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