to it, and just move forward on it, never, kind of, giving a backwards glance, and only if the whole process bottoms out does it come back and start unmaking decisions and try alternatives, eventually exploring everything reachable from A when all is done and said.

So if you, for example, use that on the maze, right, the depth-first search on a maze is very much the same strategy we're talking about here. Instead of the way we did it was actually breadth-first search, if you remember back to that assignment, the depth-first search alternative is actually doing it the – just go deep, make a choice, go with it, run all the way until it bottoms out, and only then back up and try some other alternatives.

The breadth-first traversal, gonna hit the same nodes but just gonna hit them in a different way. If my goal were to actually hit all of them, then either of them is gonna be a fine strategy. There may be some reason why I'm hoping to stop the search early, and that actually might dictate why I would prefer which ones to look at first. The breadth-first traversal is going to explore things in terms of distance, in this case, expressed in terms of number of hops away from the starting node. So it's gonna look at everything that's one hop away, and then go back and look at everything two hops away, and then three hops away.

And so if the goal, for example, was to find a node in a shortest path connection between it, then breadth-first search, by looking at all the ones one hop away, if it was a one hop path, it'll find it, and if it's a two hop path, it'll find it on that next round, and then the three hop path, and there might be paths that are 10, and 20, and 100 steps long, but if I find it earlier, I won't go looking at them. It actually might prove to be a more efficient way to get to the shortest solution as opposed to depth-first search which go off and try all these deep paths that may not ever end up where I want it to be before it eventually made its way to the short solution I was wanting to find.

So the thing is we have the starting node. We're gonna visit all the immediate neighbors. So, basically, a loop over the immediate one, and then while we're doing that, we're

gonna actually, kind of, be gathering up that next generation. Sometimes that's called the Frontier, sort of, advancing the frontier of the nodes to explore next, so all the nodes that are two hops away, and then while we're processing the ones that are two hops away, we're gonna be gathering the frontier that is three hops away. And so at each stage we'll be, kind of, processing a generation but while assembling the generation N+1 in case we need to go further to find those things. We'll use an auxiliary data structure during this. That management of the frontier, keeping track of the nodes that are, kind of, staged for the subsequent iterations, the queue is a perfect data structure for that. If I put the initial neighbors into a queue, as I dequeue them, if I put their neighbors on the back of the queue, so enqueue them behind all the one hop neighbors, then if I do that with code, as I'm processing Generation 1 off the front of the queue, I'll be stacking Generation 2 in the back of the queue, and then when all of Generation 1 is exhausted, I'll be processing Generation 2, but, kind of, enqueuing Generation 3 behind it. Same deal that we had with depth-first search though is we will have to be careful about cycles will pull past that when there's more than one possibility of how we get to something, we don't want to, kind of, keep going around that cycle or do a lot of redundant work visiting nodes that we've already seen. So I have the same idea of keeping track of a set that knows what we've previously visited, and the initial node gets enqueued just, kind of, by itself, just like Generation 0 gets put into the queue, and then while the queue is not empty, so while there's still neighbors I haven't yet visited, I dequeue the frontmost one, if it hasn't already been visited on some previous iteration, then I mark it as visited, and then I enqueue all of its children or neighbors at the back of the queue.

And so, as of right now, I'm processing Generation 0, that means all of Generation 1, so the neighbors that are one hop away, get placed in the queue, and then the next iteration coming out here will pull off a Generation 1, and then enqueue all of these two hop neighbors. Come back and other Generation 1's will get pulled off, enqueue all their two hop neighbors, and so the two hop generation will, kind of, be built up behind, and then once the last of the one hop generation gets managed, start processing the Generation 2 and so on.

And so this will keep going while the queue has some contents in there. So that means some neighbors we have connections to that we haven't yet had a chance to explore, and either we will find out that they have all been visited, so this will end when all of the nodes that are in the queue have been visited or there are no more neighbors to enqueue. So we've gotten to those dead ends or those cycles that mean we've reach everything that was reachable from that start node.

And so I've traced this guy to see it doing its work. So if I start again with A, and, again, assuming that I'm gonna process my nodes in alphabetical order when I have children, so I will through and I will enqueue, so the queue right now has just node A in it. I pull it off. I say have I visited? No, I have not. So then I mark it as visited, and now for each of its neighboring nodes, B, C, D, H, I'm gonna put those into the queue, and so then they're loaded up, and then I come back around, I say do I still have stuff in the queue? I do, so let's take out the first thing that was put in; it was B. Okay, and I'll say okay, well, where can you get to from B? You can get to C, so I'm gonna put C in the queue.

Now, C is actually already in the queue, but we're gonna pull it out earlier in terms it'll get handled by the visiting status later. So, okay, we'll put C, D, and E in and then put C behind it. So right now, we'll have C, D, H, and C again. It'll find the C that's in the front there. It'll say where can you get to from C? Nowhere, so no additional Generation 2's are added by that one. I'll look at D, and I'll say okay, well, what Generation 2's do we have from D? It says, well, I can get to E. All right, so put E on the back of the queue. Look at H; where can H get to? H can get to G, so go ahead and put G on the back of the queue.

So now, it comes back around to the things that are two hops away. The first one it's gonna pull off is C. It's gonna say, well, C is already visited, so there's no need to redundantly visit that or do anything with that. It passes over C. It finds the E that's behind that, and then it finds the G that was behind the H. The E also enqueued the F that was four hops, and the very last one, right, output will be that F that was only reachable by taking three hops away from the thing.

So this one's, kind of, working radially. The idea that I'm looking at all the things that I can get to, kind of, in one hop are my first generation, then two hops, then three hops, and so it's like throwing a stone in the water and watching it ripple out that the visiting is managed in terms of, kind of, growing length of path, and number of hops that it took to get there.

So this is the strategy that you used for maze solving, if you remember. That what you were tracking was the queue of paths where the path was AB, or ADC, or ADE, and that they grew step wise. That, kind of, each iteration through the traversal there where you're saying, okay, well, I've seen all the paths that are one hop. Now I've seen all the ones of two, now the ones three, and four, and five, and when you keep doing that, eventually you get to the hop, you know, 75, which leads you all the way to the goal, and since you have looked at all the paths that were 74 and shorter, the first path that you dequeue that leads to your goal must be the best or the shortest overall because of the order you processed them.

That's not true in depth-first search, right? That depth-first search might find a much longer and more circuitous path that led to the goal when there is a shorter one that would be eventually found if I had a graph that looked, you know, had this long path, let's say, and so this is goal node, let's say, and this is the start node that there could be a two hop path, kind of, off to this angle, but if this was the first one explored in the depth-first, right, it could eventually, kind of, work its way all the way around through this eight hop path, and say, okay, well, I found it, right?

It would be the first one we found, but there's no guarantee that would be the shortest in depth-first search, right? It would just happen to be the one that based on its arbitrary choice of which neighbors to pursue it happened to get to that there could be a shorter one that would be found later in the traversal. That's not true in the breadth-first search strategy because it will be looking at this, and then these two, and then these two, and,

kind of, working its way outward down those edges. So as soon as we get to one that hits the goal, we know we have found the shortest we can have. Question?

Student: In that case, you could return, okay, we need two jumps to get to the goal, like, as far winning the goal.

Instructor (Julie Zelenski):Yeah.

Student: How do we remember which path led us there?

Instructor (Julie Zelenski):So if you were really doing that, you know, in terms of depth-first search, what you'd probably be tracking is what's the path? Probably a stack that said here's the stack that led to that path. Let me hold onto this, right, and I will compare it to these alternatives. So I'll try it on Neighbor 1, get the stack back from that; it's the best. Try it on Neighbor 2, get that stack back, see which one is smaller and use that as the better choice, and so, eventually, when I tried it out this side, I'd get the stack back that was just two, and I could say, yeah, that was better than this ten step path I had over here, so I will prefer that. So just using your standard, kind of, backtracking with using your return values, incorporating them, and deciding which was better.

And so in this case, the depth-first search can't really be pruned very easily because it has to, kind of, explore the whole thing to know that this short little path right over here happened to be just arbitrarily examined last, whereas, the breadth-first search actually does, kind of, because it's prioritizing by order of length. If the goal was shortest path, it will find it sooner and be able to stop when it found it rather than look at these longer paths. So it won't end up – if there were hundreds of other paths over here, breadth-first search won't actually have to ever reach them or explore them.

So there are a lot of things that actually end up being graph search in disguise. That's why a, kind of, a graph is a great data structure to, kind of, get to in 106b because there are lots of problems that, in the end, if you can model them using the same setup of nodes, and arcs, and traversals, that you can answer interesting questions about that data by applying the same graph search techniques. So as long as you want to know things like, well, which nodes are reachable from this node at all? And then that just means, okay, I'm here and I want to get to these places; what places could I get to?

Or knowing what courses I could take that would lead me to a CS degree, it's, kind of, like well, looking for the ones that lead to having all your graduation requirements satisfied, and each of those arcs could be a course you could take, right, how close did that get you to the goal? How can I get to the place I'd like to be, or what are the places that this course satisfies requirements that lead to? Knowing things about connectivity is, kind of, useful in things like the, kind of, bacon numbers or erdish numbers where people want to know, well, how close am I to greatness by – have I been in a movie with somebody who was in a movie, who was in a movie with Kevin Bacon? And, for me, it turns out zero; I have been in no movies.

But have you written a paper with someone who then connects through a chain to some other famous person, right? Tells you about your, kind of, social network distance or something, or I'm linked in. You're trying to get a new job, and you want to find somebody to introduce you to the head of this cool company. It's, like, figuring who you know who knows them, right? It's finding paths through a social network graph.

Finding out things like are there paths with cycles? What's the shortest path through the cycle, longest path through the cycle often tells you about things about redundancies in that structure that, for example, when you're building a network infrastructure, sometimes you do have cycles in it because you actually want to have multiple ways to get to things in case there's a breakdown. Like if there's a freeway accident, you want to have another way to get around it, but you are interested in, maybe, trying to keep your cost down is trying to find out where are the right places to put in those redundancies, those cycles that give you the best benefit with the least cost.

Trying to find things like a continuous path that visits all the nodes once and exactly once, especially doing so efficiently, picking a short overall path for that. We talked a little bit about that in terms of what's called the traveling salesman problem. You are trying to get from – you have ten cities to visit on your book tour. You don't want to spend all your lifetime on a plane. What's the path of crisscrossing the country that gets you to all ten of those with the least time painfully spent on an airline? And so those are just graph search problems. You have these nodes. You have these arcs. How can you traverse them and visit and come up with a good solution.

There's other problems I think that are kind of neat. Word letters, we used to give out an assignment on word letters. We didn't this time, but I thought it was an interesting problem to think about. The idea of how it is you can transform one letter at a time from one word to another is very much a graph problem behind the scenes, and that largely a breadth-first traversal is exactly what you're looking for there. How can I transform this with the shortest number of steps is, kind of, working radially out from the starting word.

Your boggle board turns out, really, is just a graph. If you think of having this sequence of letters that you're trying to work through, that really each of these letters can actually just be thought of as a node, and then the connections it has are to its eight neighbors, and that finding words in there, especially finding paths, starting from this node that lead away, that don't revisit, so, in fact, doing breadth-first or depth-first search. You most likely did depth-first search when you were writing the implementation of boggle, but that's, kind of, a deep search on I found an A. I found a P. I found a P, and, kind of, keeps going as long as that path looks viable.

So, in this case, that the arc information we're using is having a sub of those letters, do they form a prefix that could be leading somewhere good, and then the, kind of, recursion bottoming out when we have run ourselves into a corner where we have no unvisited nodes that neighbor us, or only nodes that would extend to a prefix that no longer forms any viable words as a search problem. So it feels like everything you've done this quarter

has actually been a graph in disguise; I just didn't tell you. The maze problem is very much a graph search problem, building a graph and searching it.

Another one that I think is, kind of, neat to think about is, kind of, related to the idea of word letters is that often when you mistype a word, a word processor will suggest to you here's a word that you might have meant instead. You type a word that it doesn't have in its lexicon, and it says, well, that could just be a word I don't know, but it also could mean that you actually mistyped something, and so what you want to look at is having neighboring nodes, sort of, diagramming a graph of all the words you know of and then the, kind of, permutations, the tiny little twiddles to that word that are neighbors to it so that you could describe a set of suggestions as being those things that are within a certain hop distance from the original mistyped word.

And so you could say, yeah, things that have two letters different are probably close enough, but things that have three or more letters different aren't actually likely to be the mistyped word, and then that, kind of, leads to supporting things like wildcard searches where you're trying to remember whether it's E-R-A-T or A-R-T-E at the end of desperate. Those things could be modeled as, kind of, like they search through a graph space where you're modeling those letters, and then there's, like, a branching where you try all of the letters coming out of D-E-S-P, seeing if any of them lead to an R-A-T-E.

So the last thing I wanted to just bring up, and this is the one that's going into this week's assignment. I'm just gonna actually show it to you to, kind of, get you thinking about it because it's actually the problem at hand that you're gonna be solving is that the one of where I'm interested in finding the shortest path, but I'm gonna add in this new idea that all hops are not created equal. That a flight from San Francisco to Seattle is a shorter distance than the one that goes from San Francisco to Boston, and if I'm interested in getting to someplace, it's not just that I would like to take a direct flight. I'd also like to take the set of flights that involves the least, kind of, going out of my way, and going through other cities, and other directions.

And so in this case, a breadth-first search is a little bit too simplistic. That if I just look at all the places I can get to in one hop, those are not really equally weighted in terms of my goal of minimizing my total path distance, but if I, instead, prioritize – prioritize in your mind, immediately conjure up images of the priority queue, to look at paths that are, if I'm trying to minimize total distance, the shortest path, even if they represent several hops, if they're still shorter – if I have three flights, and each are 100 miles, then they represent a flight of 300 total miles, and that's still preferred to one that actually is 2,000 miles or 7,000 miles going to Asia alternatively.

And so if I worked on a breadth-first that's been modified by a total distance, that it would be a way to work out the shortest path in terms of total weight from San Francisco to my destination by a modified breadth-first, kind of, weighted strategy, and that is exactly what you're gonna be doing in this assignment. I'm not gonna give away too much by, kind of, delving too much in it, but the handout does a pretty good job of talking you through what you're doing, but it's a, kind of, neat way to think about that's

how the GPS, the MapQuest, and things like that are working is you're at a site; you have a goal, and you have this information that you're trying to guide that search, and you're gonna use heuristics about appears to be the, kind of, optimal path to, kind of, pick that, and go with it, and see where it leads you. So that will be your task for this week. What we'll talk about on Friday is caching.

[End of Audio]

Duration: 53 minutes

ProgrammingAbstractions-Lecture24

Instructor (Julie Zelenski):What are we doing today? We're gonna talk about hashing. Hashing's one of the coolest things you're ever gonna learn in 106b, so it's a good day to be here and learn something really neat, kinda clever, inventive idea for how you do search and retrieval in a very efficient way. That's covered in chapter 11 of the text and that actually is the last bit of the text, right, is chapter 11. So what we're gonna do next week is we're gonna try to pull some stuff together.

I'm gonna do one advanced lecture on some neat data structure I think that's fun to explore together and then try to kinda pull together a little bit of kind of what the big picture is now that you've seen all these things from the client side and the low level side kinda wrap it all up for you and get you ready to think about the final exam. Final exam is actually two weeks from today, so today is Friday, and our exam is at the Friday of exam week, so it's in the near future, but of course between now and then is pathfinders, your big task to get done before that. I'll be at the Terman café today after class, but this is your last chance. So if you haven't made it yet, this is the time to come because actually next Friday I am gonna be at a conference, so it is time for coffee and snacks, right today. Please come. Okay, so I'm gonna put us back to where we were a lecture ago, two lectures back to Monday, where we had talked about ways we might implement the map abstraction, right, the key value pairing to allow for quick search and retrieval and we had looked at vector and had just done a little thought experiment about what sorting the vector would buy us, right, and that would give us the ability to do searches faster because we could use binary search, right.

But in the case of add, right, we'd still be slow because the need be kinda shuffled or rearranged in the continuous memory and then that's what motivated our introduction to trees there, the binary search tree, which is designed exactly to enable that kind of searching, right, that logarithmic divide and conquer, divide in half and work your way down and then because of the flexibility of the pointer base structure there, right, we can also do the insert in the same logarithmic time because we're no longer constrained by that contiguous memory that the array fact data structures force us to. And so, at that point what we have seen this logarithmic and both get valued at and so on the PQ, right, which you were just finishing, you'll notice that logarithmic is pretty fast. In fact logarithmic for most values of n that you would normally run into in an ordinary use, right, it's un-measurable, right. So those of you who got your heap up and running and correctly working logarithmic and time was discovering that you can put 10,000, 20,000, 50,000, 100,000 things in it and just never be able to measure what it took to NQ or DQ from those structures because logarithmic is very, very fast.

So that said, that's actually a fine point to say, we have a great math notation, we don't have to look further, but in fact I'm gonna push it a little bit today and we're gonna actually getting it down to where we can do both those operations in constant time, which is to say that no matter how large the table gets, right, we expect we'll be able to insert and retrieve with the exact same time required kind of for a 10,000 table as a million entry table which is pretty neat. To note that as we start to get to these more complicated

data structures, one thing we also need to keep in mind, right, is that there are other things of trade offs, right, in adding that complexity of the code that starting with something that's simple and a little slower performer but that's easy to write might actually solve the problem we have at hand. So maybe that we don't really wanna go to one of these fancier things because we don't really need it to be that much faster because the BST, right, adds a bunch of memory, right, so left and right pointers for every cell, which is an 8 byte overhead on top of the entry itself.

Now we have to deal with pointers, dynamic memories, all this allocation, deallocation, right, we've got a lot of recursion going on in there and so just opportunities for error and stuff to creep into and if we take the steps to provide tree balancing, which we only kind of grazed the surface of. We just mentioned the need for it. It actually adds quite a lot of complication to the code to do the rotations or the rebalancing that from a simple binary search tree. So in fact the whole package, right, of building a really guaranteed log n performing binary search tree is actually a pretty big investment of human capital that you have to kinda balance against the future hit off of it running faster. I'm gonna show this strategy today that actually is kind of simple to write. It gets us solve one case and doesn't have the degenerate looming that the binary search tree did. So, let me kinda just give you a theoretical concept to think about how you do stuff. You have a big dictionary. You've got, you know, like a many thousand page dictionary, right, and you want to look up a word. You certainly don't do linear search. You're about to look up the word xylophone and you're actually not going to start from the A's and kind of page your way through, you know, 2,000 pages to get to the X's.

You don't even – like we do binary search. You can do binary searches pretty fast, right? It turns out like starting at the M's to get something that's in X, right? You tend to know about where X is and often dictionaries have little tabs along the pages that give you an idea of kinda where the breaks for certain words are. Maybe it says Xa is here and Xe is there or something, that lets you kinda more quickly just illuminate all the surrounding noise in the dictionary, kinda get straight to the right region where the X words are and then you can do a simple search from there, maybe a little bit of a linear or binary search to kind of zero in on the page where the word is you wanna find is. This is the idea behind this concept called a hash table or based on the algorithmic technique of hashing is that rather than try to figure out how to kinda keep this very large collection sorted or organized in a way that you can kind of jump around within it, it says, "Well how about we maintain a bunch of different collections." We break up the data set into little tiny pieces and if we can tell, given a key, like which piece of the data set it's supposed to be in and we'll call those pieces buckets. I'm gonna put these in different buckets.

So if I had a whole bunch of students I might put all the freshmen in this bucket and the sophomores in this bucket or something like that and then when I look for a student I would say, "Well which class are you?" You say, "Freshmen." I'd look in the freshmen bucket. Okay. You know, the idea in a simple case would be like, "Okay, well I – it has some criteria by which I can divide you into these buckets that don't overlap and everyone has a unique bucket to go in and if I can make up enough of those categorizations, have as many of those buckets around to divide you up into, then I can

hopefully keep the set that lands in that bucket pretty small, so that I don't have a smaller of students. So maybe I did it by the first letter of your last name. I could say, "Well everybody whose last name begins with W goes here and everybody who's B goes here and so on." Well, when somebody comes to me then I don't have to look through the B's. You know, and if you have a class of 200 students and you have 26 letters of the alphabet then you'd think that if it were evenly divided across those letters, it's not going to be to be, it turns out in practice, but as a thought experiment there'd be about 4 people in a bucket and it'd be pretty fast to just look through the bucket and find the one who I was looking for.

Okay, so let's take this idea. This idea of, like, there's going to be some strategy for how we divide into buckets and we call that strategy the hash function. That, given a key, and a buckets and we'll just label them with numbers. Zero to the number of buckets minus one. So let's say I have 99 buckets, I've got zero to 98 that I will take the key and the key in this case is the string key that we're using in the map, and I have a hash function which given a key is input produces a number as output in the range zero to B minus one. And that is what's called the hash value for that key and that tells me which bucket to look at, either find a match for that if I'm trying to do a get value or where to place a new entry if I'm trying to add into the table. So that's the basic idea. I'm gonna talk a little bit about hash functions. In terms of writing a really good hash function is actually pretty complicated and so we're gonna think about it in sort of a – kind of a big picture perspective and think about what qualities a hash function has to have and then I'm actually not gonna go through proving to you about what mathematically constitutes a good hash function.

I'm just gonna give you some intuitive sense of what kinda things matter when designing a hash function. So it's given a key that maps it to a number. Typically it's gonna say, you know, zero to the number of buckets minus one. Often that means that somehow it's gonna compute something and then use the mod operation to bring it back in range. So it's gonna take your number and then say, "Okay. Well, if you have five buckets to divide it in, I will give you a number zero to four." It's very important that it be stable, that you put the key in, you get a number out. If you put that same key in later you should get the same number out. So it has to be sort of reliable guaranteed mapping. It can't be random. That said, you want it to almost look as though it's random in terms of how it maps things to buckets, that you don't want it to have a real systematic plan where it puts a whole bunch of things in bucket one and very few things in bucket two or somehow always uses the even buckets and not the odds ones or something like that. So you wanna have a strategy that's gonna take those keys and use your whole bucket range in an equal manner. So that if you put 100 keys in and you have 100 buckets you're hoping that, on average, each bucket will have one thing. That you won't have these clusters where 25 things are in one bucket, ten here and then a whole bunch of buckets are empty.

So you want this kinda divided across the range. So given that we have string input, likely what we're gonna be using is the characters in the string as part of the – how can I take these characters and somehow kinda jungle them up in a way to sort of move them around in these buckets in a way that's reliable so that I get that same string back, I get

the same one. But that doesn't actually produce a lot of artifacts of similar strings, let's say mapping to the same place, hopefully spreading them around. So for example, some really simple things you could use, you could just take the first letter. So I talked about this sorta with last names. If I was trying to divide up my students, right, I could divide them by the first letter of their last name. That would use exactly 26 buckets, no more, no less, and if I happen to have 26 buckets then at least there's some chance it would get some coverage across the range. There is likely to be clustering, right, some names are a lot more common or letters are a lot more commonly used than others. So it would tend to produce a little bit of an artifact, but it would be reliable and it would use 26 buckets.

If I had 100 buckets, then this would be kind of a crummy function because in fact it would use the first 26 buckets and none of the other ones, right, if it was just taking that letter and deciding. So, you know, in some situations this would get me a little bit of the thing I wanted. Things like using the length of the word. A very, very bad clustering function here, right because there'd be a bunch of strings that are, like, six characters, seven characters, ten characters, but very few that are one or two, and very few that are more than ten, or something. You'd have this very tight little cluster in the middle, right, and would not use a large bucket range effectively at all. More likely what you're gonna try to use is kind of more of the information about the word, the key that you have. Instead of just a single letter or the count of letters, it's trying to use kinda all the data you have at hand, which is to say all the letters, and so one way to do it would be to take the ASCII values for the letters, so A is 97, you know, B is 98, is you would take a letter, a word like atoms and you'd add 96 plus 99 plus 96, you know, add them together and then you'd get some sum of them, 500, 600, whatever it is, depending on how many letters there are and then use mod to back it back into – let's say your range was, you had 100 buckets, right, so you'd want it to be zero to 99, so you'd take that thing and then mod it by 100, then take the last two digits of it. So it'd be, okay, the sum to 524 then it goes into bucket 24.

So if you did that, right, and so if act is the word that you have, right, you'd add these codes together and let's say this is 97 and then this is 99 and then that's, you know, a hundred and something and so we end up with – I'm making this number up, right – 283, then we might mod to the last two digit and we'll say this one goes into bucket 83. And if we have, you know, dog, maybe that adds to 267 and we would mod that down to 67. So this actually tries to use, kinda, all the information at hand, all the letters that we have and bring them down, and so if you figure that across the space of all the numbers that are being added here you are just as likely to get 210 as you are to get 299, you know that any of the subsequent numbers in between, then hopefully you'd get kind of a spread across your buckets. There is one thing this clusters on. Does anybody see what words cluster in this system? That you'd think of as maybe not being related, but would land in the same bucket?

Student: To rearrange some numbers.

Student: Yes.

Instructor (Julie Zelenski):Any anagram, right, so you take cat which is just a ranagram – anagram of act, right, it's gonna add to the same thing, right, and so it would be a little bit of clustering in that respect. So anagrams come down to the same place. And so the words that you expect to, kinda like, act and cat to you seem like different words, they are gonna collide in this system and that might be a little bit of something we worry about.

So one way you can actually kind of avoid that in the way that the hash function we're going to eventually use will do, is it'll do this thing where not only does it add the letters together, but it also multiplies the letter by a very large prime number that helps to kinda say, "Well the large prime number times C plus the large prime number squared times A plus the large prime number cubed times C and then it just allows the, each position of the letter actually is also then encoded in the number that comes out and so something that's positionally different won't necessarily land in the same bucket without some other factors kinda coming into play. So a little bit of trying to just take the information and jumble it up a little bit. So let me show you a little bit about how this looks. Actually I'm gonna tell you about the collisions and then I'll show you the data structure. So certainly I said there's gonna be this problem where things collide, that 83 and 83 in this simple system of adding the codes will come out the same. So it can't be that each of these buckets holds exactly one thing.

The hashing, right, although we may try to get those buckets as small as possible we have to account for the possibility that two things will collide, that their hash function, even though they were different keys, will land in the same bucket based on how I've jumbled it up and so we're trying to avoid that, but when it does happen we also have to have a strategy for it. So we have to have a way of saying, "Well there can be more than one thing in a bucket." Or some other way of when there's a collision, deciding where to put the one that was placed in the table second. And in the strategy, we're gonna use the one called chaning, and the chaning basically says, "Well if once we have something in a bucket like ACT, if we hash something and it comes to the same bucket, we'll just tack it onto that bucket, in this case using a link list." So each of these buckets is actually gonna be a link list of the entries and so the entire table will be a vector or array of those. So let me show you, a little live action. So this is a hash table that has seven buckets. In this case number 0 through 6 are assigned to each one. They all start empty, and so these are all empty link lists illustrated by the null, and then the hash function is shown here like a black box.

You don't actually know how the workings of hash function is, but you know what it's job is to do, which is you give it a key and it gives you a number in the range 0 to 6, and that same key when passed, and again gives you the same number. So if I say 106B equals Julie, it pushes 106B through the hash function and says, "Okay, well 106B comes out as one." You add those numbers together, jumble them up, the letters, and you get one. And so it went into the table, and it said, "Okay, well that's bucket number one, make a new cell for 106B Julie and stick it in the table." So now there's a non empty chain in one place, the rest are empty. If I say 107 is being taught by Jerry, 107 happens to get the same hash function. 107, 106B, different letters but it just happen to come out that way, hash to one and so it went ahead in this case just tacked it on to the front of the

list. If I do 106A being taught by Patrick, 106A happens to sum to zero in the hash function. That's all we know about the black box that ends up loading it up into the zero's slot. I say, well wait, it's being taught by Nick, it also come out to be zero and so it ends up kinda pushing on the front of that.

So I go 103 is being taught by Maron and it's in bucket five and so as I kinda type things in I'm starting to see that things are just kinda going around in difference place in the table. A little bit of clustering in the top, a little bit of empty at the bottom, but as I continue to type in strings for other course numbers, I'd hopefully kinda fill out the table to where there was kind of an even, roughly even spread across that. When I want to look something up because I wanna see who's teaching 106B, then it will use the same process in reverse. It asks the hash function, "Well which bucket should I look in?" And it says, "Oh, you should look in bucket one because 106B's hash code is one." So all the other buckets are eliminated from consideration, so very quick kind of focused down to the right place and then it actually just does a link list reversal to find 106B in the bucket where it has to be, and so the key to realizing what makes this so magical, right, is that the choice of the number of buckets is up to me. I can pick the number of buckets to be whatever I want. I can pick it to be five, I can pick it to be 10, I can be to be 1,000, I can pick it to be a million.

If I choose the number of buckets to be kind of roughly in the same order of magnitude as the number of entries. So if I wanna put a thousand things in my table, if I make a thousand buckets to hold them in and my hash function can divide stuff pretty equally across those thousand buckets, then I will expect that each of those link lists is about length one. Some will be two, some will be zero, but if it's doing its job right then I have a bunch of little tiny buckets, each of which holds a tiny fraction of the data set, in this case one or two, and it's very easy to search through them. If I know I'm gonna put a million things in, then I make a million buckets, and then I divide one out of a million things, I run it through the hash function, it gives me this number that says, "Oh, it's 862,444." I go to that bucket and the item is either there in that bucket or I have to look a little bit to find it, but there's no other place in the table I have to look. So the fact that there's a million other entries kinda near by in these other buckets is completely not important to us, right, and so the searching and updating, adding, and replacing, and removing, and stuff all happens at the bucket level and by choosing the buckets to be the number, total number are gonna be roughly equal to the size of the things, then I have this constant time access to the tiny part of the subset of the set that matters. That's pretty cool.

Let's write some code. It's kinda good to see the whole thing doing what it needs to do. So I'm gonna go ahead and go back over here. All right, so I've got my map which has add and get value and not much else there, right, but kinda just set up and ready for us to go. What I'm gonna build here is a link list cell that's gonna hold a string key, a val type value, and a pointer to the next, and so I plan on having each one of these cells, right, for each entry that goes in the table and then I'm going to have an array of these, and I'm gonna make a constant for it, and then I'll talk a little bit about the consequences of that. So I'm gonna make a constant that is – so I make it 99. So then I have 99 buckets to start

with. That's not gonna solve all my problems, it's gonna show that it's kinda tuned for maps that expect to hold about a hundred things, but – and so I have an array in this case, so if you look at deciphering this declaration is a little bit complicated here. It says that buckets is the name of the variable, it is an array of num buckets length, so a fixed size, 99 entry array here. Each of those entries is a pointer to a cell, so ready to be the head pointer of a link list for us. I'm gonna add a – well, we'll get there in a second. I'm gonna go over here and work on the constructor first. So the first thing I'm gonna do is make sure that those pointers have good values. If I do not do this, right, they will be junk and there will be badness that will happen, so I can make sure that each of them starts off as an empty list, and then correspondently, right, I would need to be doing a delete all list cells here, but I've been lazy about doing that and I'm gonna continue to be lazy just knowing that I'd have to iterate through and then delete all the cells in each of those lists that are there.

And then for add and get value what I'm gonna – else we need to do is figure out in both cases, right, which is the appropriate list to operate on. So running the hash function on my key, seeing where it tells me to look, and then either searching it to find the match for get value to return the matching value or in the add case, to search for the matching entry to overwrite if it exists and if not, then to create a new cell and tack it on the front of the list. So let's write a helper function because they're both gonna do the same thing. I'm gonna need the same thing twice. So I'm gonna put a hash function in here that, given a key and a number of buckets, is gonna return to me the right bucket to look in, and then I'm gonna write a find cell function that, given a key and the pointer to a list, will find the cell that matches that key or return a null if it didn't find it. Okay, let me put my hash function in. I have a good hash function down there in a minute that I'm gonna pull out, but I just – so what I'm gonna do, I'm gonna make an interesting choice here and I'm gonna have my hash function just return zero, which seems pretty surprising.

But I'm gonna talk about why that is. It turns out, when I'm first building it that actually is an easy way to test that my code is working and to not be dependent on the behavior of the hash function. At this point it says, "We'll just put everything in the zero's bucket." Now that's not a good strategy for my long term efficiency, but it is a good way to test that my handling of the bucket, and the searching, and the finding, and inserting is correct, and the performance I should expect to be abysmal on this, right, it should be totally linear in terms of the number of elements to add or to get them because they'll all be in one link list with no easy access to them. And then eventually I can change this to divide across the buckets and get better spread, but the correctness of the hash table isn't affected by this choice. It's kind of an interesting way to think about designing the code. The other helper I have down here is gonna be my find cell that, given the head pointer to a list and the key, is gonna go find it, and it's basically just if this ker cells key equals the key, then we return it and then if we've gone through the whole list and haven't found it, we return a null.

So this guy is returning a cell T star, right, as the matching cell within the thing, and this one is gonna provoke that little C++ compilation snafu that we encountered on the binary search tree, as well, which is that cell T, right, is a type that's declared in the private

section within the scope of my map. So outside of the scope, which to say, before I've crossed the my map angle bracket val type colon, colon, it doesn't actually believe we're in my map scope yet. It hasn't seen that and it can't anticipate it, so I have to actually give the full name with the scope specifier on it, and then because of this use of the template name outside of its class is a special place for the C++ compiler, the type name keyword also needs to go there. Okay, so there's the heavy lifting of these two things, right, getting it to hash to the right place, which I kinda short changed, and then searching to find a cell. If I go back up here to get value, and I call the hash function, given my key and the number of buckets to find out which bucket to work on, I go looking for a match by calling find cell of the key in buckets of which bucket, and then if match does not equal null then I can return match as val and then otherwise we say, "No such key found." So that's the error case, right, for get value was that if it didn't find one, its behavior is to raise an error so that someone knows that they've asked something it can't deal with.

Okay, so the key thing here usually is that the hash is some sort of constant operation that just jumbles the key up, in this case it just returned zero, but could actually look at the elements of the key, and then it does its reversal down that link list, which given a correct working hash function should have divided them up in these little tiny chains, then we can just find the one quickly by looking through that link list. If we find it, right, we've got it, otherwise, right, we're in the error case. Add looks almost exactly the same. I start by getting the bucket, doing the find cell, if it's not – if it did find it, right, then we overwrite, and then in the second case, right, where we didn't find it, right, then we need to make ourselves a new cell, and copy in the key, copy in the value, set it – in this case, the easiest place, right, to add something to a link list is I should just do append it to the front, right, no need to make it hard on ourselves, right, we're not keeping them in any particular order, whatsoever, right, they're base on kind of order of entry.

So if somebody gives us a new one, we might as well just stick it in the front, that'd make our job easy. So it will point to the current front pointer of the cell, and then we will update the bucket pointer to point to this guy. So I think we are in okay shape here. Let me take a look and make sure I like what I did, right, so hashing, finding the match, if it found a non empty, so – this is the overwrite case of finding an existing cell, we just overwrite with that. Otherwise, making a new cell and then attaching it to be the front of the bucket there. So for example, in the case where the bucket's totally empty, it's good to kinda trace that. That's the most common case when we start, right, if we hash to bucket zero and we do a search of a null link list, we should get a null out, so then we create a new cell, right, and we set the next field to be what the current bucket's head pointer was, which was null. So there should be a single to list and then we update the bucket to point to this guy. So then we should have a single list cell with null trailing it or inserting into the empty bucket case. I like it. Let's see if I have a little code over here that adds some things, car, car, cat, with some numbers. Okay, and that tries to retrieve the value of car, which it wrote a three and then it wrote a four on top of it, so hopefully the answer we should get out of this is four. Okay, if I – be interesting to do something, like ask for something that we know is not in the table, right, to see that our error case gets handled correctly, the switch key's found.

And if I were to continue to do this, I could do a little bit of stress testing to see how that zero is causing us some performance implications, but if I sat there and just put tons and tons of things in there, right, made up strings like crazy and stuffed them in that with numbers, right, I would just be creating one long chain off the zero bucket, ignoring all the other ones, right, but then it's kinda stress testing my link list handling about the adding and searching that list, but would also show us that we expect that as we got to be, have 100 entries, 200 entries, 300 entries that subsequent adds and get values, right, would show a linear increase in performance because of the cluster that we got going there.

So let me give you a real hash function. I'll talk you through what it's doing and then I'll leave you with the thought of, that writing one of these actually correctly and reliably is actually more of an advance exercise than we're gonna look at, but this is actually a hash function that's taken from Don Knuth's Art of Computer Science, some of the work that guides a lot of computer scientists still to this day, and it has the strategy I basically told you about of, for all the characters that are in the string, right, adding it into the thing, but having multiplied, right, the previous hash code that's being built up by this large multiplier which is in effect kinda taking advantage of the positional information. And so the multiplier happens to be a very large negative number that's a prime that says, okay, the first time through, right, it'll take the hash code of zero and multiply it by and add the first character. So the first character gets added in without any modification. The next character, though, will take the previous one and multiply it by one power of the multiplier and then add in the next character, and then that sum, right, on the sub [inaudible] gets multiplied again by the multiplier, so raising it to the squared, and the third, and to the fourth powers, it works down the string.

The effectiveness actually is that it's gonna do a lot of overflow. That's a very large number and it's adding and multiplying in this very large space, so in effect we're actually counting on the fact that the addition and multiplication that is built into C++ doesn't raise any errors when the numbers get so large that they can't be represented. It just kind of truncates back down to what fits, and so it kind of wraps around using a modular strategy here, like a ring. And so we'll actually end up kind of making a very big number out of this. The longer the string is, the more those multiplies, the more those powers raise. We've got this huge number, what we do is kinda truncating it back to what fits in to a long and then when we're done, we take whatever number fell out of this process and we mod it by the number of buckets to give us a number from zero to buckets minus one. And so whatever gets stuffed into here, right, it's reliable in the sense that it's, every time you put in the number you'll get the same thing back. We know it will be in the range end buckets because of that last mod call that worked it out for us and we then use that to say which bucket to go look in when we're ready to do our work.

So if I switch that in for my old hash function, I shouldn't actually see any change from the outside, other than the performance, right, should suddenly actually be a lot faster, which is hard to tell when I have three things in there, but – I'm gonna take out that error case. In this case car and cat now are probably not very likely to be going to the same bucket. In fact I can guarantee they don't go in the same bucket, whereas before they

were kind of overlapping. I'll leave that code there for a second, although really I would say that point of hashing really is not to get so worried about how it is that the hash code does it's work, but to understand the general idea of what a hashing function has to do, right, what its role is and how it relates to getting this constant time performance. So I'll give you an example of hashing in the real world, I think is really interesting to see it actually happen and people not even realizing how hashing is so useful.

So I ordered something from REI, which is this camping store and it – they didn't have it in stock, so they had to place an order for it. This is actually kinda awhile ago, but I – so then when I called to see if it's in, I call them on the phone, I say, "Oh, is my order in?" And they don't ask me my name. I thought this was very funny. Like, "Oh, it's – I'm Julie. I wanna know about my order." They're like, "Okay, what are the last two digits of your phone number?" And I'm like – you know, it's not one of those things you can answer right off the top of your head. Okay, 21. Turns out last two digits of my phone number are 21. And they say – and they go and they look in the 21 basket and then they say, "What's your name?" Now they want to know my name. I'm like, "Okay, my name's Julie." Okay, they look it up and they're like, "Okay, it's here." I'm like, "Okay." So then I go to get it, like a couple days later, and I go to the store and I'm standing in the line waiting for them, and I look up on the wall and they have 100 baskets, little wire baskets up on the wall and they're labeled 0 through 9 over here and 0 through 9 over there, and when I come in they ask me the same question. "What's the last two digits of your phone number?" Now I'm all prepared because I've already been primed on this question.

I say, "21." Then they walk over here, right, to the 21 basket here. It's like the top digit in this digit, and there's only one piece of paper in there, and they pull out my order thing, and then they get me my tent and everybody's happy. So while I'm there, I try to engage the cashier on how this is an example of a real world hashing system and I still got my tent, I'll have you know, but it was close, right. They're like, "Okay, yeah, you crazy crackpot. You can now leave now." I was like looking at it and them and I tried to talk to them about the number of digits because in some sets, right, this is a very good example of what the investment in the number of buckets is versus what tradeoff it gives you, right. Because there's this very physical sort of set up of this, it's like by choosing a larger number of buckets, right, you can more fine grain your access, but that investment in buckets means you're kind of permanently installing that story. So in this case, right, they didn't use just the last digit of my phone number. They could've done that and they would have only had ten buckets on the wall, but then they would expect, you know, ten times as many things in each bucket.

And apparently their estimate was the number of active orders in this backorder category was enough to warrant being more than ten, you know, significantly more than ten, but not so much more than 100, then in fact 100 buckets was the right investment to make in their bucket strategy. They didn't put three buckets on the wall because, you know, like what are they gonna do, have this big 3D sort of thing. They didn't enjoy this discussion, I went on for hours with them and they were, like, really not amused. But it had, you know, the three digits, then you'd get this even more likelihood that each bucket would

be empty, but – or have a very small number of things, but given their set up, that seemed to be the right strategy was to say 100 buckets and now – they didn't ask me the first two digits of my phone number. They asked me the last two. Why does that matter?

Student: Because you're using all [inaudible].

Instructor (Julie Zelenski):Yeah. It's hated a lot. If you ask, like Stanford students, right, like especially when, you know, like campus numbers used to all be 49's or 72's or something, so like, if you use the first two digits, right, you'd have everybody's in the 49 bucket or the 72 bucket or something, and a whole bunch of other ones not used. An example, even the first two digits are never 00, like there's a whole bunch of buckets that don't even get used as the first two digits of a phone number. Phone numbers never start with zeros. That they rarely actually have zeros or ones in them at all. They try to use those for the area code and stuff in the leading digits.

So I thought it was just a really interesting exercise and like, oh yeah, they exactly had kinda thought through hashing. Of course, they did not think it was hashing, and they thought I was a crackpot, and they didn't want to give me my stuff, but I paid my money and they [inaudible], but it kinda shows you, okay what you're trying to do here is try to make that number of buckets, right, roughly equal to the number of elements so that if your hash function's dividing up across your whole world, you've got constant access to what you have.

Student: Why is there an L at the end of one item?

Instructor (Julie Zelenski):You know, that's actually just a – C++ isn't for this. This is a long value that without it, it assumes it's an int and that number is to big to fit in an int and it is twice as big as a long in terms of space and so the L needs to be there, otherwise it will try to read it as an int and through away part of the information I was trying to give it. So it's the way of identifying it along constant. So let me talk about this just a little bit more in terms of actual performance, right, we've got N over B, right, so if the division is complete, right, the number of entries over the number of buckets, if we make them kind of on the same order of magnitude, kind of in the same range, so having to sum this could be to make a little bit of an estimated guess about where we're going, and then there are some choices here in how we store each bucket, right.

We could use a little array, we could use a link list, right, we could use a vector. We expect this to be very small, is the truth. We're hoping that it's gonna be one or two, and so there's not a lot of reason to buy any real expensive structure or even bother doing things like sorting. Like you could sort them, but like if you expect there to be two things, what's the point? You're gonna spend more time figuring out whether to put it in the front or the back then you would gain at all by being able to find it a little faster. So the link list is just gonna be easiest way to get kind of allocated without over capacity, excess capacity that's unused in the bucket. You know it's gonna be small, use the simple strategy. It's also, though, an important thing that the hash type is likely to do in kind of a industrial strength situation is it does have to deal with this idea, well what if that number

of buckets that you predicted was wrong? And so the map as we have given it to you, right, has to take this sponge because it doesn't know in advance how many things are you gonna put in. The only way to know that is to wait and see as things get added, and then realize your buckets are getting full.

So typically the industrial strength version of the hash table is gonna track what's called the load factor. And the load factor is just the number of total entries divided by the number of buckets. When that factor gets high, and high actually is actually quite small, in fact. If it's two, is often the trigger for causing a rehash situation, so not even letting it get to be three or four. Just saying as soon as you realize that you have exceeded by double the capacity you intended, you planned for 100 things, you got 200 things, go ahead and redo your whole bucket array. So in the case, for example, of our simple like 0 through 7 case, right, maybe it's 0 through 6. One, two, three, four, so I have one that had six buckets, right, and then I've gotten to where each of them has two or maybe three in some and one in another. The plan is to just take this whole bucket array and enlarge it by a factor of two, so grow it, and in this case, from 6 to 12, and then rehash to move everybody. The idea is that the earlier decision about where to place them was based on knowing that there was exactly six buckets and so where it was placed before is not likely to be the place it will place if you have, you know, 12 choices to lay them down into.

And ideal – in fact you don't even want it to be that case. Like, if they all land into the same bucket, right, you would have the same clustering and then a big empty clump, and so you would rehash everything, run it through the new hash function having changed the number of buckets, and say, "Well, now where does it go in the hashing scheme?" And hopefully you'd end up getting a clean table, right, where you had 12 items now with one in each bucket, ready to kinda give constant performance through that and then potentially, again if you overload your load factor, go back in and rearrange stuff again. So using this strategy a little bit like the one we talked about with the binary search tree of just wait and see. Rather than if you got, you know, just because you get two items in a bucket it doesn't immediately cause any real alarm or crisis, it waits until there's kinda sufficiently clear demand that exceeds the capacity plan for, to do a big rearrangement. So you'd end up saying that adds, adds, adds would be fast, fast, fast, fast, fast.

Starting to get just a hair slower, right, having to do a search through a few things as opposed to one, and then every now and then, right, you'd see an add which caused a big reshuffle of the table, which would be a totally linear operation in the number of elements and then they'd be fast again for another big clump of inserts until that same expansion might be complete or triggered by it growing even more. But dividing it sorta by the total number of inserts, so if you had 1,000 inserts before you overloaded and then you had to copy all 1,000 things to get ready for the next thousand, if you average that out, it still ends up being called a constant operation. So that's pretty neat. It's a – really one of the, sort of, more, I think, easily understood and beautiful algorithmic techniques, right, for solving, right, this hard problem of search and retrieval that the vector and the sorting and the BST are all trying to get at, but – and sometimes the sorter, vector and the BST are solving actually a much harder problem, right, which is keeping a total ordering of all the

data and being able to flip it, traverse it and sorted order is one of the advantages that the sorted vector and the BST have that hash table doesn't have at all.

In fact, it's jumbled it all up, and so if you think about how the iterator for the map works, our map is actually implemented using a hash table, it actually just iterates over the link list, and that explains why it almost appears completely random. If you put a bunch of things in the table and you go to iterate and visit them again, that the order you visit the keys seems to make no sense, and that's because it's based on their hash codes, right, which aren't designed to be any real sensible ordering to anybody not in the know of how the hash function works, whereas iterating over a vector that's sorted or iterating over a BST or a, in this case, our set, for example, is backed by a binary research tree will give you that sorted order. So it solves this harder problem, right, which cause there to be more kinda more investment in that problem, whereas hashing solves just exactly the one problem, which is I want to be able to find exactly this value again and update this value, and nothing more is needed than just identifying this, not finding the things that are less than this.

For example, if you needed to find all the values that were less than this key alphabetically, right, the hash table makes that no easier than an unsorted vector, whereas things like assorted vector and binary search tree actually enable that kinda search, you could find the place in the tree and kind of work your way down the left side to find the things that were less than that. That's not actually being solved by the hash at all. Its use of memory is actually comparable to a binary search tree in that it has a four byte pointer per entry, which is the next field on the link list chain, and then there's a four byte pointer for each bucket, the head of the link list cell. Given that we expect that the buckets to be roughly equal to entry, than we can kinda just summarize that as well. Each entry represented an eight byte overhead, which is the same eight byte overhead, right, that the left and right pointers of the binary search tree add.

Does it have any degenerate cases? That's a good question. So one of the things that made the binary search tree a little bit of a heavy investment was that to get that balancing behavior, even when we're getting data coming in in a way that's causing a lopsided construction, we have to go out of our way to do something special. What is the degenerate case for hash? Is there something that caused it to behave really, really badly? Anyone wanna help me out with that? Is it always good? Does it always give a good spread? Can we end up with everything in the same bucket somehow?

Student:[Inaudible] repeated it that way?

Instructor (Julie Zelenski):So if you have repeated elements the way that – both versions of the map work is they overwrite. So actually the nice thing is, yeah, if you did it again you just take that same cell and overwrite it, so in fact we wouldn't get a clustering based on the same entry going in and out. But how'd we end up with everything being in the same bucket? Mostly comes out, if you look at your hash function, when would a hash function collide? Right, and if you have a dumb hash function, right, you can definitely have some degenerate cases.

My dumb hash function of return zero, right, being the worst example of that, but any hash function, right, that wasn't being particularly clever about using all of its information might actually have some clustering in it. It is possible, for example, even with a particularly good hash function that there are strings that will hash to the same thing, and it's like, if you somehow got really, really unlucky that you – and had a hash function that wasn't doing the best job possible that you could get some clustering, but in general it doesn't require – the sole responsibility, right, for the generate case comes down to your hash function. So as long as your hash function and your inputs match your expectation, you don't have any surprises about how they got inserted the way it was in a binary search tree, which is sometimes hard to control.

Student: Just as you could underestimate the number of buckets you need –

Instructor (Julie Zelenski):Yeah.

Student:– could you over estimate the number of buckets you need by a large amount –

Instructor (Julie Zelenski):Um hm.

Student:– that's wasting a lot of it –

Instructor (Julie Zelenski):Exactly.

Student:– memory, and do you then go through and take that memory back?

Instructor (Julie Zelenski):Yeah. Yeah, so that's a great question. So a lot of data structures don't shrink, right, and for example, the vector often will only grow on demand and then even if you deleted a bunch of, or removed a bunch of elements, it doesn't necessarily consider shrinking a big deal, and so that's often a – maybe it's a little bit lazy, but it turns out, right, that having a bunch of memory around you're not using doesn't have nearly the same penalty that not having the memory when you needed it and having to do a lot of extra work to find things.

So typically, right, you would try to pick a size that you are willing to commit to, and say, "Well, no matter what, I'll stay at this size. I won't get any smaller." But that, as you grow, you might not shrink back to that size. You might just let the capacity just kinda lay in waste. There are good reasons to actually be tidy about it, but again, some of this is hard to predict. It might be that your table kinda grows big and then a bunch of things get taken out, and then it grows big again. Like, maybe you have a new freshman class come in, and then you graduate a whole bunch of people. Then at the point where you graduate them all, you're about ready to take in a new freshman class and so it might be that in fact, right, the capacity you just got rid of, you're gonna plan on reusing in a minute anyway, and so maybe that clearing it out and totally releasing may actually be an exercise that was unimportant. You're planning on getting back to that size eventually, anyway. Most hash tables tend to grow is actually the truth. Right, you tend to be

collecting data in there and they tend to only enlarge. It's kind of unusual that you take out a bunch of entries you already put in.

And so I just put a little last note, which is like, okay, well I had said earlier that when we have the map, the key had to be string type and I was gonna at some point come back to this and talk about why that was the one restriction in all our template classes, right, you can put anything you want in statter or vector or stacker queue. That map lets you store any kind of value associated with that key, but that key was string type, and so given how our map is implemented as a hash table, that actually starts to make sense, that if you're gonna take the key and kind of jumble it up and map it to a bucket, you need to know something about how to extract information from the key, and so this case, knowing it's a string means you know it has characters, you know its length, and you can do those things.

If you wanted to make a map that could take any kinda key, maybe integers is key or student structures is key or, you know, doubles is key, any of these things, it's like, well, how can you write a generic form that could use a hashing operation on that kind of key and map it to bucket numbers. You would build a two type template to kinda get this working in C++, so you can actually have a template that has a type name for the key type, a type name for the value type, and then in the member functions you'd refer to both key type and value type as these two distinct place holders. And then when the client set it up, right, they'd be saying, "Okay, I want a map that has strings that map to integers." So maybe this is words and the page they appear in a document. I have a map of integers and a vector string. Maybe this is score on an exam and the names of the students who got that score, doing it that way, and to make this work, right, there has to be some kind of universal hashing that, given some generic type, can turn it into an integer in the range of buckets.

The – what it's gonna require here is that the client get involved. That you can not write this generic hash function that will work for all types, right, if it's a student structure, what are you gonna look at? The ID field, the names field, under like – there's just no sensible way that you can talk about a generic type and talk about how to hash it, so what you would have to do is have some sort of client call back, so probably you would create this by passing out a call back that's given a key type thing, and a number of buckets would do the necessary machinations, right, to hash it into a right number. And so that would be kinda one of those coordination things between the client and the implementer about the client knows what the data is what the data is it's trying to store, and kinda how to mash it up and then the map would know, okay, given that number, where to stick it.

So that's what it would take, and then rather sorta load that onto our novice heads, we went ahead just said, "Okay, it'll always be a string, so that we can do a hash internally without having to get involved." All right, any questions about hashing? I'm gonna give you like, a two second talk about set because it's the one ADT I never said, "Well how does it work? How does it work? What do we do?" We're talking about, it doesn't add any new things that we haven't already done, so in fact I'm just gonna tell you what it does, and then your picture will be complete, right. It wants to do fast searching, fast

updating, add and remove, and it also has all these high level operations, such as intersect, union and difference, that are kind of its primary set of things to do. A bunch of the things we've already seen, right, would actually be a reasonable strategy for upholding stat, right, using some kind of vector or array. Most likely you'd want to keep it in sorted order because that would buy you the fast lookup for contains, but the add and remove, right, would necessarily be slow in those cases because of the continuous memory.

The link list, not really probably too good of an option here because it contains then becomes linear and add and remove similarly, right, liner, so it doesn't actually get you anywhere, in fact it just adds the memory in pointers and gunk like that. The binary search tree, probably a pretty good call, right, that's the – kinda meshes the advantage of Adamic structure with the sorted property to be able to give it fast update adding and removing and searching all can be done in logarithmic time if you have balancing. And then it actually also enables the high level ops, so the idea of being able to do a union intersection difference, and actually being able to walk over both of the trees in sorted order, which is very easily done with a [inaudible] reversal, makes those operations actually more efficient to implement than actually kinda having to deal with data coming at you all different ways.

It turns out hashing really won't quite do it very easily for us because of the same need about the template situations, like, well you have to have a hash function that can hash these things and do the right thing, and so the tree happens to be a good of just – if the client can just give you ordering information, you can do all the hard work without pushing hashing onto them. So in fact the way our set really does work is it does use a binary search tree. It happens to use it through another layered abstraction that there actually is a BST class that we've never really encountered directly, but the BST class just takes the idea of a binary search tree, adds balancing, and all the properties about finding and inserting and adding, and packages it up, and then set actually just turns out to be one liners making calls into the binary search tree. This is where all the heavy lifting goes, is down there.

That kinda completes the big picture, so some of stuff, right, so our map uses our hash, right, our BST uses – or set uses the BST, which is the binary search tree, right, our vector, right, using an array, and then that stacks and queues having both viable strategies both for array vector based backing as well as link list both getting good time performance. And then you saw with the PQ, right, even yet another structure, the heap, right, which is kinda a variant of the tree, and so those tend to be the kind of really classic things in which a lot of things get built, and so having a chance to use them both as a client and kinda dig around as an implementer kinda gives you this big picture of some of the more common ways that cool data structures get built. So that's all for Friday. We're gonna do some more good stuff next week. I'm gonna go to the Terman café in a little bit, if you've got some time, please come, and have a good weekend. Work hard on pathfinder.

[End of Audio]

Duration: 52 minutes

ProgrammingAbstractions-Lecture25

Instructor (Julie Zelenski):Okay, we're experiencing a little technical difficult here that's going to be resolved very quickly because Jason went to get the magic piece of equipment, but in the meantime, I'll just fake it with my big personality up here. Okay, so thing's that are going on: Pathfinder is due this Friday. So how many people have been making good progress on Pathfinder? You have one of the algorithms already implemented and working? Anybody with one already done? Almost done? Okay, maybe a little bit of a slow start on this one, but – and then there's two pieces that you do have to get ready coming in this Friday. Do remember that you can use a late day on it if you absolutely have to, but we don't really recommend it, because it does kinda put you behind getting ready for finals. And our final is at the very end of finals week, so on Friday. So a week and something from today in the 12:15 slot. I don't know what room we'll be scheduled in, but you can guarantee it won't be Terman, that's all I can tell you for sure. And when we do know, we'll let you know. Anything else administratively I need to tell you guys? Okay, Bishop.

Student:[Inaudible]

Instructor (Julie Zelenski):There is an extremely limited run optional finals. You can talk your way into one Thursday if you can convince us of the compelling need for said thing. So send an email to Jason. I know a couple of you told me at the beginning of the quarter you were in this bind about being able to make it to the Friday exam. And so we – against my better judgment, I allowed myself to be talked into that. Okay, all right, I'm gonna have to just do this kinda blind, because the piece I need is not here, so I'm going to actually just talk to you about what it says on my slides and then when Jason arrives we'll plug it back in and see what's going on. Oh, there's Jason. There we go. So what I'm going to do today is actually something kinda fun. And so I'm gonna take a case study of a particular data structure design, and the one I'm looking at there is the lexicon. So how many of you actually ever decided to just open up the lexicon file, just to see what was in there at some point during the quarter. We used it a couple of times. Anybody want to tell us what they learned by opening up that file? Anything interesting?

Student: It's a data structure called a DAWG.

Instructor (Julie Zelenski):It's a data structure called a DAWG. You're like, well that's a pretty funny name for something. And was the code particularly well-[inaudible], easy to read, easy to understand? No. Who wrote that code? She should totally be fired. Yeah, so this what I wrote kinda a long time ago, actually, to solve this particular problem. And actually I would like to key through kinda how it is I came to discover the DAWG and why it is the DAWG was the choice that made sense for what the need pattern we had for the lexicon was. So let me just refresh what the lexicon does. It is a special case of kinda set or map. It is a container where you're going stick things in, they're going to be unique, right? There's no need for any sort of duplicate. The keys are always strings. And in fact, they're English words, so that's going to be an important thing to remember, that they're not just arbitrary sequences of letters, right? They have certain patterns to them

and certain sizes that might actually be something we could capitalize on. When we know something about the domain, we're able to kinda special case or tune or data structure to solve that problem very well, in particular. It has no associated value, so it doesn't have dictionary definitions or synonyms or anything like that, right?

So it really just is exists or doesn't exist that we were interested in supporting. And it also has, in addition to the kind of being able to add words and check if a word is contained, it's also this prefixing operation that we had talked a little about in Boggle, and that came back on the midterm. Like, why was prefixing important, right? It was critical for pruning these dead-end paths, these searches on the Boggle board that otherwise would really have taken a lot of time and kinda led to nothing of any value. So what I'm gonna to talk you through is some of the obvious choices based on data structures we already know about, and see where they would go. So this is a thought experiment. It's not that I wrote all these to find out. Actually, I mostly did a lot of this on paper, trying to figure out what options we had and how they would play out and what kind of trade-offs I'd have to make to make them work. So the simplest case of any kind of collection is to think about whether vector or array can do the job for you. In this case, having a vector that's kept in sorted – alphabetically sorted – order seems like a pretty good, easy way to get this set up and running and how would it behave? So in order to do a contains (word) on this data structure, what algorithm are you going to use to do your searching? Binary search, right? It's in sorted order, right?

We've got to take advantage of that, right? If we do a linear search I'm looking for the word "mediocre," I'm going to be trucking through, you know, thousands of words before I find it. So definitely want to be using binary search. Equals-equals, comparing my strings, less than, greater than, dividing in half. And so it should run a logarithm time. Logarithm time, one of those greater performers that you learned in the PQ assignment, hopefully. It's just basically not measurable on today's computer. So even for entries of 100,000, 500,000, basically for free. How do we do contains (prefix)? Can we also do a fast prefix search given this data structure? I hope so right? Thinking about the same kind of binary search. In this case, though, looking at substring, right? So comparing I'm looking for a three-character substring, only looking at the first three characters, decide which way to go. Once I find something that matches in the first three characters, I'm done. So again, still using the sorted property to quickly narrow in on where it could be. When it's time to add a new word to this, if someone asked you to put in the word "abalone," I gotta use that binary search to find out where to go. So in logarithmic time I can tell you where the new word goes, but then to put it into position, there's gonna be some shuffling.

And that's where this operation starts to bog down, given the idea that you might be moving half or more of your items, on average, to make that space for that one to open up. It is going to require a linear time operation to get new words into the data structure. Okay, probably still, though, a reasonable approach, right? These are the operations that get done immense numbers of times in Boggle, right? You're doing a ton of contains (word)'s and contains (prefix)'s as you're exhaustively searching that big board. But the add's are done kinda once at the beginning. You build that big dictionary and then you

subsequently use it. And so if it took a little bit more time to built that thing, sorta an n-squared operation to kinda load it up, well, maybe that would be okay. The other we want to look at is space usage, and that's – and some of this is a little bit of an interesting artifact of the time I was working on this. It turns out space was actually a limiting factor in what we had available to us, in terms of core memories of our machines and what we could take up for our dictionary. In this case, it has per entry the size of the string, which we don't exactly know the string it represented. It's an abstraction to us, but we can make the assumption of kinda a simplification, that it's probably about the length of the string times of the size of the character.

And there might be a little bit more housekeeping associated with it, but that's a pretty good estimate of what it's taking. So in fact, it's the string data itself that's being represented in this vector without much else overhead. So if we say words are about average of about eight characters, about eight bytes per word. So if we have 100,000 we expect to have 800,000 bytes invested in this sort of vector arrangement. Excess capacity, a couple other things that might tweak that a little bit, but that gives us at least a good starting point to think about how this compares to alternatives. Okay, so then if we said we know that sortedness – that binary search is the real advantage of keeping it in sorted order, but the contiguous nature of the array kinda bites us in terms of insert. If we reorganize into that binary search tree to give ourselves the dynamic flexibility of wiring in the left and right halves through pointers, as opposed to contiguous memory, then we know we can also get add to run in logarithmic time. So looking at these operations, contains (word) is a tree search that's using equals-equals, left-right, deciding which way to go, and also performs logarithmically if the tree is balanced. Contains (prefix), same deal.

Keep looking at the [inaudible], working our way down, knowing which way to go. And then add can also perform in logarithmic time because it does the same search, falling off the bottom and then inserting the new word there, or finding it along the way and not having any work to do. So we can get all our operations done in logarithmic time, which we know to be kinda fast enough to actually not be worth fighting any harder for. Where did we pay for this? Nothing really comes for free, right? Overhead, right? And overhead, in this case, memory is certainly one of the places where there's going to be a pretty big cost that we now have the string data for each entry, plus a left pointer and a right pointer. And if we're doing the work to keep it balanced, which we probably ought to do if we want to avoid the degenerate cases of it kinda getting totally out of whack and degenerating into linear time, we're going to have a couple more bits, you know, bytes thrown into that factor, too. And so if we're assuming that there's ten bytes of overhead, four bytes in each of these plus another two for the balance factor, then we've added ten bytes of overhead on top of the eight bytes for the word.

Getting a little bit hefty. Something to worry about. Also in terms of code complexity, which I didn't put a bullet up for, but it's also worth kinda keeping – weighing it in your mind also which is that building the fancier, more complicated data structure probably meant you spent more time debugging it, more time developing it, right? You will have – it will be harder for someone later to maintain and deal with because it's actually just

more complicated code that somebody looking at a sorted vector isn't likely to actually – like, they want to add a new operation, let's say they add remove. You're able to take a word out of the lexicon. The lexicon doesn't currently support that. Somebody adding that on the sorted vector implementation probably won't find themselves too tripped up. Someone who has to do a remove out of a binary search tree, removing it, reattaching the subtrees and not destroying the balancing is a pretty complicated operation. So we've made – we've invested in this data structure that actually will keep on creating further development hassles for any modifications that are down the road for this data structure. So it is a big investment in your brainpower to keep it working.

So let's think about what a hash table can do for us. A hash table is the last thing that we looked at on Friday, kinda just moving away from this whole idea of sortedness and kinda moving it out of bound off to this idea of a hashing function. So if I have a hash table that has a certain number of buckets that uses a link list for chaining to handle the collision, then what I would have is my words just scattered around my table, allowing me to have quick access by hashing to the right bucket and then working my way down the bucket to C. So the contains (word) would do a standard hash table lookup. Take the word, you know, "mediate," run it through the hash function. It says oh, it's bucket three in this case. I go to bucket three. I walk down the link list. I find it or not. I can tell you whether the word "mediate," if it's in the table, had to be in bucket three. No questions asked. In that case, the expected time for that is going to be n over b, where n is the number of entries, b is the number of buckets. If we have buckets set to be in the rough order of the number of entries, then we're going to get constant access there.

Now we want to do contains (prefix). I want to know if there's any words in this table that begin with "med." Where do I look? If I run "med" through the hash function, do I expect that I would get the same number as other words that begin with "med," like "mediate" and "median" and "mediator?"

Student:

Yes.

Instructor (Julie Zelenski):You say yes. All right, let's take that to an extreme. If I expect that if a prefix in any longer words all have the same thing, then should it be the case, for example, I think about the simplest case of a prefix, "m." Should all the words that begin with "m" hash to the same place? If they did, we're in trouble, right? Now, if all prefixes did hash to the same place, then what I'm going to end up with is a really heavily clustered table where "a," "b," "c," you know, "z" – there's 26 buckets that have all the "z" words, all the "y" words, all the what not, and none of the other buckets would be used. All right, so in fact, actually, as part of the good behavior of a hash function, we're hopping that, actually, it does jumble things up. And then a word that looks like one but has another letter added on the end, we're hoping it goes somewhere totally different, that "median" and "medians" hopefully go to two totally disparate places. That if there were patterns where words that were substrings of each other all got hashed to the same location, we would end up with heavy clustering. Because there are a lot of words

that repeat parts of the other words in the dictionary. So in fact, if I want to know if there's any words that begin with "med," I have no a priori information about where to look for them. They could be anywhere.

I could hash "med" and see if "med" itself was a word, right? But let's say it was something that itself was just the beginning of something. "St," "str" – there's a lot of words that begin with that, but "str" itself is not a word. So hashing to that bucket and looking for it would only tell me if there's exactly "str," and then occasionally there might be some "str" words also there. But other than that, we just gotta look. And where do we look? Everywhere. Everywhere, everywhere, everywhere. That in order to conclude there is no word that begins with the prefix we have, the only way to be confident of that is to either find something, or to have searched the entire table and not found it. And so it does require this completely linear, exhaustive look through the entire contents of the hash table to know something, for example, is not a prefix. You might find one quickly. So in the best case, you might be in the first couple of buckets, happen upon one. But no guarantees, overall. And certainly when it's not a prefix, it's going to require a complete look-see of everything. That's kinda a bummer. Adding a word, back to the fast behavior again. Hashing in the bucket, checking to see if it's already there, adding if needed [inaudible]. So it gets good times on these two, but then kinda really bottoms out when it comes to supporting that prefix search.

Its space usage is kinda roughly comparable to that of the binary search tree. We've got the string data itself, and we've got the four-byte pointer on the link list. There's also the four-byte bucket pointer, which is over in this other structure here, this array full of buckets. If that number, though, is roughly equal to the number of cells, then you can kinda just count it as each entry had this 12-byte cell, plus a four-byte bucket pointer somewhere that you can kinda think of as associated on a per entry basis to tell you it's about eight bytes of overhead on top of the string. So 16 bytes for each entry in the table.

Okay, that's what we have so far. We've got pretty good performance on contains (word) no matter what we choose. We can get either logarithmic or constant time with either of these three data structure. So perfectly acceptable performance there. We cannot get contains (prefix) to run well on a hash table. Just doesn't happen the way hashing works. And then add can be fast on the two complicated pointer-based data structures to the right, but can't be fast on the sorted vector. And then we see kinda some trade-offs here in terms of space versus time, that there's very little space overhead in the sorted vector case, but really blossoms into two times, almost three times, the storage needed.

In the VST and hash table case, the code also got a lot more complicated when we moved up to those data structures. So then I'll tell you, actually, that it turns out that the one type of performance that was interesting to us when I was exploring this, but in fact, more important at the time, was the memory usage. That the memory usage was really the big obstacle that we had. The Official Scrabble Players Dictionary II is actually the source for the word list that we're using. It has about 128,000 words, I think, is exactly the number it has. And the average length of those is eight characters. And the file itself is over a megabyte. So on disk, if you were just to look at the listing of all the words, it's a

megabyte – a little over a megabyte there. If we were loading it into the sorted vector, we'd get something that was just roughly equal to that. It would take about a megabyte of space, plus a little bit of noise. The ones that double up that take it up to two, two and a half megabytes total. At the time we were working, and I know this is going to seem completely archaic to you, but we had these stone tablets that we worked on. And they typically had main memories of about a megabyte, maybe, like in an exciting case, four megabytes; four whole megabytes of RAM.

So it was not possible that we could dedicate one or two megabytes of your main memory to just your lexicon. It just wasn't – there just wasn't enough space to go around. So we're working in this very tight environment in terms of that. So a little bit more like, for example, today's world of embedded devices. If you're adding something for an iPod or a cell phone you have a lot less gargantuan memory. So your average desktop machine, having a gigabyte of RAM, now you're like, whatever, a megabyte here, ten megabytes there, free, all the memory you want. And so this seems kinda like a silly thing to focus on, but this was 1995, and it turns out, yeah, there was certainly more interest in keeping it tight to the memory. The Boggle lexicon that we have actually takes under a third of a megabyte. The memory that's used while it's working and searching and checking for words and prefixes is actually smaller than the on-disk form of the data we started with. So it actually does a kind of a bit of a compression as part of its strategy for storage, which is pretty interesting to think about. So I'll tell you, actually, the truth about how we first did Boggle, because it's kinda a little interesting historical perspective on it.

But Boggle was given by a colleague of mine for the first time, Todd Feldman. He was the guy who came up with the idea, which is a great idea. And he said, "Oh, this is a great assignment." And when he wrote it, he said, "Oh, I need a lexicon." So as part of kinda putting the assignment together, he was like, "Oh, we have to get some lexicons." So he looked around and he found a word list that had about 20,000 words. And I think we first found this word list and we said, "Oh, it's just impossible." It's going to take a megabyte or more and we just don't have that kind of space. So we need much smaller data files. So we had a data file that had about 20,000 words, which then takes about 200k. And he actually wrote it using a hash table. So given he wrote it as a hash table, he just didn't support contains (prefix). He just didn't put it in, and – because you can't do it efficiently. So it actually took a smaller amount of space and had a smaller word list and it wouldn't do contains (prefix). So then people wrote Boggle. And as you learned from that midterm question, you can write Boggle without contains (prefix). It spends a lot more time looking down these data ends, but it eventually bottoms out when it runs off the board and uses all the cubes.

And it actually, in some ways, almost made the program a little more satisfying to write in one odd way, which is, when you're – so it's the computer's turn, right, that uses the prefix pruning. So you would type in your words and it would be perfectly fine finding it, but it doesn't use the prefix part when it's doing the human's turn. You're typing in words; it's finding them. You're typing in the words; it's find them. Then you would go to do the computer's turn and you'd say, "Okay, go." And now it would take a really long time. It'd be hunting around the board. It would be working so hard your fan would be

coming on. You'd feel like oh, my computer's doing something. And then it would show up with twice as many words as you found or ten times as many words as you found, whatever. And in fact, then you would say, "Well, at least it had to work hard to beat me." And then you felt like I wrote this program that causes my processor to get busy, right? I feel good. And in fact, actually, the word list was pretty small. So in fact, it didn't – it wasn't even actually as likely to beat you because it didn't know as many words. 125,000 words is an enormous number of words.

The average – I've heard different factors on this, but the average person's vocabulary, English vocabulary, is about 20,000 to 30,000 words. It tends to be about 10,000 higher if you've gone to college, so maybe 30,000 to 40,000. The fact that it knows 80,000 more words than the average college graduate means it knows a lot of obscure words, too. So in fact, not only does it very quickly find all those words and then produce these ridiculous words that you've never heard of, it just – it's almost like it's taunting you by doing it in a millisecond. And back then it took a while, so it was actually like okay, it beat me, but it had to work hard. But then – so then I took over Boggle. I taught x maybe a quarter or two later and I said, "I'm going to write a lexicon. I'm going to go up and make a little project for myself, because I don't have enough work to do." And my goal was to make one that would do prefix, but that also would be tight enough on memory that we could actually load a much bigger dictionary file, because I thought I'd actually make the game even more fun to play.

All right, so here's where I started. This is a little excerpt from the middle of the dictionary. "Stra," yeah, that's totally the thing you notice, right, when you look at this thing. You see "straddle," "straddler," "straddlers," "straddles," "straddling," and then these words that you can't believe you're allowed to do this, but apparently you can put "er" and "ier" and "ies' and "ing" on almost anything out there and it makes a valid word. So although you never really thought about the fact that "straightforward," "straightforwardly," "straightforwardness," are all there, there is a huge amount of repetition. And this is where I said we're going to use some information about the domain. These aren't just arbitrary strings. They're not just random sequences of letters that are picked out of the air, that they develop over time. They're word roots. They're suffixes. There's these prefixes that mean things that actually show you that looking at this little excerpt out of the middle, there's an awful lot of words that repeat portions of other words. And that may be part of what we can capitalize on. That instead of really recording "straightaway" and "straightaways," repeating those first 11 characters and adding an "s," there's some way that I can unify the data that I have here.

The same way when we try to find codes, you have the same passage of code repeated two or three times. We're always saying unify it, move it into a helper and call it in three places. It's like can we do the same thing with our data? Can we share the parts of the words that are in common and only distinguish where they are different? Take a look at this. This is a little portion of something called a trie. A trie is spelled t-r-i-e and it's actually from "retrieval," but sadly, it's still pronounced tri, just to confuse you. And it is a variant of a tree. In this case it's a tree that has 26 children coming out of any particular node, one for each letter of the alphabet. And so starting from the root node at the top, all

the words that begin with “a” are actually down the A branch, and then there’s a B branch. There’d be a C branch and a D branch. And so the idea is that all the words, like if you think of the levels of the tree are actually like – these are all the zero if characters of the word.

And then that second level is all the characters that follow that zero if character. So here’s the “ab” words and the “ac” words and the “ax” words. Over here is the “st” words, but there’s not a “sx,” so there’s some places that actually aren’t filled out in this tree. And as you keep going down further, so that the depth you would trace to a tree would actually be tracing through the characters in sequence that make words. And so in this form, tracing out a path through this tree from the root down to a particular node tells you about prefixes. So there are words that begin with “a” and words that begin with “ac” and “ace,” and then there’s actually a little notation there that’s done by a visual of the bold and circle around it that says, “Oh, and the path that led here from the root is a word.” So “a” is a word and “ace” is a word and “aces” is a word. “Act” is a word and so is “acted” and “actor.” And in this case, the “act” part is totally shared, so everything that comes off of “act,” “acting,” “acted,” “actor,” “actors,” can all share that initial part. And for a word like “act,” it doesn’t seem that profound.

But you start thinking about words like “strawberry” or “straightforward” or “stratosphere” and realize that “stratospheric” and “strawberries” and “straightforwardly” can all leverage the other ten or 12 characters that were already invested in the root word, and that “straightjacket” and “straightforward” can even share “straight.” But there’s actually a lot of prefixes that can be unified across the space of the dictionary. So knowing that words have these patterns helps us to do this. So let’s build that. So the first form of this – and this, again, is a thought experiment. I did not actually write it this way because it would just be absolutely astronomically expensive. But it was a good way to kinda think about where I needed to go. I designed a new kinda trie node that had the letter and a little Boolean flag of is this the word, so whether it had the dark circle around it. And then it had a 26-member children array, so pointers to other nodes like this one. So totally recursive, staring from that root and then the idea is to use those 26 children as “a” being in the first slot and “z” in the last slot to make it really easy to kinda just trace letter by letter down to the levels of the tree.

So I’d have this one root node that pointed to the initial one, and then all the words that begin with “a” are off that first lot; all from “z” off the last slot. So when I want to know if a word is in the dictionary, what do I have to do in this data structure? How do I find it? I want to find the word “far.” Where do I look? Left? Right? Down? Which way? It would be great if you would tell me. Then I would understand what my data structure is.

Student:

[Inaudible]

Instructor (Julie Zelenski):Exactly. So at the very top I say okay, “f” is my first letter. Match my “f,” get me to the F node, and now it says recursively find “ar” from here. So

it's very recursive. It'll be like find the first letter and then recursively find the substring from here, working down. So find the "a" out of the next node, find an "r" out of the next node, and then when you get to that node, you'll check this little "is word Boolean," and say okay, was that path I traced marked in the dictionary as leading to something that's known to be a word? The prefix looks exactly the same, except for that last check for the as word. So given that nodes exist in this, because further down there there's some words, I just trace out a sequence of letters "str," and as long as there's something there, then I know that away from there are subsequent children beneath it that lead to words. And adding a word does the same operation. If I need to add the word "strawberry," then I start looking for "s," "t," "r," and at some point, I find some places where the nodes don't exist. Then I start creating them from there on down. So it is tracing what exists and then adding new nodes off the bottom, and then marking the final one as yes, this is a word.

What's interesting about this is that all three of these operations don't actually make any reference to how many words are in the dictionary. They are all dependent on the length of the word you're searching for or adding into the dictionary. If there's ten words in there, if there's 1,000 words in there, if there's a million words in there. But in no way does the big O depend on how big or how loaded the data structure is. That's kinda wacky. The longest word, according to the Oxford English Dictionary? There you go. You're not going to make that on the Boggle board anytime soon because it's only got 16 cubes. Even Big Boggle can't really make that puppy.

But just so you know, 45 is not so many, right? And, in fact, actually, here's a little thought for you. Not only is it not dependent on num words, it actually has a little bit of an inverse relationship with num words, especially the add operation. That as the data structure becomes more loaded, there's typically less work to do in add. So if "strawberry" is in there, and you go to add "strawberries," then actually you already have "strawberr" to depend on. You just need to build off the thing. So the fact that the more words that are in there, the more likely that some of the nodes you already need are already in place, and then you can just kinda blow past them and just add your new nodes. Now, that's odd. You think of most data structures as really as clearly as they get heavier, it takes more work to install new things in it rather than less.

Student:

[Inaudible].

Instructor (Julie Zelenski):Well, the thing is, they're organized by letter, and that 26-number array. So in fact, I don't even have to look. So I just go straight to the slot. If I'm looking for strawberry, I'll look for "s" and I go straight to the "t" slot and then the "r" slot. So in fact, if the mode was already in place, I'm not searching for it. It really just is exactly in the "r" slot. That's why there's 26 of them.

Student:

In each array?

Instructor (Julie Zelenski):Each array is 26 and they're A through Z and if there's an empty slot we're using a null. So in this case, it's optimized to make it super fast. I don't even have to look through the letters to find the one that matches. There's no matching. It just goes straight to the "r" slot, "z" slot, the "s" slot. We're going to see this is going to be a consequence we can't tolerate, but it is, at least, the starting point for thinking about how to build a data structure. So the space usage is where this thing really, really – so this is an extreme example of a space-time trade-off, right? We've got this thing that optimizes beautifully for OAV length of the word. So that means typically eight operations on average are going to be needed to add or to search for something. The space we're using here is 106 bytes per nodes, so that's this 26-member, four-byte array, plus two bytes for the character and the Boolean that are up there. One hundred and six bytes is a lot of memory, and the tree has about a quarter of a million nodes. So given the 125,000 words that I said are in the input, if you build it into a tree, they take about 250,000 nodes. That's an interesting sort of number to think about, though. You have 125,000 words. They take 250,000 nodes.

That tells you, actually, that kinda on average, each word has about two unique nodes, that even words like "straightforward" and "straightforwardly," on average, are sharing enough of their prefix – common prefix parts – that there's very little unique data added by any particular word in there that averages across the whole thing. That's pretty neat to know. However, this is just not going to fly. So here I was telling you that my main memory had four megabytes, maybe, on a good day, and now I've actually built a data structure that's going to require six times that in terms of storage. Okay, we gotta squeeze that down. So the first thing we can do is to realize that those 26 – allocated a full 26-member array, of which I only plan on using some of the slots, is pretty wasteful. So instead of saying I'm committing to a 26 per thing, I'll say well, how about if I have an array that actually is tight to the number of children coming out of this node? So the very first node will have 26. There are words that start with every letter in the alphabet. But from there, it very quickly winnows down. You have the letter "x." You do not need the full 26 children coming off of "x." There's only a few letters that follow "x" in the English language that are going to need to be fleshed out.

You need the "xa," you need an "xe," but you don't need "xj," you don't need "xk," a whole bunch of things like that. So if I change the structure here to instead of being a 26-member array of node pointers, I make it to be a pointer to a set of nodes pointers, so this is a dynamic array that I'm keeping track of the number of children in a second field here that the first one will be allocated to 26, the second will be allocated to eight. This is going to change a little bit of our running times, because now we won't be able to go exactly to the "s" slot. We'll have to go look for it. So on each step down the tree, it'll be looking through that sized array at this slot to find the right match. But it adds a constant factor, basically, on top of OAV length of the word. So the space issues of this is – we're just not storing all those nulls. And there were a lot of nulls. That the node itself is not just ten bytes, and then there's some space in this dynamic array out there, which is the number of children times the four-byte pointer, that on average, a node has six children. Across the 250,000 nodes in the data structure that's being billed from this, so really 26

was way overkill. About 20 of them had nulls for most of the nodes. And many of them, for example, at the very bottom of the tree, have completely none. They're all nulls.

So you get to a word like "straightforwardly." There's nothing after that "y," so it has zero children coming out the back of that one. And so certainly having 26 null children pointing away from that was a waste that we can tighten up. So when we get it down to this we'll have 44 bytes – or 34 bytes for the per node. Still a quarter million nodes. Still eight and half megabytes. So we're not winning any prizes just yet. But we are making good progress. That got us down a factor of three or four right there. So I still have the same number of nodes. All I changed was the pointers to the subsequent nodes further down the tree. So I – the kind of letters and how the letters connect to each other – the connectivity is still pretty much the same. It's just that in any particular node, how many children does it have? I was storing a whole bunch of nulls and now I'm not storing those nulls. So the number of nodes is still exactly as it was before. I'm going to do one other little squishing thing on this data structure and then I'm going to flip gears for a second. So still using kinda my CS side of things, how can I squash things down? I'm going to use a technique that we used in the heap. So in the case of the binary heap that we implemented into the PQ, you'll remember that we drew pictures in the handout that showed a heap with oh, I have a parent with two children, which has four grandchildren and so on. But that we actually compressed that down into an array, and in the array there was this pattern of here is the root node. So we'll call this level zero. And then the next two nodes are level one. And the next four nodes are level two and so on. So there's one here, two here, there's four here, there's eight here. And we kinda flattened it down by generation, so looking at what was at the top and what was underneath it.

When we did that, we removed all the space for pointers. Although we drew those pictures in the heap as though there were pointers there, in fact, there were not. There were no pointers being stored in the heap in this version of the PQ. And so that gets us back to kind of array like storage requirements, but by flattening it down into this array, and still kinda using the tree conceptually to work our way through it, we're getting the advantages of that traversal that's based on height of tree, not length of vector. So if we do the same thing with the trie, it's a little bit more complicated because there were a couple things about heap that made this a little easier. Heap always had two children, so there were these rules about how the tree was filled in that meant you could always compute parent and child index using this kinda divide by two, multiple by two exchange operation. In this case, what we're going to have to do is just lay it down by generation, but we don't know how big a generation's gonna be, so we're actually gonna have to keep track of it manually. So the root node will have all A through Z. So this will be level one, which is A through Z here, and then level two is – there'll be a bunch of slots here, which is the A followed by some letter, and then there'd be the B's followed by some letter, and then there'd be the C's followed by some letter, and so on. And then further down is level three, which are the three-character words. So these are all the single-character paths, two-character paths, three-character paths, kinda flattened down. And so if I look at what I changed here, my first node says it has no letter that's not the root node, it's not a word, and it says it's children begin at index one. So the next location. So it's not a multiply by two and add one or something like that.

It's okay, here's where my children begin. They're at slot one in the array. And then there are 26 of them. So that means that A, B, C, D are kinda in order there. If I look at A, "a" is a word, and it says its children begin at index 27 and there are 12 of them, so there are 12 characters out of the 26 that can follow an "a" in the sequence of a valid word. And then B's children begin at index 39, which is 27 plus 12. So that's where A's children are sitting here at index 27 to 38 and then 39 to – I think B has eight or so children – and C and so on. So we flattened it all down by generation into an array. That means all the space for pointers has now been removed. One serious consequence of this, though, is that the pointers got us flexibility, that that insert and remove was dependent on the idea that pointers really being there, pointers buying us the ability to insert and delete without doing a shuffle. That now, once I have flattened it, I have suddenly bought back the same problems that arrays have, which is if I have the words here that have "ab" for abalone, and "ac" for act and I don't happen to have any words with "ad." And I try to add the word "add," that in order to put "ad" in there, we're going to have to move everything over and then there's going to be a lot of shuffling and what not. So I'm starting to build a data structure that's losing some flexibility but saving space. So I'm, in some sense, focusing my attention on how can I build a data structure that's really fast to search, even if it was a little harder to build? Because it turns out the thing I most care about is making sure it can actually very, very efficiently look up words. Maybe building it once, being a little slow, isn't so painful. So the space issues in this gets us down to just ten bytes per node. All the pointers are gone. So there was 24 bytes of pointers that we've just eliminated. And that means we've got two and a half megabyte memory footprint right now. So we're kinda getting close to hitting the range for binary tree and hash tables. Now I'm going to go back to the domain again. So I worked on it kinda from the CS side, thinking about things I know about data structures and reorganizing arrays and heaps and pointers. I'm going to go back and look at that domain again and see if there's something else I can kinda take advantage of. So this is a different cross-section of the dictionary showing the words "adapted," "adaptor," "adaptors," "adapting," "adaption," adaptions," "adapts." So eight words that are all built off the "adapt" root with a bunch of suffixes.

I look sorta over a little bit further. I've got "desert," "deserted," "deserter," "deserting," "desertings," "desertion," "desertions," "deserts." Okay, and "detect" and "insert" and "perfect" and "invent" and "interrupt" that all are verbs for which you can apply the past tense in the gerund form and the "ion's" that tack onto that root word to give you these suffixes. And each of these words that has shown up here has exactly the same set of suffixes that can be applied to it. So I did this thing about prefixes. I went out of my way to find that – so all these words could share that same prefix, "assert," "asserter," "assertings," "asserting." But is there some way that I could take this idea of "corrupt" and "assert" and "insert," and then once I get down to that "t," where they end, is there a way where they can also share their backend parts? That those "ing's" and "ion's," can they be unified, as well? Well, why not? Why not? So here is the first version of what's going to be the DAWG, the Directed Acyclic Word Graph. It takes the idea of unifying the prefixes and now applies it to the suffixes. That there are a lot of words that end in the same letters, as well as start in the same letters. Why not just apply the same kind of unification and sharing on the back end?

So here comes in “adapt” and “adopt” and “based” and “port” and out the back end is “ports” and “portion” and “porting” and “porter” and “adopter” and “adopted” and “basting” and “bastion,” all coming off that sharing all the endings, because they have the same words that are viable and the same connections that are between them. So the idea that all sharing the “er” and the “er” leading in to the “s.” And that same “s,” for example, being shared from “portion” and “baste” and “adopt” and “adopter,” all coming into the same ending of oh, these things that end in “s” are words. Here’s an “s” that is a word for them to share. So it is directed. So it is a graph. So once we remove this – the idea that there’s a single path from the root to any node, we’re no longer a tree structure. So we can get to “t” by going through “adapt” or “adopt” or “port,” and so that no longer fits the definition of what is a tree. It’s become a graph. We can get to these nodes from different places. The arcs go one way in this case. We can’t go back up the tree to find the words in reverse. There are no cycles. So acyclic, right, you can’t just wank around some portion and get this many bananas as you want. And they have been with certain nodes that are marked, in this case, the same way they were in the trie.

This path from the root node to here does represent a word in this situation. So building this is a little bit complicated. I’ll give you a little bit of the idea of the intuition for it, and I will not get too deep into it. But largely the way you do it is you build the trie, so that it doesn’t have the unification of the suffixes; it only unifies the prefixes. And then you work backwards on the suffixes. So in some sense you look at all the words that end, for example, in “s” in the dictionary. And you find all the ones that just end in one “s” that goes nowhere, and you say look, this “s” looks like every other “s” in this thing. All these words that are just plurals or verbs that can have an “s” tacked on. Let’s share that “s.” And then so you take however many s’s there were, 1,000 of them, and you say these are all the same “s.” And then what you do is you look at all the nodes that lead into that “s.” So you’re kinda traversing the graph in reverse, and you say well look, here’s a bunch of people who lead in on an “e.” And if that “e” is a word or not, there might be some group that has “e” that’s also a word, so like the word “apple” and “apples.” But sometimes the “e” is not, where that wasn’t a stopping place, let’s say, for “parties” it wasn’t. So all the ones that are a word you can unify on the “e” that was a word, and all the ones that aren’t are that way can unify on that.

So you just work your way backwards from there. So you say what letters led into that “e?” And so kinda from the end of the words backwards, you’re looking for all the people who can end in an “s,” and end in an “e,” and end in an “ies,” and unify them up, making sure that they have all the same outgoing connections. So for example, “running” has an “ing” at the end. So does “swimming,” but there’s “swimmingly” and there’s not “runningly.” So you have to be careful about unifying things that they would kinda have to have all identical properties from this point down to the bottom of the tree to be unifiable. But it’s a pretty neat process. It’s actually what’s called DFT subset minimization, so if you want to learn a little bit about it, that’s the word to look up. And so if I do that, and I build this as a DAWG, I’m still using the same structure definition here, but what I am doing is unifying suffixes as well as prefixes and that, actually, is gonna change the number of nodes that I need drastically. It goes down to 80,000 from

my initial 250,000. And if you think about the number of words again, there were 125,000 words in the dictionary, the DAWG has just 80,000 words.

So it means, actually, once you start unifying the suffixes, it's like most words don't even have – or many of the words don't even have nodes of their own that are unique to them whatsoever. They just exist kinda as portions of other words that have all been joined together and compressed into one package. So 80,000 nodes total, about two-thirds of the number of words total. So we have ten nodes – ten-byte nodes and 80,000 of them. We are down to under a megabyte. So we've crossed that little interesting barrier of the data structure on disk. The data that was fed into it, those words, was over a megabyte and now, actually, we have created a representation that actually is compressed, that uses less space than the text file did in the first place. And that's really because it is finding all these ways in which words are like each other and sharing, making use of the existing structure in the DAWG to add new words without adding bulk. So once I've done this I have to say that I've made the data structure even less flexible. So to be clear, each of these steps has a cost somewhere else that made the code more complicated.

It also made it so that now that I have all this sharing back here, that when I'm adding a new word, so I go back to look at this picture here, that I go in to add a word and I say oh, actually, the word "portioning" is a word. Well, "adoptioning" is not. So that will necessitate if I go in to add the word "portioning," that I actually have to split it off of its existing unified branch, because if I just tacked on the "ing" on the end of "portioning" there, that it turns out that "adaptioning" and "adoptioning" would suddenly become words, too. So actually I have to be extra careful once I've got it unified this way that when I add new words that I don't actually accidentally add some additional words that I didn't plan on because these paths have been shared. So it requires a little bit more careful management. Again, the goal, though, here, in this case, was to try to build a data structure that was kinda freeze-dried. So I build it off the data structure – the input data file I had, that then would operate very, very efficiently, and I wasn't planning on making a lot of runtime changes to it.

I get this down to this and then the last thing I'm going to do to it is go back to kinda the CS side of it and see if I can squeeze it down kinda in terms of just data structure, what's being stored per node. So I have 80,000 of these. So typically, when you look at a structure, it's very rare that the idea if you change something from an int to a char, so it was a two-byte thing into a one-byte thing, that you're going to have any profound impact on the total size needed. But if you happen to know you're going to make a lot of these structures, tens of thousands, almost 100,000 of them, then it turns out every byte counts. If I can get this thing down from, in this case, ten bytes to eight bytes, or to six bytes or four bytes, then it will cut my memory by quite a factor. So what I did in the final version of it is I used a little technique called bit packing, which is to actually stash stuff in the absolute minimum possible space that I can fit it into, taking advantage of things like, for example, if there are only 26 letters in the alphabet, a character is actually a full eight-bit value, which can hold numbers from zero to 255. And I just don't need that much space. I'm not using the yen sign, I'm not using the parentheses, I'm not using the digits. So having, in some sense, the capacity to store all those distinct characters is

actually a cost I'm paying on every character that I don't need to. So in fact, I squeezed it down by dividing up into the sub-bit level. This something that's talked about a little bit in Chapter 15 if you're at all curious. It's not what I consider core material for 106B, but I'm just sorta mentioning it to kinda complete the picture. Where five bits, given that a bit is either zero or one, that I have five of them connected together, then I have 25 different patterns that I can represent in that amount of capacity there, and that's 32 different patterns, which is enough for my characters, plus a little slack.

I use exactly one bit for is word, rather than a full Boolean. All I need to know is yes or no. I also changed the way I did knowing that you were the last child of a generation, rather than having A say, "there are 12 children here." I just said, go to index 27 and start reading, and one of them will tell you it's the last. So each of them says, "No, I'm not the last," "No, I'm not the last." And then when you get to the last one, it will be marked as "yes," and that's how you'll know – that's where A ends and the next one beings. And so if I can get it down in this way I've taken what was a ten-byte structure down to four. And it turns out there's often reasons why trying to get it down to a multiple of two is actually a critical barrier. That if it were five bytes, it's actually likely to be eight, just because of what's called alignment and padding restrictions. So if were at seven and I squeeze down to six, it might not make any change. And if I squeeze down to five, it still might not make any change. It might be, actually, artificially decided that in each of the structures is going to be eight behind the scenes. But getting it down to four actually tends to be another one of those critical barriers where okay, four actually will be smaller than something that was ten.

And so I take my 80,000 nodes, I get them to be four bytes each, and then I now have a data structure that's about a third of a megabyte in memory, as opposed to the over a megabyte on disk in the text form. And so that's how it does what it does. So if you're going to look at the code, you'll see that there is this complicated bit pack structure that it does look for things char byte using a recursive character-by-character strategy of start at the top, find the matching character, then go to the children index, and then work your way down the children to find the next matching character, and so on to find its way through the data structure. So [inaudible] is where the child index, or for example, the idea that A says my children start at 27 or B starts at 39, and so I just need a number there, but I'm using a slightly smaller than an integer, just to make them all fit very tightly.

So I'll tell you a couple of interesting things about it, actually, before I close this off, which is the lexicon.dat file actually just is an in-memory representation of what the DAWG data structure looks like. And so in fact, it's very easy to take the whole data structure. It doesn't have any pointers in it. Pointers, actually, tend to be a little bit – make data structures harder to rebuild and to dehydrate to disk because they have to kinda be wired up in memory. You don't know what memory you're going to get from new and what the addresses are. This one, actually, doesn't have any pointers. Everything is actually kinda self-contained. It's just one big array of these blocks. And all the information is relative in there. It says go to the index 27, index 45. So you can take that whole block and just write it to disk, 300 bytes worth, read it back in 300 bytes worth and

it kinda is rehydrated without any extra effort. So that's actually what the lexicon.dat file is, it's just a dehydrated form of the DAWG having been build in memory. That makes it very, very fast to load. At that point, though, it's super inflexible. If you want to add a word to it, the way the structure is build, you'd have to kinda add nodes and make sure it wasn't glomming on to some existing words that were nearby, and so it would be a lot of work to actually edit the DAWG in place. In fact, when you ask it to add a word, it actually doesn't edit that one at all. You say I'd like to put a new word in. It says I'm just – this one's so perfect the way it is. I've built this one, I love this one, I'm very happy with it. If you want to put some other words into it, it just keeps a second data structure on the side, an auxiliary one over here, where it sticks the additional words you ask it to put in.

I have to say that seems like a very obvious thing to do, but in fact, I didn't even think of that for years. We had the lexicon. I said you can't use the lexicon to make new word lists or edit them. You can only load this one, beautiful, perfect one. You can't do anything else. And at some point people kept saying, "I wish I could put some words in the lexicon. I wish I could use it for the word list to keep track of the human and computer players." I'm like, "Well, you just can't do it. You don't understand. It's very complicated. You just don't understand my pain." Whatever. Whatever. I had excuses. And then, really, after some number of years of just being a baby about it, I'm like oh, of course, why don't I just keep another structure off to the side? It's an abstraction. It's a word list. You don't need to know what I do. The fact that I keep a DAWG for some of the words and just an ordinary set for some of the other words is really completely irrelevant. And as long as when you ask for words I look in both and I tell you, "Yes, it's there," why do you need to know where I keep it? Okay, so it took me a while to buy abstraction, even though I teach it all the time. It just wasn't totally making sense.

So a neat little thing, just kinda keeping both behind. And so it turns out that I learned about this, actually, from a paper that I read at the – in the early '90s about somebody building a Scrabble player, and they wanted a dictionary that supported kinda finding plays on the board and being able to extend words. And so knowing things about root words and how they extend and prefixes and suffixes was kinda the original motivation. And this kind of data structure is actually really great for any kind of word playing. So you have anagrams that you want to rearrange or words that you want to extend past suffix and prefix that this kind of data structure is really optimized for doing that kind of traversal and kinda leveraging the existing patterns in words to save space while doing so. So it was a really fun little project to do. It was kinda neat to think about. Even though today, if you needed a lexicon, you could just use a set. It's fast enough. It takes a ton of memory, but you know what, memory's cheap.

So this is not the kind of thing you would need to do today except when you were in some kinda embedded, low memory, very, very tight processor environment, but it still makes for kinda interesting sort of artifact to think about well, how you get there and what kind of things you can use. So Wednesday we will talk about some big picture design, some kinda wrap stuff, and I will see you then.

[End of Audio]

Duration: 52 minutes

ProgrammingAbstractions-Lecture26

Instructor (Julie Zelenski):Hi, there. It's Wednesday. We are coming to the end of this fine quarter. So let me tell you a little bit about administrative stuff that's from here to the end so we're keeping track of all these things. Then we'll go on and talk about what we're going to do today.

Today, I'm leaving to go out of town to go to a conference in Portland, a special interest group on computer science education. So all the CS teachers in the country get together and talk about how to torture their students for a couple days in Portland. I will be leaving at 4:00 p.m. to head straight to the airport. So if you need to see me, and you're hoping to come to today's office hours, please come at the beginning there so I can take care of what you need before you leave.

I will be out of town, but hopefully email accessible. If you needed to talk to somebody in person, Jason is the guy to follow up with if you need to see something before I come back at the end of the weekend.

We will meet on Friday and have an optional guest lecture, but I think it's actually very valuable. It's Keith Schwartz, who is the instructor for the cs106l lab class. So those of you in the lab class already know what a superstar he is. Those of you who haven't been in that class will get to see what he's all about. He's going to come and talk to you about C++ without 106. So we teach a certain variety of C++. We have a subset of language we pick. We have a set of libraries and tools, and we use certain features of that.

Our pedagogical goals, but there's a lot more to C++. So Keith is going to give you an overview on Friday. It's kind of the whirlwind tour of if you were out of 106 and you were staring to use C++ on some project you're doing or some further classes or some internship you have, what things do you need to pick up to complete the picture of how the more professional standard C++ looks as distinguished from 106b's version.

So I think that's a very useful and very practical thing to take away from this. So if you actually are available to join Friday, I think you'll get a lot out of that. Then other things that are happening, Pathfinder's coming in this Friday, so the last bit of heavy travail you have to do for us is get that assignment in by this Friday. There are no paper copies do, so all e-submit. We won't be doing the interactive grading in that sort of form. All we need is your electronic submission to get our grading done on that.

Our exam is a week from this Friday, so the very end of the exam week, we will be meeting in Krezgi auditorium, which according to the registrar's database, seats 492 students, which is enough for all of us plus some. That's over in the law school. That is the place for 106b fun to happen next week during exam week.

I think administratively [inaudible]. So what I'm going to do today is I brought a cup filled with candy. We'll start with that because that's going to be helpful. What I'd like to do is it's just a couple bits of loose ends, some things that I wanted to touch on that have

come and gone this quarter. I maybe wanted to pull them together and reach a bit of closure on. Then hopefully, if we have time, we'll move on to what happens after this. If you liked 106b, what do you do? If you don't like 106b, what do you do? What kind of options and future opportunities are there after this ends? I'm going to talk a little about some of the things that [inaudible], but I do want to talk about the final just for a second because it is kind of a last opportunity for you to show us your mastery of the material and get the grade that you've been working hard for all quarter. It does carry a certain amount of weight. 30 percent of the weight in the default system is actually placed on the final, and more of it can move to that if you have a better showing on the final than you did in the midterm. We're happy to move some of that weight over so it could actually account for even more of your total grade.

So there's a lot riding on it, and you want to be sure you come ready to show us the things that you know and that you have comprehension and mastery over. To note, though, it is going to look a little bit different than the midterm in terms of its emphasis. We are going to see a lot more of this nitty gritty dynamic data structure, Linklis, trees, graphs, pointer-based structures with the inherent complications in dealing with that kind of dynamic structure. So bringing your best game to this exam is certainly going to pay off here.

The coding that you will see will look a lot like what's in the assignments. If you already had a chance to look at the practice exam, you'll see that there are themes there that echo the things you did on the work. You want to try to make sure that's we're connecting to the effort you've put in all along. There will actually be some opportunity to think a little bit at a higher level about making and analyzing choices for design and implementation. I'm going to talk a little bit about that today because I feel like at any given moment, we're talking about how we might implement a stack and how we might implement a queue. Sometimes it helps to step back a level and think a little bit about the choices for the bigger picture of when to use a stack and queue and how to make decisions in the large about those tradeoffs that are intelligent and grounded. So I'll do a little bit of that today, and then some of that has been built up all quarter long.

It is open book and open notes. You can bring all your printouts, your readers, sections, all those sort of things. For some people, that translates to this idea that, well, I don't need to prepare because I'll have all my stuff there. If I need to know something in the exam, I'll just go look it up. That's certainly true for looking up details. You want to remember how this works or you had a piece of code that you think was similar. It could help you to brainstorm what you're trying to do this time. But you're not likely to have time to learn new things n the exam. So if there's something that you really feel you have not gotten your head around, the investment upfront before you get into the exam is where to get that understanding in place rather than trying, in the exam, to be flipping through chapter ten, trying to learn something. It may not really play out.

I highly recommend practicing on paper in an exam-like environment. So really working the way you will be working in the exam to give yourself the best, consistent prep for what's happening there. Then some people have asked about extra problems, just wanting more of them to work on. I'll give you a couple places where you can look for things.

One is that cs106 is being offered this same quarter, and they are just the honors version of our class. So they have some slightly notched up problems. But their practice and real midterm are posted on their website as well as their practice final. They have very similar coverage. They did some assignments that were like ours, some that were a little bit different, but still go places to kind of just grab actual exam problems with their solution.

The chapter exercises and section problems often are old exam problems or eventually became exam problems. So they kind of come and go in terms of being used as examples or used as exam problems. So they're definitely places to pick up extra mileage.

Okay. So let me tell you a little bit about – any questions about the final?

Okay. Let me talk about this line just for a second because I feel like we have touched on these themes again and again, but maybe it's good to have at least one moment of trying to pull this together and think a little bit about the closure and the big picture of all these things. A lot of what we're talking about in the beginning was we were giving you these abstractions, these cool things to do stuff with, a map, a stack, a queue, a vector, and problems to solve where those abstractions have a role to play. Then we spent a long time saying, okay, now that it's our job to make those abstractions work, what techniques can we use? What algorithms and data structures will help us manipulate and model those structures in efficient ways.

We certainly spent a lot of time talking about runtime performance as though that was maybe a little too much in your mind to think that was the only thing that really mattered. How fast can we make something? O of N is better than N squared. Log N is better than N, driving to these holy grails of bringing the time down.

It's certainly a very important factor that often – the difference between something being linear and something being quadratic is a matter of it being tractable at all for the problem at hand. You cannot sort a million numbers in a quadratic sort in nearly the same time. For large enough data sets, it'd be completely unfeasible. So it is important to have that big picture, to be thinking about it and know about those tradeoffs, to understand issues of scale here. But we also did some stuff on the homework about actual empirical time results, which is another way to bolster our understanding. The big O tells us these things, but what really happens in real life matches it, but not precisely. Those constant factors, we threw away, and other artifacts that are happening in the real world often give us new insights about things and challenges that we're facing.

We also need to think about things like mix of expected operations, having some be slow at the consequence of other operations being fast. It's often a tradeoff we're willing to make. On the editor rougher, we drove toward getting all the operations to be fast. In an ideal world, that's certainly the best place to be, but it's also a matter at what cost in terms of code complexity and memory used and effort put into it. It may be that we're willing to tolerate some operations being less efficient as long as the most commonly used ones are very streamlined.

We talked about these worst cases and degenerates about knowing when we can expect our algorithms to degrade and whether we want to do something about that or choose a different algorithm or provide some protection against those kind of cases coming through. To a lesser extent, we already talked about memory used, how much overhead there is. Seeing that we went to a pointer-based Linklis structure, we added the four byte overhead for every element. We went to an eight byte overhead once we got minor [inaudible] and maybe even a little bit more if we had balance factors included.

Those things ride along with our data, increasing the size and the memory footprint of it. So there's a certain amount of overhead built into this system. There's also this notion of excess capacity. To what extent do we over allocate our data structures early, planning for future growth. In some senses, save ourselves that trouble of enlarging on demand. It's preparing for a large allotment and then using it up when we find it. Then when we do exceed that capacity, having you do it again.

So where do we make those tradeoffs about how much excess capacity and when to do the enlarging? It has a lot to do with what our general memory constraints are. I would say that memory has gotten to be quite free in recent processors and computer systems. You don't really think about 100 bytes here, 200 bytes there, 1,000 bytes there. Even megabytes, you can talk about as thought it's a drop in the bucket. So this is something that isn't given as much weight nowadays as it was ten years ago when there were a lot more constraints. But with the move for a lot more imbedded device programming, and looking the iPhone or cell phones or things like that where they don't have that same liberty to be wanton about memory, it's come back to being a little more careful and keeping an eye on it.

There's also and issue here which trades off a lot with runtime, which is redundancy versus sharing. Having two different ways to get at the same piece of data often will provide you with quick access. For example, if you have a card catalogue for a library where you can look up by author, you can look up by title, you probably are maintaining two data structures. One that manipulates them, sorted by title, one that manipulates them sorted by author. They're both kind of looking at the same books in the end. So maybe there's this set of pointers in one map over here, indexed by author, another keyed by title over there.

That means that I'm repeating all my data. Hopefully I'm sharing it with a pointer, so I'm not actually really repeating all the data, but I actually have two pointers or potentially two or more pointers to the same book, depending on the different ways you can get to it. That gives me that fast access. I want to know all the books written by Leo Lionni, I can find them easily, as well as finding all the books that have King in the title. I don't have to do searches on a data set that's been optimized for only one access. Definitely, tradeoffs were more space thrown at it to get at that fast access for different ways.

Then this last one is one that I had mentioned along the way, but I think it's one that's easy to overlook, which is to get very excited about how fast you can make it, how small you can make it, how fancy it is. It has a real downside in terms of how hard it was to

write. Typically, when you go to these fancier strategies that are space efficient and time efficient, you had to pay for it somewhere. It wasn't for free that the easy, obvious code would do that. It tends to actually be complicated code that may use bid operations, that probably uses more pointers that has kind of a density to it that means it's going to take you longer to get it right. It's going to take you more time to debug it. It's going to be more likely to have lurking bugs. It's going to be harder to maintain.

If somebody comes along and wants to add a new operation, they open up your code, and they're frightened away. It might be that that code will never get moved forward. Everybody will just open it and close it immediately and say, whatever. We'll make it work the way it is, rather than trying to improve it or move it forward because it actually is kind of scary to look at.

So thinking about what's the simplest thing that could work. There's this movement that I think is pretty neat to think about. There's this idea of the agile programming methodology. It's based on the idea that rather than planning for building the most super stellar infrastructure for solving all the worlds problems. You're about to write a chess program. Actually, something that's even simpler. You're going to write a checkers program. You say, I'm going to sit down, and I'm going to design the pieces and the board. Then you think, hey, checkers is just one embodiment of this. Why don't I try to make the uber board that can be used for strategic conquest or for chess or Chinese checkers. You could have these hexagonal things.

You start building this crazy infrastructure that could handle all of these cases equally well, even though all you really needed was checkers, a very simple, 2D grid. But you imagine the future needs way in advance of having them, and then you design something overly complicated. Then what happens is you get bogged out of that. It never happens, and the project dies. You lead a lonely life by yourself, eating oatmeal. I like oatmeal.

So one of the mottos is design the simplest thing that could possibly work. I think that's a neat gift to yourself to realize that this simplest thing that could possibly work, that meets the needs that you have, that has a decent enough [inaudible] and memory constraint right there. Meets your needs in the simplest form of that. It's going to be the easiest thing to get working and running. If you later decide you want to go fancy, you can swap it out using the principles of distraction and [inaudible]. It should be that it's independent when you decide you need that fancier thing.

I've got a couple questions here because I thought this would just help us frame a little bit of thinking about these things. These are questions that I'm hoping you can answer for me. That's why I brought cup full of candy. I'm prepared to offer bribery for people who help me answer these questions. So let's talk about array versus vector. So the sequels [inaudible] builds and array is what is behind the scenes of a vector. Then the vector adds convenience on the outside of it.

Early on, I had said, once I tell you about arrays, put that back in your memory, and just use vector instead because everything array does, vector does nicely and cleanly and adds

all sorts of features. Somebody give me a list of things that vector does much better than array.

Way in the back.

Student: Bounce checking. Instructor:

Bounce checking. What else? Come on. A blow pop is on the line. I need to interrupt your – help us out. You've got bounce checking. What else?

Student: More? Instructor:

I want more. You've got nothing else for me?

Student:I don't even want a blow pop. Instructor:

You don't want a blow pop? All right. I'm going to give your blow pop to somebody else. Right here. I'm going to give the blow pop to you. Tell me what makes vector good.

Student: It has operations that shuffle elements for you. Instructor:

Yeah, it does the insert, delete, moving stuff down. Also, it does the growing. You keep adding, and it just grows. [Inaudible] don't do that. You want a [inaudible] to grow, you grow it. You take the pointer, you copy stuff over, you delete the old one, you rearrange things. You do all the work.

So vector buys you kind of array with benefits. So it seems like, then, you could kind of take away – the naïve point would be like, well, just use vector. Always use vector. Vector's always better than array. Is there ever a situation where array still is a pretty valid choice? What is it array has that vector maybe doesn't?

Student: Well, when you add things dynamically to the vector, it actually doubles the size [inaudible] increasing it by one. If you know the exact size, and you know it's not going to change, you're using [inaudible]. Instructor:

That's it exactly. One of its big advantages. If you know you only have – let's say you have a very small array planned for. The one on the practice exam was, if you're doing a calendar where you had 365 days in a year, and you're keeping track of the events on a particular day, you know how many days in the year. There's never going to be any change in that. Well, leap year.

But you know that, and you know that in advance. So why fuss with a data structure that does all this growing and shrinking and has some overhead associated with that when, in fact, you know there's exactly this number. That's all you need. In fact, it's almost a little bit more convenient to set up. You can just declare it that size, and it's ready to go. With the vector, you have to go through and add all the elements and stuff like that.

So there are a few situations where you might just say, I don't need the advantages. It adds some overhead that I don't want to pay, and I'm perfectly happy making it a small fixed array. You'll still lose bounce checking, but if you are being careful, maybe you're comfortable with making that tradeoff. By losing bounce checking, you actually are getting a little bit of the speed improvement back. The bounce check does actually cost you a little bit on every vector access that an array would not actually pay.

That's really in the noise for most programs. I hesitate to even mention it, to get you to think it matters, but it is part of the thinking of a professional program. Sometimes that may be the thing it comes down to, making your operation constant factors streamline enough for the operation you need.

So I put here stack and queue versus vector. Given that stack and queue are really just vectors in disguise, and they're vectors with less features, vectors that can't do as much. One of the implementations for stack and vector could easily be just layered on top of a vector. Why is it good to take away things?

Student:[Inaudible] tampering with contents of [inaudible]. Instructor:

Yeah. So it's enforcing discipline on how you use it. Stacks are lifo, queues are fifo. The way things go out, they way they go out are very rigid. By using stack and queue, you are guaranteeing that you won't go in and accidentally put something at the head of the line or remove something out of the middle of the stack because you just don't have access to that. It's been tidied up and packaged up into stack, which has push and pop. queue, which has MQDQ.

No other access implied or given, totally encapsulated from you. So in fact, it allows you to maintain on a discipline that otherwise the vector didn't strongly provide for you. Question?

Student:[Inaudible]. Another thing you could do is you could optimize a stack or a queue to be very good at the only active things you're doing, which is pushing and popping or RQDQ. Instructor:

I think that's worth a smartie, don't you? I think that is. Exactly. So by restricting what you have available, it actually gives you choices in the implementation that wouldn't have worked out as well, [inaudible] full array of operations. It's exactly true. Stack and queue only adding from one end mean that things like a Linklis, which isn't a good choice for implementing the general case of vector, are perfectly good choices for the stack and queue case.

So giving ourselves some freedom as the implementer. So information about how it's being used can pay off. It also means when you read code – if you see code that's using something, call the stack or using a queue, you immediately know how it works. You could say, here's this search that's actually stuffing a bunch of things into a stack. I know they're last and first [inaudible]. If you see them using a vector, you have to verify where

they put them in the vector. Where do they pull them out? The code doesn't tell you the same things. It doesn't tell you the same story as it does when you see a word that has a strong meaning to it, like stack and queue.

Generally, these [inaudible] to make about sorting things or not sorting things. We spent a long time talking about sorting and how various algorithms approach the sorting problem and what they come up with. I have a question here. What sorting algorithm is best? Is there an answer to that question? Why would I ask a question that has no answer?

Student: It depends on what you expect it to be giving you. Instructor:

It depends. That's a great thing. It depends on what you expect it to give you. How big is it? What are the contents of it? Is it random? Is it sorted? Is it almost sorted? Is it reverse sorted? Is it drawn from a range of one to 1,000 values, or is it drawn from one to three values? Huge array that has just one, two, three in it might necessitate a different approach than one that has one to max sine in it.

So there is no answer to that. It is depends. What do you know? If you don't know anything, give it to someone else. If you don't know, there's certain things that perform well. You know in the average case, you don't have degenerate behaviors. For example, [inaudible] is a great sort for the general case of an N log N that doesn't use any space or any extra space the way word sort does. It doesn't have any degenerate cases. For an unknown set of inputs, it works pretty well.

If you happen to have a little bit of data, it's almost sorted. Even Barack Obama's not favorite bubble sort has a chance of being in the run because it's already sorted.

Is it worth sorting? Calculating the payoff, we had a problem on one of the section exercises once. I think it's a really good one to think through and revisit. This idea of certain operations, many operations are much more efficiently implemented if the data's sorted. You can think of them off the top of your head. Oh, you want to have the minimum? If it's sorted, it's very easy. You want the maximum? Very easy if it's sorted. You want to find the median. Also very easy if it's sorted. You want to find duplicates. Once they're sorted, they're all lined up together. You want to find the mode, which is the most frequent element. Also easier if it's sorted because they'll all be groups together in a run. So you can do a linear pass, counting what you see to find the longest run.

If you had two arrays and you wanted to merge them or intersect them to find their common elements, it's much easier if they're sorted. All these things that are hard to do when the data's in a random order actually have much more streamlined algorithms once you invest in sorting it. So there was a problem that was like, how do you know it's worth doing that? If you only plan on writing one merge, and you could do it in N squared, is it worth it to sort it, N log N, so you can get an O out of merge?

So it has a lot to do with how often you plan on doing the merge. How big is your data set? What are your constraints on the times there. It is interesting to think about that

relationship. It isn't for free, but potentially, it's an investment by sorting it so you can do a lot of fast searches later.

This one actually is interesting because they actually solve a lot of the same problems, and they differ in one small way. If you had a set – so set is being backed by a balance by a [inaudible] string that actually is a sorted data structure internally – versus a sorted vector. So if your goal were to keep track of all the students at Stanford, and you wanted to be able to look up people by name, then the contains operation of the set is actually doing these minor ray search on assorted vectors using the same path, but working through a vector.

So we can make the searching operations, like contains and other look-up based things, algorithmic in both cases. So what advantage – this sort of vector, where I just used the interchangeably, is there something one can do that the other can't or some advantage or disadvantage that's assumed in that decision?

Student: You don't have duplicates in a set, but you can have duplicates in your sorted vector. Instructor:

Yeah, so the set has another concern with just how it operates. You can't put duplicates in, so if you have two students with the same name, then in the set, it turns out you're all coalesced down to one. Your transcripts all get merged to your advantage or not.

What else does set buy you that vector doesn't? How hard is it to edit a sorted vector? You want to put something new in there? You're shuffling. You want to take something out? You're shuffling. The movement to the pointer-based structure there, the research stream behind this, is giving us this editability of the data structure. The convenient logarithmic time to remove and add elements using the pointer wiring to do that, that vector doesn't have.

That tradeoff is actually one that situation may be that you just don't do a lot of edits. That's actually a very common situation. The sorted vector happens to be a very underutilized or under appreciated arrangement for data because for most things that are getting a lot of edits, having a linear time edit isn't going to be a painful consequence. You're doing a lot of searches, and you're getting the finer research there, where you really care about it, with no overhead. Very little overhead. A little bit of excess capacity, but none of the pointer-based trouble that came with the set.

So the set actually adding onto that another eight bytes, potentially even a little bit more, of overhead on every element to give us that editability that maybe isn't that important to us. So if you're looking at that thing and saying, I'd like to be able to have a sorted structure for both the set and the sorted vector, let you access the elements in order. The interrater of the set goes in sort order, going from zero to the end. The vector gives me order. I can browse them in sort order. I can search them using sorted. Where they differ is where does it take to edit one? The one that invested more in the memory had faster editing.

The P-queue. This is kind of interesting. The P-queue is also a sorted structure, but not quite. It's a much more weakly sorted structure. So if you're using the [inaudible] heap like we did on the P-queue [inaudible], it does give you this access to the maximum value, and then kind of a quick reshuffling [inaudible] the next and so on. But it doesn't really give you the full range. For example, if you were looking for the minimum in a P-queue, where is it?

It's somewhere in the bottom. It's in the bottom-most level, but where across the bottom, no knowledge, right? It could be at any of the nodes that are at the bottom of the heap. So in fact it gives you this access to a partial ordering of the data. In this case, it's relative to the prioritization, but not the fluid form of sorting the way a vector is. For example, if P-queue doesn't give you things for finding duplicates or finding the minimum or doing the mode it's not actually an easy operation because you have it in a heap form. I can get to the max quickly.

So if I cared about pulling them out in sorted order to browse them, I could stuff things into a P-queue and then pull them out, but what's interesting about that is it requires destroying the P-queue. The P-queue isn't really a place you store things and plan on using them again. It's the place you stick things in order to get them out in an order that's only value is in the de-queuing of them, pulling them out.

So finding the most promising option to look at in a search or finding the most high-priority activity to take on or something. But it doesn't really have the – if I wanted to be able to look at all the students in my class in sorted order, it's like, well, I could stick them in a P-queue and then pull them out, but it's sort of funny. I have to put them in the P-queue just to pull them back out. It wasn't a good way to store them for that browsing operation to be easily repeated. If I put them into a sorted vector, I could just keep iterating over it whenever I need to see it again.

So a lot of the things that it comes down to is having a pretty good visualization of what it is that going to a pointer-based data structure buys you, and what it costs you, relative to the continuous memory. That's one of the tensions underneath most of these things. It's like, putting more memory into it by virtue of having a pointer wire as opposed to a location arrangement. There's no information that's stored to get you from a race of zero to a race of two. They're just continuous in memory. So it saves you space because you only have to know where it starts and access continuously for that.

This pointer means there's more kerversals. There's more memory. There's more opportunity to air, but it gave you this flexibility. It broke down this continuous block, that's kind of a monolith, into these pieces that are more flexible in how you can rearrange them. So thinking about these things is an important skill to come away from 106b with that, yeah, there's never one right answer. You're investing to solve this other problem, and there's going to be downsides. There's no obvious triumph over all. It's fast, it's cheap and it's small and easy to write solutions out there.

So I put up here – I had to whittle this down to the five or six things that, at the end of 106b, I want you to feel like, that's what I got. That's the heart of the goal. The MVPs – the most valuable players of the 106b curriculum here. You're looking back at it, and you go, where did I start, where did I end up, and how did my knowledge grow? I'm hoping these are the things that stand out for you. That was a real growth area for me.

One is this idea of abstraction. That was stressed early on. Here's a stack, here's a queue, here's a set. They do things for you. We don't know how they work yet, but you make cool things happen with them. So you solve maze problems and random writers and do neat things with dictionaries without knowing how they work. So you're leveraging existing material without knowledge of how it works. It helps to keep the complexity down. You can solve much more interesting problems than you could without them.

Try to go back and imagine the Markov random writer problem. Now imagine doing it without the map in play. So you had to figure out that given this character, go look it up and find a place to store it, and then store the character in some other array that was growing on demand, as you put stuff in there. You would never have completed that program. You would still be writing it today, having pointer errors and whatnot behind the scenes.

Having those tools already there, it's like having the microwave and the food processor, and you're about to cook instead of having a dull knife and doing all your work with a dull knife and a fire. You actually have some real power there.

Recursion's a big theme in the weeks that followed that. How to solve these problems that have this similar structure, looking at these exhaustive traversals and choice problems that can be solved with backtracking. Having this idea of how to think about thing [inaudible] using the recursive problem solving and how to model that process using the computer was a very important thing that comes back.

We look at algorithm analysis, this idea of making this sloppy map that computer scientists are fond of, talking about the scalability and growth, knowing about curves and the relationships and how to look at algorithms. The more formal analysis. Still, in this case, there's still quite a bit more formulas to this than we really went into. If you were go on in CS and look at it, you will see there's quite a lot of rigor that can be looked at in that area.

We were getting more an intuition for how to gauge things about algorithms and their growth. Then this backside, all about implementation strategies and tradeoffs. So now that we've benefited from those abstractions, and we've solved some cool problems, what's it like to actually be the person building them? What are the techniques? What does it take to wrestle a pointer or a Linklis or a tree, heap graph into doing something really cool for you. How does that inform your information, what you expect about it and the coding complexity that was inherent in achieving that result?

Hopefully in continuing with the theme that 106a started you on was this appreciation for [inaudible]. In the end, getting code that works, that's ugly, just isn't, hopefully, what you're satisfied with. You want to approach problem solving to produce beautiful, elegant, unified, clean, readable code that you would be happy to show to other people and be proud of and maintain and live with. The kind of work that other people you work with would appreciate being involved with.

Things that are not so much – I don't think these are the most valuable pointers, but what came along the way, the idea of pointers and C++ being part of the – C++ and its pointers, they're one and the same. We choose [inaudible] for a language, so as a result, we get to learn some C++ because it's so heavily invested with pointers. We also have to do some practice with the pointers to get our jobs done. I think of those as being off to the side. I don't think of those being the platform of things I worship. They are things we need to get our job done.

I did put a little note about pointers here because there were quite a lot of comments on the mid-quarter evals that said, pointers, pointers still scare me. They make me crazy. I don't know enough about pointers. Here's a fact for you. You'll probably never feel like you know enough about pointers. Today, tomorrow, ten years from now, you'll still kind of feel like there's some real subtlety and trickiness in there.

They're an extremely powerful facility that gives you that direct control over allocation and deallocation and all the wiring and sharing that is important to building these complicated data structures. But there are so many pitfalls. I'm sure you've run into most of these, if not all of them at some point, where you mishandled a pointer, failed the allocator, deallocate, and the way it gets reported, whether the compiler or the runtime error that you see is helpful in sorting it out is a real cause for long hours to disappear in the name of mastery of this.

If you're still wary of pointers, you probably should be. Think of this as your first go-around. We're still at the journeyman's stage for programming here. So although there were pointers in 106a, they were very hidden. This is the first time you really see it and are exposed to it and trying to get your head around it. If you go on to 107 and 108, you'll just get more and more practice. It will become more solid as you keep working your way through it, but at this stage, it's okay and commonplace to feel still, a little bit – you see the star, and you take a double take. That's just where you should be.

It's just like in the Chinese restaurants. They put the little red star by food you want to be careful of. That's why they put the star for the pointer.

I also wanted to zoom out farther, a decade from now, when you are thinking back on the whole of your education and what you learned. Do I want to really remember how to type in keyword and the sequel plus generic template facility? No. If you code in C++ every day, you'll know that. If you don't, you can forget it. It's long gone.

What I hope you'll carry away from you is a couple concepts that broaden outside of CS. One is this idea of algorithmic thinking, how you approach something logically and step-wise and draw pictures and analyze it to understand the cases and to debug when things aren't working well. I think that's a skill that transcends the art of computer programming into other domains, any kind of thinking logically and problem solving and analyzing in that way.

I also hope that some of the experimental things we did with the sort lab and the P-queue and some of the big O and thing we tried to do in class together help you to [inaudible] back of the envelope calculations. This is something I'm a little bit of a – it's a pet issue of mine, which is, I think, in the era of computers and whatnot, really easy to get distanced from numbers and start thinking, I just punch numbers into formulas and one comes out. It must be the truth. Not having any intuition as to whether that's the right number, whether that number makes sense, whether it's grounded, and whether you actually are [inaudible] or so off because you've actually made an entry error, and you don't even notice it.

So becoming comfortable with this idea of using numbers to drive things, being able to take some time trials, do some predictions, do some more time trials, match those things up, do a little sketching on your numbers and see the things that are working out. So being comfortable having the math and the theory reinforce each other and not get too distanced from the fact that those numbers do play out in the real world in ways that are interesting. Don't just let the numbers themselves tell the story. There really is more to connect it up with.

This idea of tradeoffs. You may not remember how you can write a Linklis or how the hash table does the things it does and how to get one working, but hopefully you will take away this idea that the answer to most questions begins with the phrase, it depends. If I say, is it better to use a hash table or [inaudible] search tree, the answer will be, it depends. There's no, it's always better to use this sort. It's always wrong to use this approach. It's, it depends. Do you have the code written for that, and it works well, and you don't care how long it takes? Then bubble sort actually could be the right answer. Barack Obama not withstanding.

But knowing whether memory is at premium, time is at premium, what your mix of operations are, what language you're working with, what program expertise you have, what timeline you have for developing it all can play a role in deciding what choices to make and what strategy to pursue. So being very comfortable with this idea that it's about gathering information before you can commit to one strategy as opposed to it's always going to be this or that.

So that's the 106b thing. I have a couple slides here on things that happen after 106b. I'm just going to go ahead and show you. This would be a great time to ask questions. In particular, if there's anything you're curious about. Where do we go from here, and how do we make good decisions about things that follow?

The most obvious following courses from CS – if you took 106b and you hated it, I'm very sorry. Hopefully the scarring will heal over and you'll lead a productive and happy life elsewhere. If you did like it and you think, I'm kind of interested in this. What do I do next to explore further? Where else can I go with this? I put up the intro courses with their numbers and names, and I'll talk a little bit about what each of these does to give you an idea of what opportunity it offers to you.

The obvious follow onto the programming side – 106a and b are programming courses, building your programming mastery. 107 follow in the same vein. We typically call those systems courses in CS nomenclature. That means practical, hands on programming, getting more exposure to the system.

So 107 builds on 106b's knowledge in a class called programming paradigms. Part of what it's trying to explore is different languages. So you actually look at the language, C, kind of stepping back to old school. You look at the language scheme, which is a functional language that was developed for AI purposes that has a long history coming from the math side of things. Probably looking at some scripting and flexible [inaudible] like Python. We kind of mix them up a little bit, so it's not exactly the same things, but looking at other languages, other ways of doing things.

The other component of 107 is also looking at some more low-level things. What happens? When you pass your code off to the compiler, what is it doing? What kind of analysis does do of your code? How does it actually generate machine code? What happens in the machine layer, understanding some more of the performance implications and how things are expressed. Getting a better understanding of pointers actually comes up because there is a certain amount of low-level coding that's being explored in this class.

So building on the programming skill as we move toward a Unix Linux-based environment. So away from the Mac and Windows. Just continue building larger programs with complicated features to grow your mastery. Internally, we call this class programming maturity, even though it's not its title. We see it that way in the sequence in terms of just growing you into a more versatile programmer, seeing different languages, different things and getting more understanding of what happens under the hood.

It is a five-unit class. It's typically offered fall and spring. Most people thing it is about as much work as 106b. Some a little more or a little less in different parts of the quarter, depending on what your background is and how 106b worked for you. In a model, I think most people think of it as being – would you guys say 107 about as hard as 106b?

Some ways, it's harder. Some ways, it's not. Maybe that's the deal. The 108 class which follows onto 107, but that prerequisite is particularly strong. It's a Java class. So moving in the other direction. Rather than moving down an extraction, moving up on the extraction, looking at more large-scale design, modern programming technique, using object-oriented facilities, thinking about large-scale programming. You do a big team

project in the end, so a three week, three person, massive effort that is the first really big scale thing that you'll be exposed to in the lower-division curriculum.

Student:[Inaudible]. Instructor:

Sometimes in 107l, which has some relationship to the 106l in the way that it's more hands-on, more C++, I don't know whether it will be offered this spring or not, but if you look in Access, it will tell you at least whether we have it tentatively on the schedule. I think that comes and goes with the interest from the students and the availability of someone to teach it. Student:

[Inaudible] can we watch [inaudible]? Instructor:

Yeah, 107 and 108 are offered on TV usually once a year. 107 is offered on TV this spring. The one in fall is not taped. It might be that the old lectures from spring are somewhere that you could dig them out, but I actually don't happen to know for sure. 108 is on TV, I think, in the fall, not in the winter. So the fall lectures are probably still around for 108, so you could take a look at them.

You could certainly look at their websites, and there's often a lot of old materials and handouts and stuff that gives you at least some idea of things that have been covered in the most recent offering of it. Certainly showing up in the first week is one way to get an idea of what you're in for.

108 is a four-unit class. Because it's in Java, there's certain things about Java, the safety and the attentiveness of the compiler, makes the error handing a little bit easier. So I don't think most people think of 108 as quite as intense as 107 or 106b, but that project at the end has quite a reputation as being a real time suck. So it has lots to do with being scheduled and motivated and working through that team project at the end. I think that's where most people find the big challenge of 108.

Then on the math side, in the nomenclature for CS, we have this idea of theory, introducing you to more the formalism for big O and discrete structures and doing proofs and thinking about logic. So we have a 103 a and b and a 103x, which is a combined, accelerated form of a and b, which serves as math classes. They are math classes for introducing the kind of mathematics that are important for computer scientists. So that is for people who are thinking about a CS major, another good course to get in early. Pretty much everything in the upper division of the CS major has these things layered down as prereqs. So these are the intro courses that serve as the gateway to prepping you to go on, and then looking at things in a more topical way.

Looking at networks or security, HCI or artificial intelligence layers on a grounding in theory and programming that came from the intro courses.

So that said, we're in the midst of a really serious curriculum revision, which is on the table and being designed right now, and was voted on by our faculty and improved and

has to go through the school of engineering and get approval. Then there's going to be a little bit of jostling as everything gets worked out. But started next year, you're going to see some changes in the department. Some of these courses are going to get morphed, and they won't be exactly the way they are today.

As a Stanford undergrad, you actually have the option of graduating under any set of requirements that are in play any time you're an undergrad. So you could graduate under this year's requirements, or you could wait. I think for most, the right answer's going to be wait because I think the new arrangement is very much more student-friendly. It has more flexibility in the program.

If you look at the CS program as it is, it has a lot of pick one of two, pick one of three, pick one of two, where you're very constrained on what you can choose. The new program is going to be more track-based where you pick an area of interest. You say, I'm really interested in the graphics. You do a little depth concentration in that, and then just have room for a very wide selection of electives to fill up the program as opposed to a smorgasbord where we forced you to pick through a bunch of different areas.

So for most students, I think it gives you flexibility that actually gives you the options to pick the classes that are most interesting to you and get the material that you want. It's a growing recognition of the fact that CS has really broadened as a field. Some say that our major looks the same way it did ten years ago, a classic definition of here's these seven things that all matter. The things aren't seven anymore. There's no 15 of them, 25 of them.

If you look at the spectrum of things that computer science is now embracing, we can't say, you have to take one of all these things without keeping you here for eight years. So we are allowing that our field's footprint has gotten so large that we need to let you pick and choose a little bit to get the slice that seems most appropriate for where you're headed.

So you'll see more about that. If you are thinking about being a CS major and have not yet declared, I would advise that you add yourself to what's called the considering CS list, and you can get to this from the CS course advisor's page, which, off the top of my head, I think it might be CSadvising, but I'm not positive. I'll put the link on our webpage, when I remember what it is.

It is a mailing list. So we have a mailing list of the declared undergrads, but we also have a list for people who are at least potentially interested in the major and just want to be kept up to date on things that are happening. There's going to be a lot of activity on this list in the upcoming months as we publish the information about the new curriculum. So you'll know what's coming down the pipes so you can make good decisions for preparing for what will be [inaudible], what the transition plan will be as we move into the new courses and how to make sure you land in the right place when all is said and done.

Then I put a little note about other majors that exist. CS is the home base for people who are solving problems using computers. It spans a large variety of people doing a lot of different things. Then there's a couple other majors that we participate in, in the interdisciplinary sense, that UCS is part of a larger context for doing things. So computer systems engineering, which is between us and EE, which is a little bit more about hardware.

The symbolic systems program, which is run by linguistics and includes cooperation from psychology, philosophy, communications, CS looking at artificial intelligence and modeling human thought and understanding intelligence as expressed in that way. Then there's a mathematical computational science program that's homed in the statistic department that is joint between math, computer science and the MS and E. It's more of an applied math degree. So taking mathematical thinking and using computers and statistical things to analyze and think about things. A lot of financial people end up being drawn to that particular one or people who are doing risk analysis or that sort of combination of things.

But where CS has a role to play in all of these because it's a great tool for solving these kind of problems. So different ways to get at different pieces of the major.

I've put a couple notes of things that I think are worth looking into in case they, at some point, are the right thing for you to look at. Section leading, so joining the 106 staff and shepherding the next generation of students through the program is a great way to strengthen your own knowledge. There's nothing like making sure you understand recursion as to try and explain it to someone else who doesn't understand it. I can really do a lot for the personal connection to you and your craft as well as just being involved in a great community. The section leaders are a really wonderful, dedicated group of students who are fun to be around and great people to make friends with. So I highly encourage you to take a look at that.

We interview for that every quarter, so usually about week six, we put out applications and bring people in for interviews. We're always looking for new people to join on a quarterly basis.

There are research opportunities all around the campus. One that the CS department sponsors in particular is their [inaudible] summer institute. So matching professors with undergrads to do work in the summer for slave wages. But great opportunity. You can get started on thinking about how research might be part of your future. One of the best preparations for thinking about grad school.

We do have an honors program. Like most, we have a chance to stake out on an independent project and show your mettle. We have a co-terminal masters program where you can get some of the depth before you leave this place. Not to mention, there's a million fabulous things to do with Stanford, getting involved with arts or nonprofit organizations or sports or the student leadership or whatever.

Maybe the most important memory I have from my undergrad was that the trickiest part of my undergrad was learning what to say no to. Realizing that there are 10,000 fabulous, worthwhile opportunities, and doing all of them only means sacrificing your sanity and the quality of the things you do. So be careful and thoughtful about choosing the things you do and embracing them. Be okay with saying, I won't do this.

Even though it's great to go overseas, and I would've liked that opportunity, it's not going to fit, and I'm okay with that. Don't beat yourself up over it and cause yourself too much grief trying to do too many things.

I won't be here on Friday. Keith is going to come. He's going to teach you about C++ in the real world. I'm going to see you guys Friday at the final. I think that's all I need to say for today.

[End of Audio]

Duration: 51 minutes

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