ProgrammingAbstractions-Lecture19

Instructor (Julie Zelenski):I've got the big box of graded exams, which I was hoping we'd get back to you today. And we do have them back, and we have the solution and stuff to go back with it, so I'll spend a little time at the end kind of talking about it. But it's – just so you know, it's something behind us now.

Assignment 5 is due today. So we'll do a little query on how much time got spend on Assignment 5. Hopefully this was a little bit lighter than the other ones, but we never know until we actually ask. How many people managed to do that in under ten hours? Okay, that's a big chunk. All right, I'd say over half of you. How many 10 to 15? A few people, kind of on that thing. More than 15? Way in the back, all right. When you – how – was it fun, a fun time? Were you playing with an algorithm or just getting it done?

Student:[Inaudible].

Instructor (Julie Zelenski):Okay. This next week assignment, Assignment 6, is now turning our attention from client-side programming to implementation-side programming. So we've done a lot of work so far that is dependent on using vector and grid and stack and queue and lexicon, and all sort of things. Now that we're starting to kind of open up that black box and look under the hood and see what's going on, we're now gonna change roles.

And so this is the – where things actually get a little bit hairier on the implementation side, is that suddenly the management of memory, the exposed pointers, the use of link lists, the use of raw array, suddenly is on your plate. You are implementing those things to provide the convenience that you've been counting on for the many weeks prior to this. So it does require kind of, I think, a real attention to detail and kind of perseverance to get through some of these things.

So the priority queue is the assignment you'll be working on, so today we'll talk about stack and queue as implementation topics. And then you'll be taking the idea of the queue and kind of twisting it a little bit into this form called the priority queue. And there are two implementations we give you that you just get to play with, and then there are two implementations that you'll get to write.

There's not a lot of code, actually, in either of them, but the code that's there is very dense. So kind of like back to that recursion things where it's like, well, any particular piece of it is not – it's not the length that gets you, it's the intensity and the kind of management of difficult data structures that's gonna be this week. So it's due a week-and-a-half from today. And I do think of this as being one of your heftier pieces, so this is not one you want to put off to the last minute try to get that done.

What I'm gonna do today is I'm gonna finish talking about vector. We'd gotten to where we had a skeleton vector that was – mostly appeared to be working at the end of Friday's lecture, but there's actually a couple things about it that we need to fix before we can

move away from that. And then we'll look at stack and queue as kind of our next thing: how do you implement a stacked queue, what options do you have there?

So the reading to go along with this, this is pretty much all Chapter 10: Class Templates, and then the next thing we'll look at is the editor buffer case study, which is covered in Chapter 9.

Student:I have a [inaudible] that relates to sorting. Are there metrics that determine like how well something is sorted or unsorted?

Instructor (Julie Zelenski):What do you mean by "how well" I guess is –

Student: Well I don't know. I mean something like – I was just gonna imagine, for instance, that if you could define some kind of scale that like at one end it would be completely sorted [inaudible] and then on the other end it would be completely unsorted, although I mean that's a hard to define –

Instructor (Julie Zelenski):Yeah, well and – I think you get the choice of how you define that. Some people, for example, would use things like, well what are the percentage of elements that are already in the correct position, is one measure of sorted. Right? And so that would be interesting in sorts that are trying to move things to the right place if it's already in exactly the right place.

Other times what you're more interested in is things that are out of order, so it might be that you know if you had the data sorted, like the front half was sorted and the back half was sorted. A lot of things are in the wrong place but it still is pretty sorted by – and so if you were considering some other variant of it, which is what are the number of sorted runs within it, so if you look at contiguous sequences. And sometimes you count things, like what's called the number of inversions, which is the number of pairs that are out of order relative to each other.

And so depending on kind of what things you're trying to look for, and that probably has a lot to do with what algorithm you're using, is what things can it take advantage of. Does it – is it better at taking advantage of things that are already in the right place? Or better at taking of things that are already in an ordered sequence with its neighbors, as to whether that would allow the algorithm perform more efficiently on that dataset.

But there is not one metric that everyone uses when they're saying how much sorted is, you know what percentage or what – how close to sorted or unsorted it is. It's kinda a matter of how you define your term as to what you mean by close.

Student: Right. But algorithms actually use something like that sometimes?

Instructor (Julie Zelenski):Sometimes, right? I mean some algorithms tend to actually depend on those features. So for example, there's one that looks for sort of sorted runs and then merges them. It's called a natural merge, so where you're looking for existing

sorted runs and then merging them from the bottom-up. And it's like that one, depending on the length and the number of runs in there, its runtime could be expressed using those metrics, which themselves are a measure of how close it is to being in the order that merge – this natural merge sort wants it to be.

So sometimes they do use that as a way of telling you about how does this perform, given this number of inversions or this number of placements and things like that. So it is one of the factors you're kind of looking at when you're trading off algorithms that depend on the ordering being given to start with, to just say things about their performance.

Anybody have an interesting sort that they actually had a fun time with, they learned something cool that they would love to have an audience to talk about? Way in the back?

Student: Heap sort is kind of cool.

Instructor (Julie Zelenski):Heap sort is very cool. Heap sort's a great algorithm, so anybody who – and how many other people actually ended up using heap sort as the one they did? You guys are in luck because it turns out the – part of the assignment review this week involves using a heap, so you guys are like already one step ahead of the game.

But heap is a great data structure and it actually has an awesome sort that actually has properties of N log N, so and actually it is a reliable N log N, doesn't have this kind of worst case degenerate thing hiding in there the way quick sort does, doesn't use any extra storage, auxiliary storage outside of it. It has a pretty neat algorithm that's kind of based on using the array itself as this notion of a heap, kind of a tree. So that's a great sort actually, it's kind of – I think one's that actually underappreciated for what it does.

Anybody else have a cool sort? Did anybody write a sort they thought was particularly fast when they were done? Sorts that beat our sorts that were really – you'd use in practice for fun? Did you guys all just write – what's the one that's alphabetically first on the list? This usually happens, whatever one I use first, it turns out half the class does. What was the one I listed first?

Student: Bingo.

Instructor (Julie Zelenski):Bingo. Did I put bingo first? Oh, that's a dumb sort. I think the last time I put comb sort first. And then it turned out everybody did comb sort because it was like the one that was in the front or something.

Student: So I found you could do the sorting algorithm called flash sort, which only looks for integers that uses an equation, basically to determine about where each integer should be in the final sorted array. And I wrote an implementation of it. I'm not sort how well it worked out, but –

Instructor (Julie Zelenski):That's cool.

Student:– it was actually really close to operating in [inaudible] space, even up to something like 10 million elements.

Instructor (Julie Zelenski):Oh, that's awesome.

Student: And it was more than twice as fast as the quick sort provided in the – [Crosstalk]

Instructor (Julie Zelenski):Excellent. Excellent. That's cool.

Okay. All right. So where did we leave off last time? I'm gonna reiterate these things that are quirky about the last changes I was making at the end, was taking an existing container class that held only strings and turning it into this template form of it that would now hold any kind of thing I wanted to stick into it. And along the way, I had to make a couple changes. And each of them actually kind of has a little – some pitfalls in getting this a little bit wrong, and so I just want reiterate them.

There's a lot of examples of these in the textbook. So you will see all of the templates that are implemented in the Chapters 9, 10, 11, and so on, do have examples of this. And for this first assignment, we won't be building it as a template, so this is actually kind of more like a future thing to kind of get prepared for when you're writing a template later.

But just a reminder about the things that have to happen is this template type name elem type, the template header we'll call that, gets placed sort of all over the place when you make that change. The first place it appears is in the .h on the class interface. You'll say template type name elem type class vector, class stack, class whatever it is you're building. That now means that the class is a template type whose name will only really exist in the form vector angle bracket string close angle bracket, from that point on.

In the .cpp, it gets repeated on every single member function definition. There's not one overarching kind of scope around all of those. Each of those has its own little independent setup for the scope, and then it has this additional sort of goo that needs to go on it about, oh, it's the template form of the member function size that is based on the vector class template. So you'll have to kind of get this all over the place.

And then the scope of the class, its new name is now whatever your original class name was but with the angle bracket elem type colon colon; there is no longer a vector unadorned once you've made that change. And so, when the client uses it, you'll always see that. And even on the scope resolution that's being used, even in the member functions themselves have this same adornment.

And then elem type gets used everywhere you have otherwise said it's storing strings. This is a string argument, this is a string return type, this is a local variable that's string, this is a data member that is a, you know, array of strings, whatever. All those places where you were saying – you were earlier committing to a precise type, you're gonna now use elem type everywhere.

It's pretty easy to do in like nine out of ten places you need to do it and leave one of them out. And the funny effect of that will be that sometimes it'll – you'll almost – it'll go unnoticed for a while because you – let's say you wrote it as a vector of strings originally. You change it to be a vector template, you left one of the places where there should've elem type still says string. But then you keep testing on vectors of string because you happen to have a lot of vector string code lying around that you were earlier using.

And what happens is you are instantiating the vector with string and it's kind of – the mistake is actually being hidden because the one place where it's wrong, it happens to be exactly string, which is what you're testing against. And it was only when you went out of way to then put vectors of int that you would suddenly get this type mismatch in the middle of the code, where it says, oh, here's this place where you were declaring a variable of string and you're trying to put an integer into it or something, or you're trying to have an array that's declared as integers and it really needs to hold – it's declared to hold strings and it really needs to hold ints.

And so you will – one way to shake that out early is actually to be sure that whatever testing code you're doing on one type, you also do it again on a different type, to make sure that you haven't accidentally got some subtle thing hiding in there.

And then this last thing is about how the templates are compiled, which is truthfully just a quirky behavior of the way C++ requires things to be done based on how the compiler's implemented, that the template is not compiled normally. So all normal files, you've got random.cpp and name.cpp and boggle.cpp, you always put those in the project, and that tells the IDE you want this one compiled. And it says, "Please compile this and link this in with all the code."

Once you have code that is templatized in here, you don't want it to be compiled in advance. It can't be compiled in advance, is really the problem. That looking at that code only describes the pattern from which to build vectors, not how to build one vector and compile it for all intents and purposes. It's on the fly gonna make new vector, vector, vector. So we pull it out of the project. We remove it, so we don't want the compiler trying to do something with it in advance.

For some compilers, actually, you can leave it there and it'll be ignored. The better rule to get used to though is just take it out because some compilers can't cope with it. You don't want to actually get into any bad habits. Pull it out; it no longer is compiled like an ordinary implementation file. You then change your header file to bring in that code, into the visible interface space here, at the very end by doing this #include of a .cpp file.

In no other situation would you ever want to #include a .cpp file. So as a habit, I'm almost giving you a little bit of a bad one here. Do not assume this means that in any other situation you do this. It is only in this one situation. I have a template class, this is the header for it, it needs to see the implementation body as part of the pattern to be able, to generate on the fly the client's particular types, and this is the way we're getting the code in there.

The way I would say most library implementations, sort of standard implementations in C++ do this, is they just don't bother with having a separation in the .h and the .cpp at all. They just put everything into the .h to begin with. And they don't even pretend that there is an interface separated from the implementation.

That's sort of sad because and sometimes that interface implementation split is an important psychologically of how you're thinking about the code and what you're doing. And I think jamming them together in the same file clouds that, that separation that we consider so important. But that is probably the most practical way to handle this because then it just doesn't – you don't have problems with thinking you can compile this and having to muck with this #include of cpp which is weird. So you'll see it done the other way in other situations, too.

All right, so that was just like I wanted to reiterate those because those are all little quirky things that are worth having somebody tell you, than having to find them out the hard way by trial and error. Any questions about kind of template goo? This is like the biggest class I've had in years. What do you guys – like is this midterm day? Was that why you guys are here or just because you heard I was gonna dress in a skirt and you wanted to see it? There's like three times as many people as we always have.

I'm gonna code. I'm gonna code, I'm gonna go to my new computer. You see my new fancy computer? It's like – I like it when we just code. I feel – like when I do it on the slides, I always feel like it's so dry, it's so boring, and it looks like it just you know came out of nowhere. It's good to kind of see it in real time and watch the mistakes be made live.

So what we're gonna do over here is we've got the myvector that we were working on last time. So it's got a skeletal interface. It has size add get set at, and some of the other functions are missing. The ones like clear and insert at and remove at, and the overloaded bracket operator and things like that are not there. Okay. But as it is, it does work.

In our simple test right here, we put nine, ten, one, and then we printed them back out. So let's just go ahead and run that to see that if I put a nine, ten, and a one and I iterate over what I got there using the get at and size, it seems like it has put ten – these three things in and can get them back out. Okay.

Now, I'm gonna do – I'm gonna show you something that's gonna make us a little bit sad and then we're gonna have to figure out how to fix it. That there is behavior in C++ for any class that you declare, that unless you state otherwise, that it's capable of assigning from one to another, making a copy. So all I added here at the bottom was another vector I called W, who – I should make this size a little bigger. This is screen is, I think, a little different than my other resolution, so let's crank that number up; make it a little easier to see what we're doing.

I declared a new myvector also of int type. It's name is W, and I said W=V. So this is actually true of all types that you declare in C++ that by default they have a default

version of what we call the assignment operator, that does kind of what's – you can imagine just a memberwise copy of V onto W.

So that works for structs, think about it in the simple case of a struct. If you have a struct with X and Y fields and you set this one to zero zero. And then you have point one has – set the X and Y to zero zero, and you say Point 2 = Point 1, it copies over the X and Y values. So you end up with two points who have the same field values, whatever they were, the same numbers. The same thing is true of classes, that unless you state otherwise, if you copy one instance, one object on top of another, it copies the values of the fields.

Okay, let's look at the fields to see how this is gonna work for us. The fields in myvector right now are a pointer to a dynamic array out here, two integers that are recording how many slots are used in that array and what its capacity is, and then there's no other data members, so we have three data members. So when I have made my V and my W, each of them has D space for one pointer here at the top and then these two integers.

And in the first one, if you remember a little about how the code works, it allocated it to some default size. I don't remember what it was, maybe it's, you know, ten or something. And then it fills it up from left to right.

So the first one I had, I'm gonna put in the numbers, you know, one, two, three, let's say. So I add one, I add a two, I had a three, then the number used will be three and let's say the number allocated is ten. So this is num allocated, this is num used. Okay. So subsequent things being added to that array will get added at the end. We know what our bounds are and all this sort of stuff. Okay.

Now I say W in my code, I say, "Oh yeah, just declare W." Well, W's constructor builds it a ten member array, sets the num allocated to ten, sets the num used to zero, and it's got kind of empty ready to go vector to put things into. When I say W=V, it's just gonna take these fields kinda one by one and overwrite the ones that were in W. Let's watch how that works out.

So we copy the num allocated over the num allocated, ten got written on ten, okay. We write the num used on top of the num used, okay. And then we copy the pointer on top of the pointer. This is pointer assignment, this is not doing any what we think of as a deep copy, it's a shallow copy or an alias. That means that W no longer points to here, its arr points to there.

All right. This can't be good, right? If you look at this picture, you've already got to be worried about where this is heading. You know just immediately you will notice that this thing: orphaned. Right? No one's holding onto this, this thing's hanging out in the heap, this guy's dead in the water. Okay, so we've lost some memory.

That, in itself, doesn't sound terrible, but now we have V and W both pointing to the same array. They have the same values for these two things but they're actually

independent. And now it is likely that if we were to continue on and start using V and W, we're gonna see some really strange effects of the two of them interfering with each other.

So for example, if I do a V.SetAt – well, let's do W for that matter, so I say W.SetAt at index zero, the number 100. So W says, you know, check and make sure that's in balance. It says, "Oh yeah, the index zero is within – it's greater than or equal to zero, it's less than my size. Okay." And it says, "Go in and write 100 in there."

As a result, it turns out V.GetAt also changed. And that would be pretty surprising that these happen. So now, they're just colliding. They have – there's one piece of memory that's being shared between the two of them, and when I change one of them, I'm changing both of them. They are not independent. They now have a relationship based on this assignment that's gonna kinda track together.

There are actually worse consequences of that than that. So this looks kind of innocent so far, not – "innocent" isn't the word but at least it's the opportunity for malicious error still seems like, well okay, you have these values that are changing. The really terrible situation would happen when let's say V keeps growing, they keep adding more things.

So they add the numbers four, five, six, seven, eight, nine, and then it fills up. So it has num used as ten, num allocated as ten, and it goes through and it says, "Oh, it's time to grow. I want to add an 11." And the process of growing, if you remember, was to build an array that was twice as long, copy over the front values, and so copy up to here, have this back half-uninitialized and then deallocate the other one. So delete this piece of memory, and then update your pointer to point to this new one, that now is allocated to a length of 20.

Okay, when you did that, W just got screwed. W points to this piece of memory that you just took away. And then you will be much more sad than just getting a junk value out or a surprisingly changed value out. Now if you ask W to go get the value at position zero, all bets are off. Right? It might happen to work, it probably will work for a little while, but at some point this memory will get reclaimed and reused and be in process for something else, and you'll just have completely very random looking undetermined results from access to that W.

So this really just can't exist. We do not want it to be the case that this default memberwise assignment goes through. It will do us no good. So in the case for objects where memberwise copying is not what you want, you have to go out of your way to do something about it.

The two main strategies for this is to really implement a version of assignment operator that will do a keep copy. That's actually what our ADTs do. So the vector and the map and stack and queue are setup to where if you say V=W, it makes a full copy. And that full copy means go take the piece of memory, get another new piece of memory that big, copy over all the contents.

And so then, you end up with two things that have the same number of elements in the same order, but in new places in memory. They're actually fully duplicated from here to there. And at the point, V and W don't have any relationship that is gonna be surprising or quirky in the future. They just happen to get cloned and then move from there.

That's the way kind of a professional sort of style library would do it. What I'm gonna show you instead, because I don't really want you to learn all the goopy things about how to make that work, is I'm gonna show you how to just disallow that copying. To make it to where that it's an error for you to attempt to copy V onto W or W onto V, by basically taking the assignment operator and making it inaccessible, making it private.

So I'm gonna show you how I'm gonna do that. Well actually, first I'll just show that like the truth about these kind of errors. And these errors can be really frustrating and hard to track down, so you just don't want to actually mess with this. If I have W=V here, and then I say W.SetAt zero is 100. So I was supposed to put in the numbers – let me go – numbers I know where to look for.

I'm gonna put in one, two, and three. I change it in W and then I write another for loop here that should print out the values again. This is where I'm gonna see. So like 1, 2, 3 now have 100, 2, 3. And then actually I'm even seeing an error here at the end where it's saying that there was a double free at the end. The free is kind of the internal name for the deallocation function.

And in this case, what happened is that the destructor for V and W were both being called. As I exited scope, they both tried to delete their pointer. So one of them deleted it and then the second one went to delete it, and the libraries were complaining and said, "You've already deleted this piece of memory. You can't delete it twice. It doesn't make sense. You must have some error." So in fact, we're even getting a little bit of helpful runtime information from what happened that might help us point out to what we need to do.

So let me go and make it stop compiling. So there is a header file in our set of libraries that's called disallow copy. And the only thing that is in disallow copy is this one thing that's called a macro. It's kind of unfortunate that it has to be done that way. Well, I can't find the header file, so I'll just tell you what it is. And it looks like this. And I say disallow [inaudible] copying, in all capital letters, and then I put in parentheses the name of the class that I'm attempting to disallow copying on.

You can have a semicolon at the end of this or not, it doesn't matter actually. I'll put one just because maybe it looks like more normal to see it that way. And by doing this in conjunction with that header file it's bringing in the macro that says put into the private – so we always put this in the private section. So we go to the private section of our class, we put this disallow copying macro in here. And what this will expand to is the right mojo.

There's sort of a little bit a magic incantation that needs to be generated, and this is gonna do it for you. That will make it such that the – any attempt to clone a vector using that memberwise copy, and so that would happen both in direct assignment from V to W, or in the cases where copies are made, like in passing by value or returning by value, those actually are copy situations as well, that it will make all of those things illegal. It will take the standard default behaviors for that, and make them private and inaccessible and throw errors basically, so that you just can't use them.

And if I go back to – have made this change, I go back to the code that was trying to do this, and I'll take out the rest of the code just so we don't have to look at it, that it's gonna give me this error. And it's gonna say that in this context, the errors over here, that the const myvector, and there's just a bunch of goo there. But it's saying that the operator equals, is the key part of that, is private in this context.

And so, it's telling you that there is no assignment operator that allows one myvector to be assigned to another myvector. And so any attempt by the client to make one of those copies, either from passing, returning, assigning, will just balk right then and there and not let the kind of bad thing that then will just have all sorts of other following errors that we would not want to have to debug by hand.

So this will become your mantra for any class that has memberwise variables to where copying them in a naive way would produce unintended effects. So if all the variables in here were all integers or strings or, for that matter, vectors or other things that you know have true deep copying behavior, then you don't need to go out of your way to disallow copying.

As long as each of the members that's here, that if you were to do an assignment from one variable of this type to the other, it would have the right effect, then you don't need to disallow. But as soon as you have one or more variables where making a simple assignment of it to another variable of that type would create some sort of long-term problem between those two objects, you want to disallow that copying and not let that bad thing happen. So things that have a link list in them, things that have pointers in them, dynamic arrays in them, will all need this protection to avoid getting into that situation.

Question?

Student: Could you somehow overload the assignment [inaudible]?

Instructor (Julie Zelenski):Well, you certainly can so that's kind of the first strategy I said, which is like, well, you can make them deep copy is the alternative. So one way is to say, "Well the shallow copy is wrong. What are you gonna do about it?" So I'm saying don't let shallow copy happen. The other alternative is to replace shallow copy with a real deep copy. That's what ours do.

If you're interested to look at that, you can look at our code and see what we do. It just goes through – you can imagine what it would take. It's like, okay, get some information from here, make a new array, copy the things over, and then at that point you will be able to continue on with two things that are cloned from each other, but no longer have any relationship that would cause problems.

It's not that it's that hard but the syntax for it is a little bit goopy, and ours in C++ we're not gonna see. So you can look at it; if you want to as Keith at 106L, he will tell you everything you want to know.

Any questions about – over here?

Student: So the things inside of this copy, can be anything? It can be one of the variables you declared?

Instructor (Julie Zelenski):So typically it'll be the name of a class.

Student: But could be like it'd just have a variable inside your class, you don't want people really to copy [inaudible]?

Instructor (Julie Zelenski):Well it doesn't really work that way. The disallow copying is saying I'm taking this type and making it not able to be copied, and so it is an operation that applies to a type, not to a variable. And so, in this case, the name here is the name of the class who we are restricting assignment between things of myvector type. And so, it really is a type name not a variable name.

If I said disallow copying of like arr or num used, it actually won't make sense. It'll expand to something that the compiler won't even accept. It says, "I don't know what you're talking about." It expects a type name in that capacity, so what thing cannot be copied. It is things that are of myvector type.

So there's not a way to say you can't copy this field by itself or something like that. It's really it's all or nothing. You can copy one of these objects or you can not copy it. And once you have some field that won't copy correctly without help, you're gonna wanna probably disallow the copying to avoid getting into trouble.

All right, so you got vector; vector's a good thing. I'm gonna add one more member function of vector before I go away, which is insert at. I'll call it E. The insert at one is the – allows you to place something in an index in the – which one I just copy? There I go. Insert at index in an element type. So if the index is before the beginning or off the end, I'll raise the same out of bounds error, so that's I just picked up that piece of code from there.

I also need to have this little line, which is if the number of elements used is equal to the capacity, then we need to make more space. So it – just like the case where we're adding to the end, if we're already at capacity, no matter where we're inserting, we have to make

some more space. So we go ahead and do that, the initial step of checking to see if we're out of capacity and enlarging in place.

Once we've got – we're sure we have at least enough room, we're gonna move everybody over. So I'm gonna run a for loop that goes from size minus one to – whoops, I need an I on that. Is greater than the index – actually, I want to do greater than or equal to index. And I'm gonna move everything from index over in my array, and then array of index equals E.

Let me think about this and make sure I got the right bounds on this, right? This is taking the last element in the array, it's at position size minus one; and then it's moving it over by one, so it's reading from the left side and moving it over by one. And it should do that all the way down until the last iteration should be that I is exactly equal to index. And so, it should move the thing out of the index slot to the index plus one slot, so along the way moving everybody else there.

Why did I run that loop backwards?

Student:[Inaudible].

Instructor (Julie Zelenski):Yeah, it overran the other way. Right? If I try to do it the other way, I try to copy, you know I have the numbers four, five, and six in here and I'm planning on inserting at zero, I can't start from the beginning and four on top of the five and then on top of that, without making kind of a mess of things. So it's actually – I can make that work but it's definitely sort of messier. It's sort of easier to think about it and say, "Oh, well just move the six over, then move the five over, then move the four over, and then write your new element in the front." And before I'm done, I do that.

And so, I have insert at; and I could probably test it to make sure that it does what it's supposed to do: V.InsertAt position zero, put a four in the front, and then move this loop down here. So I have one, two, three, and then I put a four in the front. Oh, look at that, something didn't work. Want to take a look at that? Four, one, two, what happened to my three? Where did my three go? Why did I lose my three?

How does it know how many elements are in the vector? It's keeping track of that with num used. If you look down here, for example, at add, when it sticks it in the last slot, it updates the num used and increments it by one. I wasn't doing that over here at insert, right? And a result, like it didn't – you know it's admirable job. It moved all the numbers over, but then it never incremented its internal counter, so it thinks actually there's just exactly still three elements in there. So I'd better put in my num used ++ in there too. Ah, four, one, two, three. So my number was there, I just wasn't looking at it. It was just a little further in there. Okay.

All right, so with that piece of code in, I think we're in a position to kinda judge the entire strategy of using a dynamic array to back the vector. The remove at operation is similar, in terms of needing to do that shuffle; it just does it in the other direction. And

then we have seen the operations, like set at and add and get at, and how they're able to directly index into the vector and kind of overwrite some contents, return some contents. And so you kinda get a feel for what it is that an array is good at, what things it does well, and what things it does poorly.

So a vector is an abstraction. You offer up to somebody a vector it's a list of things. That the tradeoffs that this implementation of vector is making, is it's assuming that what you most care about is that direct access to anything in the vector because that's what arrays are good at. It is a trivial operation to say start at the beginning and access the Nth member, because it just does a little bit of math. It takes the starting address in memory and it says, "Oh, where's the tenth member? It's ten positions over in memory." And so, it can do that calculation in no time, and then just directly access that piece of memory.

So the – if what you are really concerned about is this ability to retrieve things or update things in that vector, no matter where you're talking about, the front, the back, the middle, in constant or one time, this design is very good for that. What are the operations it's gonna actually bog down on? What is it bad at? When do you see sort of the consequences, let's say, of it being in contiguous memory, being a disadvantage rather than advantage?

Student: Adding something at the [inaudible].

Instructor (Julie Zelenski):Adding something where?

Student: At the beginning.

Instructor (Julie Zelenski):At the beginning. Certainly any operation like this one, the insert at or the remove at, that's operating in the front of the array is doing a big shuffle, things forward, things back, to make that space, to close down that gap. And so, for an array of large enough size, you'd start to notice that.

You have an array of 1,000, 10,000, 100,000, you start inserting at the beginning, you're gonna see this cost of the insert shuffling everything down, taking time, relative, in this case, linear, in the number of elements that are being moved. Which on average, you figure you're inserting kind of randomly in positions, it could be half of the elements or more are gonna need to move.

It also actually pays a little cost in the resizing operation. That happens more infrequently. The idea that as you get to capacity, you're growing, in this case we're doubling, so we're only seeing that kind of at infrequent intervals, especially as that size gets large. Once you get to 1,000, it'll grow once to be 2,000. Well then, it'll grow once and be 4,000. So you'll have a lot of inserts before you'll see that subsequent resize operation.

But every now and then, there'll be a little bit a glitch in the resizing where you would say, "I'm adding, adding –." If you're adding at the end, adding at the end is easy, it

increments the num used and sticks it at the end. But once every blue moon, it'll be a long time as it copies. And then you'll be go back to being fast, fast, fast, fast, fast, till you exhaust that capacity. And then again, every now and then, a little bit a hiccup when it takes the time to do the resizing.

Overall, the number of inserts, that cost can kind of be averaged out to be considered small enough that you'd say, "Well, amortized over all 1,000 inserts it took before I had to copy the 1,000 when I grew to 2,000, each of them paid 1/1000th of that cost, which ended up making it come out to, on average, still a constant cost," is one way that we often look at those analyses.

So this tells you that like there's certain things that the array is very good at. It has very little additional memory overhead. Other than the additional space we're kind of storing in the capacity when we enlarge, there really isn't any per element storage that's used in addition to the number or the string or whatever it is we're storing itself. So it has very low housekeeping associated with it. And it does some things very well but other things a little less efficiently because of having to stick it in memory meant we had to kind of do this shuffling and rearranging.

So the other main alternative you could use for a vector is a link list. I'm actually not gonna go through the implementation of it because I'm actually gonna do stack as a link list instead. But it is interesting to think about, if you just sort of imagine. If I instead backed vector with a link list, first of all could I implement all the same operations? Like is it possible to use a link list, can it do the job if I'm determined?

Like what will it take, for example, to get the element at the nth position from a link list design? How do you know where the nth element of a linked list is? It's [inaudible] list, you know. Help me out. [Inaudible].

Student: You have to actually dereference the pointers N times.

Instructor (Julie Zelenski):That's exactly what you got to do; you got to walk, right? You got to start the beginning and say, "Okay you're zero. And after you is one, and after that's two." And so you're gonna walk, you're gonna do N arrow next, to work your way down. You will start at the beginning and – and we're gonna learn some new vocabulary while we're here. And so that means that accessing, for example, at the end of the list is gonna be an expensive proposition.

In fact, every operation that accesses anything past the beginning is doing a walk all the way down. And so if you planned on, for example, just printing the list or averaging the list or searching the list, each of those things is going, "Give me the zero, give me the next, give me the third." You know it's gonna just keep walking from the beginning each time. Like it's gonna turn what could've been a linear operation into N squared because of all those start at the beginning and walk your back down.

So for most common uses of a vector, that would be so devastating that you just wouldn't even really take it seriously. That every time you went to get something out of it or set something into it, you had to make this enormous traversal from the front to the back would just be debilitating, in terms of performance implications.

The thing that the link list is supposed to be good at is that manipulation of the memory. That the operation, like insert at or remove at, that's doing the shuffle, the link list doesn't have to do. So for example, the worst place you can insert at in a vector is zero, if it's implemented using an array because you have to shuffle everybody over. The best place you can insert at in a link list is actually at zero because then you just tack a new cell on the front.

And then sort of further down the list, right? It's not the act of splicing the new cell in that's expensive; it was finding the position at which you needed to do the splice. So if you want to insert at halfway down the list, you have to walk down the list, and then you can do a very quick splice but you had to get there.

So it makes this – it's kind of an inverted set of tradeoffs, relative to the array form, but adds a bunch memory overhead, adds a bunch of pointers, adds a bunch of allocation and deallocation. So there's a bunch of code complexity, and in the end you don't really get anywhere I think you'd want to be. So it's very unusual to imagine that someone would implement something like the vector using a link list strategy, although you could. So.

Let's talk about stack, instead. I have a little stack template that I started: mystack. So I push in pop and size on. Okay. I'm lazy, so I'm going to do the easiest thing in terms of implementing stack: that is to layer.

Okay. So once you've built a building block as an implementer, there's nothing that stops you from then turning around in your next job and being a client of that building block, when implementing the next thing you have to do. It may very well be that the piece that you just built is a great help to writing the next thing, and you can build what's called a layered abstraction. I'm ready to build stack and I've already gone to all this trouble to build vector, it's like, well. It turns out one of the ways I could implement a stack is to use something like a vector or an array.

Now I could build it on a raw array, but I happen to build something that kind of has the behaviors of a raw array: the tradeoffs of a raw array, the Big-O of raw array, and it actually just manages convenience, it has error checking and stuff in it. It's like why not just use that? I'll pay a little bit of cost in terms of taking its level of indirection onto things, but it's actually gonna be worth it for saving me sort of the grungier aspects of the code.

So if I'm gonna put stack contents into an array, and if I were going to push A then B then C, right? So then, the stack that I want to have would look like this. There's the top of the stack up here, where C is the last thing on. The bottom of the stack is down here; with the first thing I pushed on, which is A. I could put this in an array. Seems like the

two obvious ways to do this would be, well, to put them in this way: A, B, C, or to put them in this way. Okay. So this would be the top of the stack is over here, the bottom of the stack is over there, and then down here it's inverted. This is the top of the stack; this is the bottom of the stack.

Okay, they seem symmetric, you know at the very least, but one of those is a lot better for performance reasons. Which one is it? Do you want to go with Strategy 1 or Strategy 2?

Student: One.

Instructor (Julie Zelenski):One? Why do I want one?

Student: What?

Instructor (Julie Zelenski):Why?

Student: Because you don't have to move all the –

Instructor (Julie Zelenski):Exactly. So if I'm ready to put my next element in, I'm ready to put D in, that in the Strategy 1 here, D gets added to the end of the vector. Right? Adding to the end of vector: easy, doesn't require any shuffling, updates the number. Sometimes it has to grow but easiest thing to do, add to the end. In this version, adding to the top would be moving everybody over, so that I have to update this to D, C, B, A, and slide everybody down. So I had to do insert and a shuffle: bad news, right?

Similarly, when I'm ready to pop something, I want to get the topmost element and I want to remove it that the topmost element here being at the end, remove at is also fast at the very end. Right? Taking the last element off, there's no shuffling required; I just [inaudible] my num used. If I need to do it from here, I have to shuffle everybody down. So it seems that they were not totally symmetric, right? There was a reason, right? And I had to think it out and kind of do it and say, "Oh yeah. Okay, put them at the end."

So if I do that and I go over here, I'm gonna finish doing the things that make it a proper template. I just want to look at mystack, ooh, and I have the things here, is that I don't have anything to do with my constructor or destructor. I'm just gonna let the automatic construction and destruction of my members work, which is to say give me a vector, delete my vector. All that happens without me doing anything.

Then I want to know my size. I don't know my size but the vector does for me, so I tell the vector to do it for me. When I want to push something, I tell my vector to add it for me. I'm telling you, this is like a piece of cake. And so then I say elem type top equals elems.GetAt. I can actually use the – I'm back to using the real vector, so I can use anything that has at the last position. Elems.RemoveAt, elems.size [inaudible], then I return it. Almost but not quite what I want to do, but I'll leave that there for now.

And then that's defining the function I wanted, right? So they're all just leveraging what the vector already does for me, in terms of doing the remove. I guess RemoveAt actually, the name of that member function. The add that does the growing and the shifting and all this other stuff happened. Oh good, I have a stack and I didn't have to do any work. I love that. So let's see if it works.

Mystack.s, push. Pop one thing off of it just for – even though I actually got what I was supposed to get. Oh, I'd better include the right header file; I'm gonna do that. Doesn't like size; let's see what I did with size that I got wrong. Okay, should be size with arguments, there we go. What did I put? One, two, three, and then I popped and I got a three. Hey, not bad, not bad for five minutes of work.

There's something about this pop method though that I do want to get back. So push actually is totally fine, that's just delegating the authority to kind of stash the thing in the vector. Pop right now, what's gonna happen if there isn't anything in the stack when you pop? Something good? Something bad? Something helpful? Wanna find out? Why don't we find out? Never hurts to just try it.

So like right now, it seems like it just you know digs into the elem. So if the elem size is zero, there are no elements in the stack, it'll say, "Oh, give me the elements of negative one," and then it'll try to remove that negative one. I got to figure those things aren't good things. I'm hoping that we'll push one thing and then we'll pop two things. See how it goes.

Hey, look at that. [Inaudible] access index negative one and the vector size zero, so that was good. It was good that we got an error from it. We didn't actually like just bludgeon our way through some piece of memory that we didn't know it or anything crazy like that. But the error that we got probably wasn't the most helpful. That upon seeing that error, it might cause you to go – kind of get a little bit misled. You might start looking around for where am I using a vector? Where am I making a call to vectors bracket?

Like I look at my client code and all I see is use of stack. The fact that stack is implemented on vector, is something I'm not supposed to know or not even supposed to be worrying about. And so having it poke its head up there, might be a little less clear than if, instead, it had its own error message.

And so I can actually just go in here to mystack and say, "Yeah, you know, if write my size is zero, error, pop empty stack." That you know it doesn't really change the behavior in any meaningful way. It's like it still gets an error, but it gets an error in this case that's more likely to tell somebody where the trouble is and where to look for it: so start looking for pop on a stack, instead of expecting to go look for a vector access. So just being a little bit more bulletproof, a little bit more friendly to the client by doing that, that's what.

And so all the operations on this stack are O of one right now, the add, the pop are – push and pop are both adding to the end of the array, which is easy to do. And so the – this is a pretty efficient implementation and pretty easy to do because we're leveraging vector.

What we're gonna look at next time is what can a link list do for us or not, does it provide anything that might be interesting to look at. What I'm gonna talk about instead is I'm gonna give you the two-minute summary of what the midterms looked like. And then we will give them back to you, which is, I'm sure, what you just wanted on your nice Monday morning.

So the histogram, can I have one of the solutions here? So you've got your really just rock solid normal distribution but with a kind of very heavy weighting sort of toward the end, right? The median was I think in the mid-70s: 74, 75; the mean was a little bit lower than that, and so a very strong performance overall. I'd say that probably the most difficult on the exam was clearly the recursion question, whereas I'd say the average on that was about eight or nine points down from perfect, so.

But there was actually – it's kind of a very interesting clumping. A bunch of people who were totally perfect, a big bunch that kinda got something right in the middle, and then some smaller groups on the other end. So they're very kind of separated into three groups.

The solution has some information, has the grading stuff, stuff like that, kind of gives the thing. The most important thing though is if you are at all concerned about how to interpret your grade, so if you feel like you want some more information, the check, check-plus stuff on the assignments kind of confuses you, you just want to have an idea, the right person to talk to is me. So you can certainly come to my office or send me an e-mail, if you just kind of where you're at and how to make sure you're at the place you want to be going forward, I'm happy to talk that through with you.

So in general, I thought the exam was very, very good. It was a little bit long, I think. Even though people did manage to answer all the questions, I think had it been a little shorter we might've have gotten a little bit higher scores overall. But in general, I thought the scores kind of right in the target range we wanted to have, so I'm very pleased.

If you would like to come up, I'm gonna have to figure out how to do this in a way that does not actually make a huge – I think what I will do is I'm gonna do the alphabet. I have them alphabetized by last name, and I'm gonna start by putting kind of As, you know A through C in a little pile, and so on, and sort of across the room. So depending on where your last name is, you might want to aim yourself toward the right part of the room. Okay?

So I've got a little stack of like As and Bs, let's say; this is kinda Cs and Ds and something else.

[End of Audio]

Duration: 51 minutes