ProgrammingAbstractions-Lecture18

Instructor (Julie Zelenski):Okay. Well welcome, welcome. We got a couple parents who have braved the other activities of Parents Weekend to come here, so thank you for coming to see what it is that we do here in CS106B.

We are still in the process of grading the midterm. In fact, most of our grading, we're planning on taking care of tomorrow, no, Sunday, in some big, massive grading sessions. So with any luck, if it all goes well, we'll actually have them ready to go back on Monday, but if somewhere it turns into some disaster, it may take us a little longer, but hopefully. That's the plan. So you get to have your weekend without worrying about it. Just put it aside and have some doing some stuff.

What I'm gonna talk about today is I'm gonna briefly go through scanner, really fast. Actually, what I really want to talk about is vectors, so maybe I won't even bother with the scanner. We'll just talk about vector, which is, the reading going on with this, Chapter 10: Implementing the Class Template.

So what we're gonna do here for the next two weeks is we're gonna take and put all those templates we did a lot of stuff with. Right? We saw vector, we saw stack, we saw a queue, we saw map, we saw set. And now it's time to say, "Well, how do they work? How do they manage to do the things they do, efficiently, safely, robustly; what is it like to implement those things?"

And so it's – we've had a good run of being the client, which hopefully has impressed you with like why you want these things around. And now, with some knowledge about linked lists and pointers and Big O, how can we make some make those operations run efficiently and make them do cool things? What does it take to be the backend side?

So that's kind of gonna be our role for pretty much the next two weeks straight. And then at the end, we'll kinda come back and try to join both sides together and get a vision of where we're at. But it is time to be implementer.

We'll be doing the Terman Café today after class, so hopefully those of you whose parents aren't here, or your parents can come with us too, we'll – anybody who has time is definitely welcome to come over and have a nice latte with me and hang out. And tell me about their midterm success or their midterm troubles, or just things that are on your mind. So hopefully some of you will be able to join me.

So I'm gonna finish up a couple slides from last time, and then I'm gonna mostly go on to do some code with you together. But I just wanted to try to give you some perspective on the value of abstraction and the idea of an abstract data type, or an ADT, and why this is such a powerful and important concept in management of complexity.

So we saw this, for example, in those first couple assignments that the client uses classes as an abstraction. You have the need to manage something that has a queue-like behavior:

first in, first out. So what you want is something that you can put things in that will enforce that: put things at the end of the line, always return to the head of the line. And that how it managed what it did and what went on behind the scenes wasn't something we had to worry about or even concern ourselves with.

And in fact, even if we wanted to, we couldn't muck around in there. Like it wasn't our business, it was maintained privately. And that is a real advantage for both sides, right? That the client doesn't have to worry about it and can't even get in the way of us, that we can work independently and get our things done.

And so one sort of piece of terminology we often use here, we talk about this wall of abstraction. That there is kind of a real block that prevents the two of us from interfering with each other's process, as part of, you know, combining to build a program together. And there's a little tiny chink in that wall that we call the interface. And that's the way that you speak to the stack and ask it to do things on your behalf, and it listens to your requests and performs them.

And so if you think about the lexicon class, which we used in the Boggle and in the recursion assignment that managed a word list, the abstraction is, yeah, you say, "Is this word in there? Add this word to it. Load the words from a file." How does it store them? How does it do stuff? Did anybody go open up the lexicon.cpp file just to see? Anybody who was curious? And what did you find out?

Student:I just – I think it ended up in there [inaudible].

Instructor (Julie Zelenski):You ended up in there and when you got in there, what did you decide to do?

Student: Leave.

Instructor (Julie Zelenski):Leave. Yeah. Did anybody else open it up and have the same sort of reaction? Over here, what did you think?

Student:I didn't really understand.

Instructor (Julie Zelenski):Yeah. And what did you see?

Student: It was a mess.

Instructor (Julie Zelenski):It was a mess. Who wrote that code? My, gosh, she should be fired.

It's scary. It does something kind of scary. We'll talk about it. Actually, at the end, we'll come back here because I think it's actually a really neat class to study. But in fact, like you open it up and you're like, "I don't want to be here. I want to use a word list. Let me close this file and let me go back to word list. Add word, contains word, okay." And you're happy about that. Right?

It does something very complex that turns out to be very efficient and optimize for the task at hand. But back to Boggle, you don't want to be worrying about that; you got other things on your plate and that's a fine place to just poke your nose right back out of. So if you haven't had a chance to look at it, because we will talk about it later, but the person who wrote it, was a crummy commoner. I'm just saying that they would definitely not be getting a check-plus on style, so.

So I drew this picture. It's like this wall. Right? So when you made a lexicon, you say, "Oh, I want a lexicon. Add this word. Add that word. Does it contain this word?" And that there is this big wall that – and you think of what's on the other side as a black box. The black box is the microwave that you push buttons and food gets hot. How does it really work? Ah, you know, who knows. You don't open up the back. You don't dig around in it. And this little chink here, that's the interface. And the lexicon provides a very small interface, if you remember, adding the words, checking contains word and prefix, reading the words from a file, really not much else beyond that.

And then now what we're starting to think about is, "Well, what's this other side look like?" What goes over here is that there is this implementer who does know the internal structure. Who does know what shenanigans are being pulled on that inside, who does have access to that private data, and who, upon request when you ask it add a word or look up a word or look up a prefix, internally trolls through that private information and returns information back to the client, having mucked around in those internals.

So when you say add a word, maybe what its got is some array with a count of how many words are used, and it sticks it in the next slot of the array and updates its counter. And we don't know what it does but it's not really our business as the client, but the implementer has kind of the full picture of that.

And then over here on this side, if the client attempts to do the kind of things that are really implementer specific, it tries to access directly the num words and the words array to go in and say, "Yeah, I'd like to put pig in the array. How about I do this? How about I change the number of words? Or how about I stick it in the slot at the beginning and overwrite whatever's there." That an attempt to do this should fail. It should not compile because these fields, that are part of the lexicon, should have been declared private to begin with, to make sure that that wall's maintained. Everything that's private is like over here on this side of the wall, inaccessible outside of that.

And so why do we think this is important? Of all the ideas that come away from 106B, I think this is one of the top three is this idea of abstraction. We actually even call the whole class Programming Abstractions. Because that the advantage of using this as a strategy for solving larger and more complicated problems is that you can divide up your tasks. That you can say this is an abstraction, like a word list, and kind of have it be as fancy as it needs to be behind the scenes but very easy to use from the outside.

So that makes it easy for me to write some piece and give it to you in a form that's easy for you to incorporate. We can work on our things without stepping on each other. As you get to larger and larger projects beyond this class, you'll need ways of making it so three people can work together without stepping on each other's toes the whole time. And classes provide a really good way of managing those boundaries to keep each other out of each other's hair.

And there's a lot of flexibility given to us by this. And we're gonna see this actually as we go forward, that we can talk about what a vector is. It keeps things in index order. Or a stack is, it does LIFO. But there is no guarantee there about how it internally is implemented, no guarantee expressed or implied, and that actually gives you a lot of flexibility as the implementer.

You can decide to do it one way today, and if upon later learning about some other new technique or some way to save memory or time, you can swap it out, replace an implementation with something better, and all the code that depends on it shouldn't require any changes. That suddenly add word just runs twice as fast or ten times as fast, would be something everybody could appreciate without having to do anything in their own code to take advantage of that. So these are good things to know.

So what I'm gonna do actually today is I'm gonna just stop doing slides, because I'm sick of doing slides. We do way too many slides; I'm bored with slides. And what I'm gonna do is I'm gonna actually go through the process of developing vector from just the ground up. So my plan here is to say our goal is to make the vector abstraction real, so to get dirty, get behind the scenes and say we know what vector is. It acts like an array. It has these indexed slots. It's this linear collection. It lets you put things anywhere you like in the vector.

We're gonna go through the process of making that whole thing. And I'm gonna start at the – with a simple form that actually is not templatized and then I'm gonna change it to one that is templatized. So we're gonna see kind of the whole process from start to end.

So all I have are empty files right now. I have myvector.h and myvector.cpp that really have sort of nothing other than sort of boilerplate stuff in them. So let me look at myvector.h to start with. So I'm calling this class myvector so that it doesn't confuse with the existing vector class, so that we have that name there. And I'm gonna start by just putting in the simple parts of the interface, and then we'll see how many other things we have time to kind of add into it. But I'm gonna get the kinda basic skeletal functions down and show how they get implemented, and then we'll see what else we'll try.

So having the size is probably pretty important, being able to say, "Well how many things will I put in there," get that back out, being able to add an element. And I'm gonna assume that right now the vector's gonna hold just scranks. Certainly that's not what we're gonna want in the long run, but I'm just gonna pick something for now so that we have something to practice with. And then I'm gonna have the get at, which give it an index and return something.

Okay, so I think these are the only ones I'm gonna have in the first version. And then the other ones, you remember, there's like a remove at, there's an insert at, there's an overloaded bracket operator, and things like that I'm not gonna show right away.

Question?

Student:[Inaudible].

Instructor (Julie Zelenski):Oh yeah, yeah. This is actually you know kind of just standard boilerplate for C or C++ header files. And you'll see this again and again and again. I should really have pointed this out at some point along the way is that the compiler does not like to see two definitions of the same thing, ever. Even if those definitions exactly match, it just gets its total knickers in a twist about having seen a class myvector, another class myvector.

And so if you include the header file, myvector, one place and you include it some other place, it's gonna end up thinking there's two myvectors classes out there and it's gonna have a problem. So this little bit of boilerplate is to tell the compiler, "If you haven't already seen this header, now's the time to go in there." So this ifindex, if not defined, and then the name there is something I made up, some symbol name that's unique to this file. When I say, "Define that symbol," so it's like saying, "I've seen it," and then down here at the very bottom, there's an end if that matches it.

And so, in the case where we have – this is the first time we've seen the file it'll say, "If not to define the symbol it's not." It'll say, "Define that symbol, see all this stuff, and then the end if." The second time it gets here it'll say, "If not define that symbol, say that symbol's already defined," so it'll skip down to the end if. And so, every subsequent attempt to look at myvector will be passed over. If you don't have it, you'll get a lot of errors about, "I've seen this thing before. And it looks like the one I saw before but I don't like it. You know smells suspicious to me." So that is sort of standard boilerplate for any header file is to have this multiple include protection on it.

Anything else you want to ask about the – way in the back?

Student: Why would it look at it multiple times, though?

Instructor (Julie Zelenski):Well because sometimes you include it and sometimes – for example, think about genlib. Like I might include genlib but then I include something else that includes genlib. So there's these ways that you could accidentally get in there more than once, just because some other things depend on it, and the next thing you know. So it's better to just have the header file never complain about that, never let that happen, than to make it somebody else's responsibility to make sure it never happened through the includes.

So I've got five member functions here that I'm gonna have to implement. And now I need to think about what the private data, this guy's gonna look like. So now, we are the low-level implementer. We're not building on anything else right now, because myvector is kind of the bottommost piece of things here. We've got nothing out there except for the raw array to do the job for us.

So typically, the most compatible mechanism to map something like vector onto is the array. You know it has contiguous memory, it allows you to do direct access to things by index, and so that's where we're gonna start. And we'll come back and talk about that, whether that's the only option we have here. But what I'm gonna put in here is a pointer to a string, or in this case it's gonna be a pointer to a string array.

And so in C++ those both look the same, this says arr is a single pointer that points to a string in memory, which in our situation is actually gonna be a whole sequence of strings in memory kind of one after another. The array has no knowledge of its length so we're gonna build is using new and things like that. It will not know how big it is. It will be our job to manually track that.

So I'm gonna go ahead and have two fields to go with it, and I'm gonna show you why I have two in a minute. But instead of having just a length field that says the array is this long, I'm actually gonna track two integers on it because likely the thing I'm gonna do with the array is I'm gonna allocate it to a certain size and then kind of fill it up. And then when it gets totally filled, then I'm gonna do a process of resizing it and enlarging it.

And so, at any given point, there may be a little bit excess capacity, a little slack in the system. So we might start by making it ten long and then fill it up, and then as we get to size 10, make it 20 or something like that. So at any point, we'll need to know things about that array: how many of the slots are actually used, so that'll be the first five slots are in use; and then the number of allocated will tell me how many other – how many total slots do I have, so how many slots can I use before I will have run out of space.

Both of those. Okay, so there's my interface. So these things all public because anybody can use them; these things private because they're gonna be part of my implementation; I don't want anybody mucking with those or directly accessing them at all, so I put them down there.

All right, now I go to myvector.cpp. So it includes myvector.h, so that it already has seen the class interface so it knows when we're talking about when we start trying to implement methods on the myvector class. And I'm gonna start with myvector's constructor, and the goal of the constructor will be to setup the instance variables of this object into a state that makes sense.

So what I'm gonna choose to do is allocate my array to some size. I'm gonna pick ten. So I'm gonna say I want space for ten strings. I want to record that I made space for ten strings so I know the num allocated, the size of my array right now is ten, and the number used is zero. So that means that when someone makes a call, like a declaration, that says myvector V, so if I'm back over here in my main.

Say myvector V, but the process of constructing that vector V will cause the constructor to get called, will cause a ten-member string element array to get allocated out in the heap that's pointed to by arr, and then it will set those two integers to know that there's ten of them and zero of them are used. So just to kind of all as part of the machinery of declaring, the constructor is just wired into that so we get setup, ready to go, with some empty space set aside.

So to go with that, I'm gonna go ahead and add the destructor right along side of it, which is I need to be in charge of cleaning up my dynamic allocation. I allocated that with the new bracket, the new that allocates out in the heap that uses the bracket form has a matching delete bracket that says delete a whole array's worth of data, so not just one string out there but a sequence of them.

We don't have to tell it the size; it actually does – as much as I said, it doesn't its size. Well somewhere in the internals, it does know the size but it never exposes it to us. So in fact, once I delete [inaudible] array, it knows how much space is there to cleanup.

Yeah?

Student: Are you just temporarily setting it up so the vector only works on strings?

Instructor (Julie Zelenski):Yes, we are.

Student: Okay.

Instructor (Julie Zelenski):Yes. We're gonna come back and fix that, but I think it's easier maybe to see it one time on a fixed type and then say, "Well, what happens when you go to template? What things change?" And we'll see all the places that we have to make modifications.

So I have myvector size. Which variable's the one that tells us about size? I got none used. I got none allocated. Which is the right one?

Student: Num used.

Instructor (Julie Zelenski):Num used, that is exactly right. So num allocated turned out to be something we only will use internally. That's not gonna – no one's gonna see that or know about it, but it is – the num used tracks how many elements have actually been put in there.

Then I write myvector add. So I'm gonna write one line of code, then I'm gonna come back and think about it for a second. I could say arr num used ++ = S, so tight little line that has a lot of stuff wrapped up in it.

Using the brackets on the dynamic array here are saying, "Take the array and right to the num used slot, post-incrementing it so it's a slide effect." So if num used is zero to start with, this has the effect of saying array of, and then the num used ++ returns the value before incrementing.

So it evaluates to zero, but as a slide effect the next use of num used will now be one. So that’s exactly what we want, we want to write the slot zero and then have num used be one in subsequent usage. And then we assign that to be S, so it’ll put it in the right spot of the array. So once num used is five, so that means the zero through four slots are used. It’ll write to the fifth slot and then up the num used to be six, so we’ll know our size is now up by one.

What else does add need to do? Is it good?

Student: Needs to make sure that we have that [inaudible].

Instructor (Julie Zelenski):It’d better make sure we have some space. So I’m gonna do this right now. I’m gonna say if num used is equal to num allocated, I’m gonna raise an error. I’m gonna come back to this but I’m gonna say – or for this first pass, we’re gonna make it so it just doesn’t grow. Picked some arbitrary size, it got that big, and then it ran out of space.

Question?

Student: So when the – for the vector zero, first time it gets called it’s actually gonna be placed in index one of the array?

Instructor (Julie Zelenski):So it’s in place of index zero. So num used ++, any variable ++ returns the – it evaluates to the value before it increments, and then as a side effect, kind of in the next step, it will update the value. So there is a difference between the ++ num used and the num ++ form. Sometimes it’s a little bit of an obscure detail we don’t spend a lot of time on, but ++ num says first increment it and then tell me what the newly incremented value is. Num ++ says tell me what the value is right now and then, as a side effect, increment it for later use.

Student: So the num used gets changed in that?

Instructor (Julie Zelenski):It does get changed in that expression, but the expression happened to evaluate to the value before it did that change.

Question?

Student: And is it really necessary to have myvector:: or is –

Instructor (Julie Zelenski):Oh yeah, yeah, yeah, yeah.

Student:– there any way for –

Instructor (Julie Zelenski):So if I take this out, and I could show you what happens if I do this, is what the compiler's gonna think that is, is it thinks, "Oh, there's just a function called size that take the arguments and returns them in." And it doesn't exist in any context; it's just a global function. And then I try to compile it, if I go ahead and do it, it's gonna say, "Num used? What's num used? Where did num used come from?" So if I click on this it'll say, "Yeah, num used was not declared in this scope." It has no idea where that came from.

So there is no mechanism in C++ to identify this as being a member function other than scoping it with the right names. You say this is the size function that is part of the myvector class. And now it has the right context to interpret that and it knows that when you talk about num used, "Oh, there's a private data member that matches that. That must be the receiver's num used."

So one other member function we got to go, which is the one that returns, of I do this, it almost but not quite, looks like what I wanted. So I say return arrays of index. They said get in the next and returned the string that was at that index. Is there anything else I might want to do here?

Student: Check and see if the index is [inaudible].

Instructor (Julie Zelenski):Well, let me check and see if that index is valid. Now, this is one of those cases where it's like you could just sort of take the stance that says, "Well, it's not my job." If you ask me to get you know something at index 25, and there are only four of them, that's your own fault and you deserve what you get.

And that is certainly the way a lot of professional libraries work because this, if I added a step here that does a check, it means that every check costs a little bit more. When you go in to get something out of the vector, it's gonna have to look and make sure you were doing it correctly. And there are people who believe that if it's like riding a motorcycle without a helmet, that's your own choice.

We're gonna actually bullet proof; we're gonna make sure that the index isn't whack. And I'm gonna go ahead and use my own size there. I'm gonna say if the index is less zero or it's greater than or equal to, and I'm gonna use size. I could use num used there but there are also reasons that if I use my own member functions and then later somehow I change that name or I change how I calculate the size, let's say I change it to where I use a linked list within an array and I'm managing the size differently, that if I have used size in this context, it will just change along with that because it's actually depending on the other part of the interface that I'll update.

And so I'm gonna go ahead and do that rather than directly asking my variable. I'll say if that happens I say error, out of bounds. And on that, that error will cause the program to halt right down there, so I don't have to worry about it, in that case, getting down to the return.

So I feel okay about this. Not too much code to get kind of a little bit of something up and running here. Let’s go over and write something to test. Add Jason, and here we go. Okay, so I put some things in there and I’m gonna see if it’ll let me get them back out.

And before I get any further, I might as well test the code I have. Right? This is – one of the things about designing a class is it’s pretty hard to write any one piece and test it by itself because often there’s a relationship: the constructor and then some adding and then some getting stuff back out. So it’s a little bit more code than I typically would like to write and not have a chance to test, having all five of these member functions kind of at first.

So if I run this test and it doesn’t work, it’s like well which part what was wrong? Was the constructor wrong? Was the add wrong? Was the size wrong? You know there’s a bunch of places to look, but unfortunately, they’re kind of all interrelated in a way that makes it a little hard to have them independently worked out. But then subsequent thing, hopefully I can add the like the insert at method without having to add a lot more before I test.

Okay. So I run this guy and it says Jason, Illia, Nathan. Feel good, I feel good, I feel smart. Put them in, in that order. See me get them out in that order. I might try some things that I’m hoping will cause my thing to blow up. Why don’t I get at ten? Let’s see, I like to be sure, so I want you to tell me what’s at the tenth slot in that vector. And it’s, ooh, out of bounds, just like I was hoping for. Oh, I like to see that. Right?

What happens if I put in enough strings that I run out of memory? And we talked about a thing – it’s set it to be ten right now. Why don’t I just make it smaller, that’ll way it’ll make it easier for me. So I say, “Oh how about this?” I’m only gonna make space for two things. And it just: error, out of space. Right? That happened before it got to printing anything. It tried to add the first one. It added the second one. Then it went to add that third one and it said, “Oh okay, we run out of space. We only had space for two set aside, you asked me to put a third in. I had no room.”

So at least the kind of simple behaviors that I put in here seem to kind of show evidence that we’ve got a little part of this up and running. What I’m gonna fix first is this out of space thing. So it would be pretty bogus and pretty unusual for a collection class like this to have some sort of fixed limit. Right? It wouldn’t – you know it’d be very unusual to say well it’s always gonna hold exactly 10 things or 100 things or even a 1,000 things. Right?

You know one way you might design it is you could imagine adding an argument to the constructor that said, “Well, how many things do you plan on putting in there? And then I’ll allocate it to that. And then when we run out of space, you know you’re hosed.” But certainly a better strategy that kind of solves more general case problems would be like, “Oh let’s grow. When we run out of space, let’s make some more space.”

Okay, let's think about what it takes to make more space in using pointers. So what a vector really looks like is it has three fields, the arr, the num used, and the num allocated. When I declare one, the way it's being setup right now, it's over here and it's allocating space for some number, let's say it's ten again, and then marking it as zero. Then as it fills these things up, it puts strings in each of these slots and it starts updating this number, eventually it gets to where all of them are filled. That when num used equals num allocated, it means that however much space I set aside, every one of those slots is now in use.

So when that happens, it's gonna be time to make a bigger array. There is not a mechanism in C++ that says take my array and just make it bigger where it is. That the way we'll have to do this is, we'll have to make a bigger array, copy over what we have, and then, you know, have it add it on by making a bigger array full of space.

So what we'll do is we'll make something that's like twice as big, I'm just gonna draw it this way since I'm running out of board space, and it's got ten slots and then ten more. And then I will copy over all these guys that I have up to the end, and then I have these new spaces at the end.

And so I will have to reset my pointer to point to there, update my num allocated, let's say to be twice as big or something, and then delete this old region that I'm no longer using. So we're gonna see an allocate, a copy, a delete, and then kind of resetting our fields to know what we just did. So I'm gonna make a private helper to do all this, and I'll just call that enlarge capacity here.

Question?

Student: Is this like [inaudible]?

Instructor (Julie Zelenski):Well it is – it can be, is the truth. So you're getting a little bit ahead of us. But in the sense that like, you know, if the array has a thousand elements and now we got to put that thousand-first thing in, it's gonna take all thousand and copy them over and enlarge the space. So in effect what is typically an O of one operation, just tacking something on the end, every now and then is gonna take a whole hit of an end operation when it does the copy.

But one way to think about that is that is every now it's really expensive but kind of if you think of it across the whole space, that you got a – let's say you started at 1,000 and then you doubled to 2,000, that the first 1,000 never paid that cost. And then all of a sudden one of them paid it but then now you don't pay it again for another 1,000. But if you kind of divided that cost, sort of amortized it across all those adds, that it was a – it didn't change the overall constant running time.

So you have to – you kind of think of it maybe in the picture. It does mean every now and then though one of them is gonna surprise you with how slow it is, but hopefully that's few and far between, if we've chosen our allocation strategy well.

So I'm gonna say – actually, I'm gonna do this. I'm gonna call this thing actually double capacity. So one strategy that often is used is you could go in little chunks. You could go like ten at a time all the time. But if the array's gonna get very big, going by tens will take you a lot of time. You might sort of use sometimes the indication of, "Well how big is it already?" Then if that's about the size it seems to be, why don't we go ahead and just double.

So if we start at 10, you know, and then we go to 20, then we go to 40, then we to 80, then as we get to bigger and bigger sizes, we'll make the kind of assumption that: well, given how big it is already, it's not – it wouldn't surprise us if it got a lot bigger. So let's just go ahead and allocate twice what we have as a way of kind of predicting that it's pretty big now, it might get even bigger in the future.

That isn't necessarily the only strategy we could use, but it's one that actually often makes pretty decent sense. And then at any given point, you'll know that the vector will be somewhere between full and half capacity, is what you're setting up for.

So let's do this. Oops, it's not ints; string bigger equals, and I make a new string array that is num allocated times two, so twice as big as the one we have. And now I go through and I copy. I'm copying from my old array to my bigger one, all of num used. In this case, num used and num allocated are exactly the same thing, so. And then I'm gonna reset my fields as needed.

I'm gonna delete my old array because I'm now done with it. I'm gonna set my pointer to this, so that's doing pointer assignment only. Right? So I deleted the old space, I've copied everything I needed out of it, I'm now resetting my pointer to the bigger one, and then I'm setting my num allocated to be twice as big, num used doesn't change. Okay, so went through the process of building all these things. And as noted, that is gonna cost us something linear in the number of elements that we currently have, so.

So this guy is intended to be a private method. I don't want people outside of the vector being able to call this, so I'm actually gonna list this in the private section. It's not as common that you'll see member functions listed down here. But in this case, it's appropriate for something that you plan to use internally as a helper but you don't want anybody outside just to be calling double capacity when they feel like it.

Question?

Student: So normally [inaudible] an array, you couldn't actually declare a size like that, though, right, with a variable?

Instructor (Julie Zelenski):You know I don't understand the question. Try that again.

Student: You know in this case to enlarge it, you use a variable as the – for the size of the array. When you normally declare an array, you couldn't do that, right?

Instructor (Julie Zelenski):Well if you did it –

Student: It has to be a constant.

Instructor (Julie Zelenski):So like if you used this form of an array, you know, where you declared it like this. Did you see what I just did?

Student: Yeah.

Instructor (Julie Zelenski):Yeah. That way won't work, right? That way is fixed size, nothing you can do about it. So I'm usually totally the dynamic array at all times, so that everything –

Student: So you do it when you're declaring it on a heap?

Instructor (Julie Zelenski):Yes.

Student: Okay.

Instructor (Julie Zelenski):Yes, exactly. All I have in the – stored in the vector itself is a pointer to an array elsewhere. And that array in the heap gives me the flexibility to make it as big as I want, as small as I want, to change its size, to change where it points to, you know all the – the dynamic arrays typically just give you a lot more flexibility than a static array.

Student: That's really stack heap.

Instructor (Julie Zelenski):It is. Array is a stack heap thing. When you put on the stack, you had to say how big it is at compile time and you can't change it. The heap let's you say, "I need it bigger, I need it smaller, I need to move it," in a way that the stack just doesn't give you that at all.

So when I go back to myvector.cpp, the place I want to put in my double capacity here is when num used is equal to num allocated, but what I want to do is call double capacity there. After I've done that, num allocated should've gone up by a factor of two. Space will be there at that point. I know that num used is a fine slot to now use to assign that next thing. So whenever we're out of space, we'll make some more space.

And so I'm gonna – right now, I think my allocated is still set at two. That's a fine place. I'd like it to be kind of small because I'd like to test kind of the – some of the initial allocations. So I'll go ahead and add a couple more people so I can see that I know that – at that point, if I've gotten to five, I'm gonna have to double once and then again to get there. And let's, I'll take out my error case here, see that I've managed to allocate and move stuff around. Got my five names back out without running into anything crazy, so that makes me feel good about what I got there.

So I could go on to show you what insert and remove do; I think I'm probably gonna skip that because I'd rather talk about the template thing. But I could just tell you – I could sketch the [inaudible]: what does insert at do, what does remove at do? Basically, that they're doing the string – the array shuffling for you. If you say insert at some position, it has to move everything down by one and then put in there. Whereas at is actually just tacking it onto the end of the existing ones. The insert and remove have to do the shuffle to either close up the space or open up a space.

They'll probably both need to look at the capacity as well. That the – if you're inserting and you're already at capacity, you better double before you start. And then the remove at, also may actually want to have a shrink capacity. Where when we realize we previously were allocated much larger and we've gotten a lot smaller, should we take away some of that space.

A lot of times the implementations don't bother with that case. They figure, "Ah, it's already allocated, just keep it around. You might need it again later." So it may be that actually we just leave it over-allocated, even when we've deleted a lot of elements, but if we were being tidy we could take an effort there.

What I want to do is switch it to a template. So if you have questions about the code I have right here, now would be a really good time to ask before I start mucking it up. Way in the back?

Student:[Inaudible].

Instructor (Julie Zelenski):I will. You know that's a really good idea because if I – at this point, I'll start to change it and then it's gonna be something else before we're all done. So let me take a snapshot of what we have here so that I – before I destroy it.

Question?

Student: When does the deconstructor get called?

Instructor (Julie Zelenski):Okay, so the destructor gets called in the most – there are two cases it gets called. One is when the – so the constructor gets called when you declare it, and then destructor gets called when it goes out of scope. So at the opening brace of the block where you declared it in, is when the constructor's happening, and then when you leave that. So in the case of this program, it's at the end of May, but and if it were in some called function or in the body of a for loop or something, it would get called when you enter the loop and then called as – destroyed as it left.

For things that were allocated out of the heap, so if I had a myvector new myvector, it would explicitly when I called delete that thing when I was done with it. So especially when the variable that holds it is going away, right? Either because you're deleting it out of the heap or because it's going out of scope on the stack.

Here?

Student: When you have string star array, arr, it's a pointer to a single string and then later you can use a new statement to link it to –

Instructor (Julie Zelenski):Yeah.

Student:– an array. So how does it know when it –

Instructor (Julie Zelenski):How does it know? It doesn't, is the truth, is that when you say something's a pointer to a string, the only guarantee is really there and says it's an address of someplace where a string lives in memory. Whether there's one string there or a whole sequence of strings is something you decide and you maintain, and so the compiler doesn't distinguish those two cases. It does not know that you made exactly 1 string at the other end of this or 500.

And so, that's why this idea of tracking the num used and num allocated becomes our job; that they look exactly the same in terms of how it interprets them. It says, "Well it's the address of where a string lives in memory, or maybe it's a sequence. I don't know." And so it lets you use the brackets and the new and stuff on a pointer without distinguishing – having any mechanism in the language to say this is a single pointer and this is an array pointer. They look the same.

Student: So you could use the bracket notion on a single pointer?

Instructor (Julie Zelenski):You certainly can.

Student: And then –

Instructor (Julie Zelenski):It's not a good idea but you can do it. So legally in the language, it just makes no distinction between those. They are really commingled in way that – and so that's one of the more surprising features of C and C++ and one's that a little bit hard to get your head around is it doesn't track that it's a pointer to a single thing versus a pointer to an array. That they're both just pointers and it's mine. And that allows opportunity for errors, when you mistreat them as the other type for example.

Question?

Student: Can you go over in your secret view file in the [inaudible] of what's going on with [inaudible] real quick because it's just –

Instructor (Julie Zelenski):Yeah, yeah, yeah, so let me – it was basically the thing I drew over here, but I'll do it again just to watch it, is that let's start – let's imagine I have a slightly smaller num allocated so it's a little less for me to write.

So let's say that I'm gonna use a num allocated of two, so this allocates two. So when I construct it, it makes a block that holds two things and num used is zero. So I do two adds: I add A, it increments num used to one; I had B, it increments num used to two. I try to add C. It says, "Oh, well num used equals num allocated. We're gonna go to double capacity now."

So double capacity has this little local variable called bigger. And it says bigger is gonna be something that is four strings worth in an array, so it gets four out there. It does a full loop to copy the contents of the old array on top of the initial part of this array; so it copies over the A and the B, into there. And then it goes, "Okay, I'm done with this old part. So let me go ahead and delete that."

And then it resets the arr to point to this new one down here, where bigger was. So now, we got to aliases of the same location. And then it sets my num allocated to say and now what you've got there is something that holds four slots. And then that used call here says, "Okay and now writer the C into the slot at three."

So the process here is the only way to enlarge an array in C++ is to make a bigger one, copy what you had, and then by virtue of you having made a bigger array to start with, you have some more slack that you didn't have before.

Daniel?

Student: How does it delete arr with a star?

Instructor (Julie Zelenski):You know it has to do with just delete takes a pointer. It does it – so a star arr is a string, arr is a pointer to a string. So both forms of delete, delete and delete bracket –

Student: So conceptually –

Instructor (Julie Zelenski):– a pointer.

Student:– there is a start there because it's delete –

Instructor (Julie Zelenski):Well effectively, yeah. It's delete the thing at the other end of the pointer, really. But it's funny. Delete says take this address and reclaim its contents. And so it doesn't really operate on a string, per se, it operates on the storage where that string is. And so I don't know whether you want to call that is there an implicit star there or not, it really is about the pointer though rather than the contents. So saying that address has some memory associated with it, reclaim that memory.

Student: If I could raise –

Instructor (Julie Zelenski):Uh-huh.

Student: So when you're first declaring or when you're making a pointer like string bigger, string star bigger, you have to declare it with the star notion. But then later on, you don't ever have to use that again?

Instructor (Julie Zelenski):You pretty much won't see that star used again. Right? It's interesting that things like bigger sub I and erase sub I implicitly have a D reference in them. And that can be misleading. You think, "Well how come I'm never actually using that star again on that thing to get back to the strings that are out there?"

And it has to do with the fact that the bracket notation kind of implicitly D references in it. If I did a star bigger, it would actually have the effect of giving me bigger sub zero, it turns out. You can use that notation but it's not that common to need to.

Student: And so down on the last, one line up from the bottom, it says array equals bigger. You don't have to –

Instructor (Julie Zelenski):Yeah, if you did that, if I did say –

Student: If you said array –

Instructor (Julie Zelenski):Star arr equals star bigger, I would not be getting what I want. Right? What it would be doing is it would say follow bigger and see what's at the other end, so that would follow bigger and get that string A. And then it would say follow ARR and overwrite it with that A, so it would actually have the effect of only copying the first string from bigger on top of the first string of array. But array would still point to where it was, bigger would still point to where it was, and they would – we would've not have updated our, the pointer we really wanted to point to the new array.

So there is a difference. Without the star, we're talking about the changing the pointers; with the star, we're talking about the strings at the other end. And so we're – this is a string assignment. It says assign one string to another. Without the star on it, it's like assign one pointer to another; make two pointers point to the same place. When you're done with this, bigger and arr will be aliases for the same location. That's a very important question though to get kind of what that star's doing for you.

Here?

Student: After arr is bigger, can you delete bigger after that?

Instructor (Julie Zelenski):If I deleted bigger, at that point arr is pointing to the same place. And so remember that having two or three or ten pointers all at the same place, if you delete one of them, they actually effectively are deleted. The delete really deletes the storage out here. And then if I did that, it would cause arr to then be pointing to this piece of memory, and not a good scene will come from that. It means that when it later goes back in there and starts trying to read and write to that contents at any moment it could kind of shift underneath you. You don't own it any more; it's not reserved for your use.

So if we did that, we’d get ourselves into trouble. All right? So there should basically be a one-to-one correspondence between things you new and things you delete. And so in the myvector case, we newed something in the constructor that we’re gonna delete in the destructor. If at some point along the way we got rid of our old one and get a new one, that’s the new the one that’s gonna get deleted. If we deleted it midstream here, we would just be asking for havoc when we start accessing that deleted memory.

Way in the back?

Student: Is it possible to create a pointer to something – a pointer to the address that’s one off the end of the original array and then just create an array just off the end there?

Instructor (Julie Zelenski):Not really. So new doesn’t let you decide where you want something. So you’re point being to think, “Well I can tell you what this address is, can I just make space right there and then I won’t have to copy.” And it turns out new just doesn’t give you that kind of control. You ask it for space, it finds it wherever it has and you can’t – there isn’t even a mechanism where you could suggest where you’d like it to be. You could say, “Well let that place right there would be really handy. Could you please give me that one?” It just doesn’t give it you.

So you’re – this is the way that typically you have to manage a dynamic array. And this is actually one of the big drawbacks to continuous memory as a reason for implementing things is that the fact that it has to maintain contiguousness. Means you have to shuffle and move and copy this block without the flexibility of something like a link list where every cell is independently manipulated.

There?

Student: Why does [inaudible] the delete brackets arr as delete just arr?

Instructor (Julie Zelenski):So the difference is that if you allocated something with new string bracket, new something bracket, you need a delete bracket. If you actually use delete without the brackets, it thinks there’s a single pointer and there’s only one string at the other end. Where delete brackets says delete the whole gob of strings that are there.

If you don’t do it, it’s not – the consequence is not that big; it’s like some memory gets orphaned, some things don’t happen. But to be totally correct, they go hand in hand: if you use new with brackets, use delete with no brackets, if you use new with brackets, use delete with brackets. So it’s either both with brackets or both without.

Student: So even though arr is just point to the first place, the brackets knows the –

Instructor (Julie Zelenski):Yeah, it does. And so that kind of makes me feel like I’m a lair because I said well the array doesn’t know its length. Well it does somehow. Internally it is maintaining some housekeeping but it doesn’t expose it to you. So when you say delete bracket arr it knows, “Oh, there’s a bunch of strings and I got to do a bunch of cleanup on them.” But it doesn’t ever expose that information back to you. It doesn’t let you depend on it, so it’s up to you to maintain that information redundantly with it.

All right, let me see if I can make it a template. I probably can’t do this actually fast enough to get it all done today, but we can at least get started on it. So then, I introduce a template header and I make up the name that I want here, so same class header now other than typing elem type. Then I look through my interface and I see places where I previously had said it’s strings, it’s strings, it’s storing strings. And I say it’s not actually storing strings; it’s gonna store elem type things, it’s gonna return elem type things and it’s going to have an array of elem type things.

So I think that’s everything that happened to the interface. Let me see if I see any other places that I – so the interface part is kind of small. There’s one other change I’m gonna have to make to it but I’m gonna come back to it. I’m gonna look at the code at the other side for a second. And I say, “Okay, well that wasn’t so bad.”

Now it turns out that it gets a little bit goopier over here because that template type name has to go on every one of these: introduce them to the template type and elem type. And now there’s another place where it needs to show up. So the full syntax for this is now saying this is a template function, depending on elem type, and it’s actually for the myvector who is being – we are writing the myvector constructor for something whose name is myvector angle bracket elem type.

So there’s gonna be a lot of this goo. Every one of these is kinda change its form, from just looking like the ordinary myvector class scope doesn’t really exist any more. Myvector is now a template for which there’s a lot of different class scopes, one for each kind of thing being stored. So myvector int is different than myvector string. So we say, “Well, if you were building the myvector constructor for myvector string, it looks like this.” Or you know having filled an elem type with those strings.

So everywhere I was using string, I got to change to elem type in the body as well. And then I kind of take this guy and use it in a bunch of places. I’m gonna use it here and then I’m gonna have to do it down here, on that side, do it here, and it’s gonna return something of elem type, here. It’s a little bit of a mess to do this, and the code definitely gets a little bit goopier as a result of this. It doesn’t look quite as pretty as it did when it wasn’t a template, but it becomes a lot more useful.

Okay. Then I need to look for places that I used a string. And every place where I was using string, assuming that’s what I was storing, it now actually turns into elem type. So my pointers and the kind of array I’m allocating is actually now made into elem type. The rest of the code actually didn’t say anything specific about what’s its doing, just copying things from one array to another. And now, depending on what the arrays are, it’s copying ints or strings or doubles.

And then other places in the interface where I'm doing add or I'm going get at, I have to be describing the things that are coming in and out as elem type so that they can be matched to whatever the client's using. I think the rest of it looks okay.

Student: Why do you have to write template type name, and elem type above every –

Instructor (Julie Zelenski):Because you just have to, because it's C++. Because the thing is, that piece of code is, itself, a template, so these are like little mini-templates. So that I had the interface, which said here's the template pattern for the interface, and each of these says when you're ready to make the size member function for a vector of int, it comes off this template. So this template describes what the size member function looks like for any of the myvectors you might instantiate. And it describes the template because, in fact, we need to build a new size for ints versus doubles versus strings.

It's even funny because you think of my size like, "Well size doesn't even use anything related to the elem type." But in fact, each of the member functions is kinda specific. It's not just a myvector size; it's the myvector int size, the myvector string size. And that for some of the member functions it's quite obvious why you need a distinct copy. Get at returns an int in some cases and a double in others; but even though ones that don't appear to have any dependence on the elem type, actually are separated into their own individual versions.

So I think I got all of that fixed, and then I'm gonna have to do one thing that's gonna seem really quirky. And it is very quirky but it is C++. Let me show you what I'm gonna do. Is I'm going [inaudible] out of the project. Okay, stop compiling that. And I'm gonna change how it is that myvector gets compiled by doing this.

Okay. Take a deep breath. This is really just an oddity of C++. So the situation is this: that templates aren't really compiled ahead of time, templates are just patterns. You know? They like describe a recipe for how you would build a myvector class. But you can't just compile myvector and be done with it because until the client uses it, you don't know what kind of myvectors you're building. Are they myvectors of ints or strings or pseudo structures?

So it turns out that the myvector really needs to get compiled at the usage, at the instantiation. When you're ready to make a myvector of students, it then needs to see all the code for myvector so it can go build you a myvector for students on the fly. In order to see that code, it actually has to be present in a different way than most code.

Most code is compiled, instead of .cpp, it just gets compiled once and once for all. The random library, random integer doesn't change for anybody usage, there's a random.cpp. It compiled the function. You're done. So the template code does not get compiled ahead of time. It doesn't get listed in the project. What happens is the .h typically has not only the interface, but actually all the code.

And so the two ways to get the code in here, one way is I could've just put all the code down here. And that's the way a lot of professional code gets written, it has the interface followed by all the template code right after it. I like to keep us thinking about the interface and the implementation being separate, so I'm actually taking the interface and keeping the .h, keeping this [inaudible] over here in a .cpp.

And then I'm using the #include mechanism in a very unusual way. That almost never would you want to, in a regular usage, to #include another .cpp file. For templates, we're making an exception. And we're saying, "Well in this case, because I really need that code there," the #include mechanism is basically saying go take the contents of this thing and just dump it in here.

It really is an include mechanism. It says, "Go get this file and take its text contents and dump it right into this file." So that when somebody's trying to import the myvector.h, they're getting both the interface plus all the code that we'll generate a pattern from.

So this is definitely just a quirk. There's no consistency between how other languages that do stuff like this expect this. This is just unique to C++ and its compilation mechanisms that require this kind of sort of slight variation in handling. So we'll see this for all the templates we'll use is that they will not be included as normal cpp files, they will get included in the .h. And there is this exact pattern, which is reproduced for every one of the ones in the text. You'll see it on stack and queue and integer. That it becomes the kind of boilerplate we'll use when making a template.

So in general, I'd say be very wary of anything that looks like this. This is not a normal thing to do and we're doing it just specifically to kind of keep up the illusion that the interface and the implementation are kept separate because there's actually some good thinking that comes from that. But the way the compiler sees it, it doesn't want them to be separate, and so we have to accommodate it with this little hack, let's say, here.

So once I've done that, I go back to lecture. If I change this to be myvector string, I'm hoping that everything will still work. Which it did, kind of amazingly.

Daniel?

Student: So where is the myvector.cpp file at?

Instructor (Julie Zelenski):So it's actually just living in the same directory with this, the way myvector.h is. So typically, like your .h files are just sitting in the directory – .cpp is sitting in the same directory. That's where it's gonna look for it when it goes #including is in the kind of local contents.

Student: But like where is that? Like is it in resources?

Instructor (Julie Zelenski):No, it's just – if you look – you know this is the directory I have, this is the contents of my, you know all my files, my project files, where the thing gets dumped.

Student: Oh, okay.

Instructor (Julie Zelenski):It's just sitting there with all the code.

And I should be able to change this now to put some numbers in and have it do both. I just did it with strings and now I'm gonna do it with ints. And voila, we now have something that holds ints. So a certain amount of goo that went from the simple form to the template form, but a lot of power gained. Suddenly we took this thing that was one purpose, that held strings only, and you just made it to where it can hold anything you can think of to stick in there by making little machinations in the syntax there.

So we'll see a lot more of this. It's not the first, nor the last, syntax that – for templates that we're gonna be playing with, but that will be next week. Come to Café with me, if you got some time. Otherwise, I will see you on Monday.

[End of Audio]

Duration: 53 minutes