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