ProgrammingAbstractions-Lecture05

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Student:[Inaudible]

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

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

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

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

Student: Is there a remove all method?

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

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

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

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

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

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

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

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

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

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

Student:[Inaudible]

Instructor (Julie Zelenski):The square brackets.

Student: No, the [inaudible].

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

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

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

Student:[Inaudible]

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

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

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

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

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

Student:[Inaudible]

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

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

Student:[Inaudible]

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

Student: Ten in that case means what?

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

Student:[Inaudible]

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

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

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

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

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

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

Student:[Inaudible] double bracket number and then –

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

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

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

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

Student: Same question.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Student:[Inaudible]

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

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

Student: So there's no square bracket operator?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Student:[Inaudible]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[End of Audio]

Duration: 48 minutes