ProgrammingAbstractions-Lecture12

Instructor (Julie Zelenski):Hey there. Okay, we'll start with talking about just where we're at in terms of material, right, as we're gonna keep talking about pointers, which we just got started with on Monday, and go on to use them to do some stuff with recursive data in the form of a link list.

The reading is a little bit scattered at this point, mostly. Handout 21 is probably the best resource for the material I'm talking about today, and then once we work through the link list material we'll go back to the text and start looking at chapter 7 and fall by chapter 8 in order.

We'll spend a couple days on chapter 7 though, because there's a lot to talk about there in terms of algorithms and analysis and sorting and things like that.

[Inaudible] today, and some [inaudible] going out, so first off let's ask a little bit of joy pull, how much joy there was in the recursion assignment. Let me first see if anybody wants to own up to having spent less than 10 hours total getting all those problems working.

All right, that's good. That's a good sign, that you must really kind of have your recursion head on. Ten to 15 hours? Another big, healthy chunk there. Fifteen to 20? Oh, a little bit of that. And then more than 20?

And so in terms of sort of hours spent per line written, this probably had sort of one of the highest ratios you've seen, right? You know, you work on it 10 hours and you write four lines. The output field's a little bit sparse relative to the effort that went into writing it.

That's a very sort of symptomatic of kind of what recursion is like, right? It is an elegant and direct way of solving the problem, but a lot of the work gets spent in just figuring out what that formulation is, and once you have the formulation, getting it down in code is not the time-consuming part. There's some details to handle and things like that.

But your bigger time spent certainly should be conceptual, kind of figuring out how to break it down and get recursion to work for you.

The next assignment going out is Boggle. Boggle is a legacy of the CS106 program. I assigned Boggle for the very first time in 1994, so we have actually had Boggle kind of ongoing, on and off, for almost 15 years, and every time we've tried to replace it with another kind of big recursion program, we're always dissatisfied and we come back to Boggle, because we think it's just one of the best programs to have at this point in the quarter. So you're seeing kind of a long tradition here.

I'll tell you what I think is really neat about Boggle to kind of get you psyched. We give you a little bit more time to work on it, because actually this is your first big, complete

program, right? Not a bunch of little parts, not a little problem to solve here, one big thing to do, which is to write a program that plays Boggle against a human.

Lets the human find words, lets the computer find words, kind of head-to-head. It uses recursion in two ways, both in doing searches for the human as well as doing searches for the computer, so it actually has a chance to kind of flex those recursion muscles that you've been working on this week.

It uses some of the ADTs we've seen so far, the grid and the lexicon come into play; potentially some of the others as well. A lot of different pieces to kind of bring together, you'll do some interacting with the user. There's a little bit of drawing. Although we did most of the drawing code for you, you have to kind of interact with our library to get those things up on the screen.

And a lot of pieces to kind of bring together. So at this point we're at a really good milestone for, you know, you're feeling good, you've got a handle, it kind of touches on everything we have talked about so far, right, shows up in this program, and it's your first kind of big, complete kind of put the whole thing together, and that means it's a really good opportunity to also be very conscious of style.

On the recursion problems, certain things that are smaller, the opportunities to distinguish good stuff from bad stuff aren't that broad. But in the context of a whole program, there's actually a lot of choices you can make that can turn out well or poorly, depending on how much time and care you're putting into those early decisions, right, can really make a big difference in the long run.

So this is one where I think up front thinking and planning and kind of doing a good job with the style really pays off in terms of getting an easier ride all the way to a good solution.

It's also really cool when you're done because you write this program that, in fact, right, will kick your butt every time you play it, and I think that's actually a real important milestone for you, is to write a program – at some point you have to write programs that do things you can't do yourself, right, that you can't do by hand, to realize that you're capable of harnessing this power to write things, right, that extend beyond the capabilities of you yourself I think is a real significant accomplishment, and one that you can take a lot of pride in.

And so when you get to where you have a working Boggle and you're playing it and it's beating you every time, you don't have to feel bad about your vocabulary, what you get to feel good about is your programming skills in having done that.

And so every year I always get a couple of people who give me screenshots of them actually beating the computer. Because it's so rare, it's worthy of recording to prove.

We used to keep a machine, actually, in my office a while ago that was just running a game of Boggle and my officemate and I would occasionally go over and look at it and type in words, right? And then we'd just let it sit there. Like, you can take as long as you want, that's the way – the game is stacked for you. As patient as you are, you can win if you're just willing to sit and work on it for a very long time.

So we'd leave it there for days. Students would come in and help us, we'd keep finding one word at a time, one word at a time, you know. After we'd had it running for a week or two we'd finally say okay, we found all the words, there are no words left. And then we would let the computer's turn go and it would find all the words we had not found, and then we would feel dumb. But we would keep trying, and every now and then we would win.

A little note about paper copies, right, is that we are asking you to turn in both an electronic copy and a paper copy. The paper copy serves as the place where we're writing a lot of the comments and working with you on the IGs. It is important that you turn in both, and the reports back from the section leaders is that there's been a little bit of delinquency on that. That pretty much everybody's getting an e-submit in, but they have been a little bit sloppy about following through on the paper copy.

We have not been making a big deal out of it, but we do plan on being a little bit tighter about that in the future. So if you have been taking advantage of us and turning them in kind of when and where you feel like it or not at all, do be aware that we will start enforcing a late-day policy, that if the paper copy comes in late, right, then we will be calling it late, even if the electronic submit was made on time.

So there really is no reason to – you know, once you've got the e-submit, just print it right out, bring it on in, and all is well.

All right, anything else kind of on the administration angle that I could – okay.

Now I'm going to show you a movie that's going to make everything about pointers clear, okay? All right, speaking of that officemate and I who I used to play Boggle with, this is Nick Parlante, who actually is now my next-door neighbor in the Gates building. And he in 1999, with a little too much time on his hands, did a little experiment in Claymation that will reveal all the mysteries of pointers in three minutes. Here we go. [Video]

Hey, Binky, wake up, it's time for pointer fun. [Video]

What's that? Learn about pointers? Oh, goody. [Video]

Well, to get started, I guess we're gonna need a couple pointers. [Video]

Okay. This code allocates two pointers, which can point to integers. [Video]

Okay, well, I see the two pointers, but they don't seem to be pointing to anything. [Video]

That's right. Initially, pointers don't point to anything. The things they point to are called pointees, and setting them up is a separate step. [Video]

Oh, right, right, I knew that. The pointees are separate. Er, so how do you allocate a pointee? [Video]

Okay, well, this code allocates a new integer pointee and this part sets X to point to it. [Video]

Hey, that looks better. So, make it do something. [Video]

Okay. I'll dereference the pointer X to store the number 42 into its pointee. For this trick, I'll need my magic wand of dereferencing. [Video]

Your magic wand of dereferencing? That's great. [Video]

This is what the code looks like. I'll just set up the number, and – [Video]

Hey, look, there it goes. So doing a dereference on X follows the arrow to access its pointee – in this case, to store 42 in there. Hey, try using it to store the number 13 to the other pointer, Y. [Video]

Okay. I'll just go over here to Y and get the number 13 set up, and then take the round of dereferencing and just – ooh. [Video]

Oh, hey, that didn’t work. Say, Binky, I don't think dereferencing Y is a good idea, because, you know, setting up the pointee is a separate step, and I don't think we ever did it. [Video]

Hm, good point. [Video]

Yeah, I mean, we allocated the pointer Y but we never set it to point to a pointee. [Video]

Hm, very observant. [Video]

Hey, you're looking good there, Binky. Can you fix it so that Y points to the same pointee as X? [Video]

Sure, I'll use my magic wand of pointer assignment. [Video]

Is that gonna be a problem like before? [Video]

No, this doesn’t touch the pointees, it just changes one pointer to point to the same thing as another. [Video]

Oh, I see. Now Y points to the same place as X. So wait, now Y is fixed, it has a pointee. So you can try the wand of dereferencing again to send the 13 over. [Video]

Uh, okay. Here it goes. [Video]

Hey, look at that. Now dereferencing works on Y. And because the pointers are sharing that one pointee, they both see the 13. [Video]

Yeah, sharing – uh, whatever. So are we gonna switch places now? [Video]

Oh, look, we're out of time. Just remember the three pointer rules. Number 1, the basic structure is that you have a pointer, and it points over to a pointee. But the pointer and pointee are separate, and the common error is to set up a pointer but to forget to give it a pointee.

Number 2, pointer dereferencing starts at the pointer and follows its arrow over to access its pointee. As we all know, this only works if there is a pointee, which kind of gets back to rule number 1.

Number 3, pointer assignment takes one pointer and changes it to point to the same pointee as another pointer. So after the assignment, the two pointers will point to the same pointee. Sometimes that's called sharing.

And that's all there is to it, really. Bye-bye, now.

Instructor (Julie Zelenski):There you go. Whoo. So if you ever wanna meet Nick Parlante, who lives next door to me in the Gates building, you can tell him that you enjoyed his clay movie and that all your mystery of pointers has been cleared up.

All right, let me – let me show you some code. I'm going to keep talking about the pointer idea, draw some pictures to try to get a visualization of what's going on, and then I'm going to go on to show you some of the things you do with pointers. What are pointers good for, right? Right now, it seems like it's kind of an obscure way to get at things you already know how to do. I'm going to show you some of the things that then it enables you to do that you couldn't do if you didn't have the pointer around.

So I've got a little bit of code, right, that declares some variables, and I'm gonna draw the same picture I had at the end, right, that what is the stack frame for main look like? It's got an integer num, and it has two pointer variables, P and Q, and those things actually all live in the stack.

So stack variables are those that you declare kind of as you open a scope, as you enter a function, things like that. Space for them is automatically allocated and de-allocated. You never see explicit news or deletes on those that are there. Those pointers, as they are set up here, have not been initialized.

So just like num has some junk contents, right, so if I have just a declared numb and then I do cout, num, endl, then I'll get some junk number. It might be zero, it could be 6,452, it could be negative 75. We don't know what it is, right, but accessing it, right, is technically legal, right? It won't complain to you about it, right, but it's unlikely to produce any valid result.

Similarly with pointers, if I say P and Q, right, without saying what they're assigned to, then these have junk addresses in them – junk numbers. Now this turns out to be a little bit more dangerous, the use of an uninitialized pointer relative than an uninitialized variable, that if I were just to look at that number, that's unlikely to get me into trouble.

If I try to follow that address as though it were a good address, that's when you end up with Binky's head getting blown off. If I've done nothing to P and Q and I say star P equals 42, it takes this junk contents, which is, like, some address to somewhere, and follows it and tries to write a 42.

There's no guarantee, for example, that that number, that address there is valid, that it actually makes sense, that it actually even kind of exists in terms of the process space so far. It's kind of like calling a phone number just by dialing random digits on the phone.

It's, like, more likely than not, right, you're gonna get the, you know, that number's been disconnected. It might be that you'll randomly call somebody and start having a phone conversation with them. The same with this 42. It's like it might be that it'll write somewhere and start storing a 42, but on top of something that wasn't intended to have that kind of adjustment made to it.

So it's a very dangerous operation, right, that can lead to strange results, immediate crashes, you know, other strange results later on, when you didn’t plan on it, other things changing that you weren’t expecting because of this improper access.

So P and Q, right, sort of that – if I see P equals new int, then over here in the heap, so stack here, heap over here, P points to a new integer variable stored out in the heap, and now that's star P equals 10.

So that word pointee, that's a Nick made-up word. No – you won't hear that pretty much anywhere other than the Binky pointer video, and sometimes I use it too, because I think it's a cute word. But it is not quite the official term for that, so P pointing to this integer out in the heap. And then I have these two lines that I talked about last time that I want to review what's going on that star P, right, it's a star Q, any time you see the pointer with that star on it, right, that's applying the dereference.

And so it's saying we're not talking about manipulating the pointer itself, but what's at the other end of it. So reaching out to the integer. So the way P is declared there, I want to say P is a pointer to an integer, then the type of the expression star P is integer type. It refers to the integer that it is pointing to, which in this case is out in the heap.

And so if I said Q equals new int, it gets a space out in the heap. And then when I say star Q equals star P, that says right to the integer out here, having read an integer from there to copy. So it copies the integer value that P is pointing to on top of the integer value Q is pointing to. So copying the integers.

The next line, where the stars are not present, where it just says Q equals P, that is doing just pointer assignment. In this case, it's manipulating the pointers themselves, not what they point – the integers that they're pointing to. So when I say Q equals P, it effectively copies the address that P has.

So P is pointing to the address, you know, 30,012, and so what's really at sort of the lowest level is actually written this number here. Then it writes that same number in this box, which causes Q to point to the same place as P now. And so changing star P, changing star Q, actually both of them actually are sharing or aliasing, and so there's only one integer that now we have two different ways of getting to via these two different pointers.

So here's a little trick about C++, right, that's unique to those of you who've come from the Java world, is that it is your responsibility to deallocate things that come out of the heap. So whereas on the stack memory is allocated and deallocated automatically, everything in the heap is explicitly allocated and explicitly deallocated.

If I say new, it made some space and set it aside and reserved it for my purpose in the heap, great. When I'm done with it, when I no longer need that piece of memory, I've discarded it, I've, you know, removed it, that student graduated, whatever it is that causes me to no longer need it, then it is the responsibility of the programmer to issue the delete call using the delete operator on the pointer to say, follow this pointer and take the space that it's currently pointing to and mark it as reclaimable – no longer in use, ready to be vacated for someone else's purposes.

So when I say delete P, it causes the heap manager to say, oh, okay, this address is now free or available. As a consequence of that, it may or may not actually do anything right away with that piece of memory. It might be that what it just did was mark it as available, kind of it's sort of like having a list of vacancies in an apartment building.

You might say okay, well, I'm moving out of apartment 100. You might say oh, okay, well, apartment 100 is empty. It may go in and clean the apartment, it might go in and kind of write something over it, it may not. There's actually no guarantees there. But it does mean that now if somebody comes in and wants an apartment, right, we could give them that one. So if somebody asks for a new integer, we might give them that address, 30,012, because we no longer have it marked as being in use.

So if you were to delete something and then start to use it again, so if I deleted P and then tried to do this – star P, you know, equals 100, I'm asking for trouble. That piece of memory has been deallocated, it might be in use somewhere else. At this point, I don't

own it anymore, I don't have that reservation. It's kind of like using your keys to get back into that apartment that you've moved out of.

It's like well, someday, somebody else's stuff is going to be in there, right? And you are squatting, right? You can be arrested for that. Well, in C++, right, you can receive program crashes and other kind of mystical behavior from that, and so it is something we have to be a little careful about.

When we're done we say delete, but not before, and that once we have said delete we don't reference that piece of memory after it's been deleted.

I put a little note here. I said, well, delete Q. So here's a thing to note, is that we delete things that we allocated new on. It is not the case that we delete every pointer, though. If we have five pointers that point to one piece of memory, if I delete it, no matter which pointer, you know, I used when I made the delete call, they all hold the same address, and I deleted what was at that address.

So if I've deleted the address 30,012, no matter how many other pointers hold that same address and point to that same place, I do not make multiple delete calls. So in this case, if I did delete P and delete Q where they're currently aliased, right, I would be trying to delete a piece of memory that's already deleted.

Again, no guarantees about whether this will produce an immediate bad effect or some later bad effect, or maybe just turn into a no-op, but no matter what, it's not a good practice to get into of accidentally deleting things more than once.

So a little bit more management on our part, right, than we're used to in Java, and then that last little bit of code I showed at the bottom, right, was just the use of Null, capital N-U-L-L, which is the C++ equivalent of the Java lowercase null, it's the zero pointer, and it's used as the sentinel value. Typically when you have a pointer that you know doesn't point anywhere, you use that to say, well, here it is, I'm going to initialize it to Null so I know that that's the known sentinel value that says there's no current address that it's holding.

So very much some of the things you've already seen in Java, right, this idea of having declaring things, and calling new to get new things that are out in the heap. The delete step is certainly new. It's one we'll work our way up to. To be honest, to be fair, if you just had no delete calls in your program, the consequence would be that pieces of memory like this one that I allocated under Q and then later lost track of would be orphaned. And so we'd end up having a bunch of memory that we had set aside and reserved for our use that we weren't currently using.

If we kept doing that over a long period of time, we'd eventually kind of clog the heap with a bunch of waste that would not be reclaimable and it could slow the program down and even in an extreme case, right, could cause your machine to have significant performance problems.

For the shape of programs we write and the size of them and the amount of running time they are, not having deletes in them is actually kind of not a particularly tragic result. So often, right, we'll encourage you to write programs without delete in them where you make news, and work your way up to putting deletes in, because in fact, right, they're tricky to get right and if you get them wrong, the consequences are sort of more deadly than just having the orphans get filled up in your heap.

All right, so any questions on kind of the basic little set up here?

Student: So the pointee or whatever, of Q original one?

Instructor (Julie Zelenski):Yeah?

Student: There's no way to delete it anyway?

Instructor (Julie Zelenski):There's no way to get – I lost its address, right? I didn't store it anywhere, so it gave me this address. Maybe this is a, you know, address, 48,012, that it gave me that address, I wrote it down, right, and then I overwrote it with something else. And so I no longer have it. So unless I put it somewhere, right, I have no way to know where that piece of memory is.

Student:[Inaudible] Why wouldn't it be, like, delete star P?

Instructor (Julie Zelenski):Well, you know, it's interesting, it kind of feels a little bit more like it's delete star this, right? It's like delete was at the other end. But it just happens to be that the operator takes the address itself, not the number that's at the other end or the string or whatever it's a pointer to, so, the operator.

So it takes a pointer and it says take the contents at that address and mark it as available for someone else's use.

Student: Is P [inaudible]?

Student: So P still holds the same value. That's a great question, right? So P had 30,012. I said delete 30,012. The heap manager took that request and kind of marketed it, but P still holds that address. And so if I did a star P equals 100, it tries to right back there. So in fact the delete operation doesn't change P. So it still holds the same address.

So to be really careful about it I might want to say if I set it to delete P, I could set P equal to Null right after it, kind of as a safety catch for myself, to remind myself that whatever number it used to have is no longer valid. That address, right, is no longer mine to reference.

So as a habit, some programmers do do that. Every time they make a delete call, the next thing they do is set that pointer to Null so that they don't actually even have a chance of reusing that address that no longer is valid.

So this kind of control actually is really valuable. At first glance it just seems like it's like extra work. It's like well now not only do I have to allocate, I also have to deallocate. But that gives you actually very good control of exactly when it got allocated and exactly when it got deallocated, and that means you can control the memory usage of your program very precisely in a way that in Java, Java doesn't actually let you get involved in this.

You can allocate things, but it decides when and where it's going to start doing deallocation. It may wait until it thinks that there's a need to do this what's called garbage collection, it may do it a little bit all the time, it may do a big bunch at one time. And it doesn't give you control in the way that C++ really just leaves it up to you.

You decide when you needed something and when you're done with it, which does actually, for professional programmers, considered a real advantage in very precisely dictating about how memory gets used, and when and where.

So a couple things that I just – things I wanted to remind you before we went on to do some other things, right, is that pointers are distinguished by the type of the pointee. That you can have a pointer to a double, a pointer to a string, a pointer to a scanner. You know, any of the types you have seen, right, there is a corresponding star of that type that is a pointer to one of those things. There are, in fact, right, pointers to pointers.

We're not going to see that too commonly in this quarter, but there are, for example, int double stars, which is a pointer to something which is a pointer to an integer. That has kind of two arrows that lead to an integer, finally.

All of these pointers, right, are distinguishable types that if, you know, two things that both point to double are the same type of pointer, but a pointer to a double is not the same thing as a pointer to an integer, which is not the same thing as a pointer to a string. So actually, the type system considers these different kinds of pointers to be different.

In truth, right, the way they're being managed behind the scenes is they are all just addresses, but it is important to know that what's at the end of that address is an int as opposed to what being at that address as a string. And the compiler does strongly enforce this idea that if you told me what's at that address is a string, then it wouldn't be appropriate to suddenly start treating what's at that address as though it were an integer. So it actually doesn't allow you to kind of mix those types up.

Now pointers are uninitialized until they're assigned. When you declare a pointer, right, you just get a junk address dereferencing it either to read or to write some of that contents, right, is just asking for bad trouble.

The consequences are not that predictable, so one of the things I had tried to say last time was one of the reasons I think pointers are considered a little bit scary is because there are certain opportunities to make mistakes here, and those mistakes have pretty drastic consequences.

Using an uninitialized integer is not that terrible. You know, if you're trying to sum the numbers in an array and you forget to initialize your sum, you get a sum that's junky but it doesn't crash your program, right? It just uses that number as the starting value and sums from there.

It produces results you kind of figure out what happened, but it doesn't crash. Many times, right, the use of a pointer that hasn't been initialized will cause an immediate crash, or it could cause sort of some later consequence to crash, right, that would make it even more mystical to try to sort out what happened.

So it's partly because those bugs are – have pretty serious consequences and kind of sometimes difficult to understand consequences.

So we do dynamic allocation via new, so dynamic meaning that just we get to choose what we're allocating, when and where, at a runtime situation saying how much memory we want and when we want it, using new.

And then when we're done with it we do a manual deallocation via delete. If we forget to delete, we orphan memory. As I said, an important thing to be conscious of but not in the early stages something to take too heavily. And then accessing any deleted memory, right, it has unpredictable consequences, similarly to using uninitialized pointers, right, going back to addresses that have been marked freed is kind of like going back to your apartment after you don't have a lease on it anymore, right, like you put your stuff in there, it might get stolen, right? Now a god idea.

So now let me tell you a little bit about how pointers work with arrays. We have not talked much about the C++ built-in raw array, that I've encouraged you to use vector where you would have used array because a vector is actually just much easier to deal with and it has a lot of convenience and safety features built into it.

But as part of kind of understanding how the basics of the language work, it's worth knowing that pointers and arrays have a little relationship that – in how they're expressed in C++.

So this piece of code that I have here, right, declares an int pointer called ARR. And it immediately initializes it to a new integer array in the heap that has 10 slots. I didn't quite draw 10, but that's a good idea of it.

So the new operator, in addition to taking a single type – making a single string or an int or a double out in the heap, also has an alternate form where you add the square brackets on it and then you say, inside the square brackets, how large an array of those things you would like.

So this is creating an array of integers 10 numbers long out in the heap, which array is going to point to. Having done that, right, array now has slots zero through nine available to it in which it can write and do things. And so the next thing it does is go through and

use the bracket notation that's standard for arrays to access in turn elements zero through nine index and assign them each their index number. It's missing two numbers that I couldn’t fit there.

And so that is the syntax in C++ for creating a new dynamic array out in the heap, accessing it either to read or to write to those contents. It looks just like the vector there. And then when I'm done with it, because it came out of the heap it was dynamically allocated manually by me, it's my job to manually delete it.

And there's a slight variation on delete that matches the use of the new int bracket, the delete bracket. So if I made an array of things out there, then I need to use the delete array form of it, so the delete bracket as opposed to the standard delete.

If I use just the standard delete, it's like it thinks there's only one integer at the end of this, and it only kind of tracks that space. So the delete bracket says there's whole arrays worth of integers, make sure all of them get reclaimed.

Now again, like most memory errors, right, if you do the wrong thing it won't necessarily have really clear – it's not like it provides an error message and stops and says you used the wrong form of delete, it's more likely to actually kind of just misunderstand your intentions and kind of blunder on, producing a little bit unpredictable results from there.

So raw arrays in general are trouble. We are definitely gonna use them a little bit later, in fact to build things like vector and stack and queue, right, we need to actually have this facility, right. The [inaudible] facility is what those tings are layered on top of. But as a client, I'm going to say that traditionally, right, that things that you would use an array for, it's just much better practice to use a vector for.

So I'll mention sort of why it is that, you know, arrays are trouble, right? Well, they're mainly allocated and deallocated, so that means you have to make the new calls, you have to make the delete calls. So forgetting one or both of those, right, has pretty serious consequences. Arrays don't know their length. That once I have created this array and it points to these 10 members, there is no mechanism to ask the array, you know, array dot length or tell me your length – it doesn’t know.

Internally, it may or may not have some housekeeping that's tracking it, but it's not exposed to you as a client. So it will be your job, if you are creating a manual array, to also be tracking its size. And so everywhere you were trying to use that array you'd also need to know well, here's where the address of that array, its location in the heap, and here's how many members go. You'll have to kind of just track those two things around and pass them in tandem.

There is no balance checking. If I access the 12th member of this, it actually just uses a very simple calculation, which says well, here's where the array starts in memory. All arrays are laid out contiguously. If you ask me to find the 12th one, then I just kind of

walk down to where the 12th spot should be, even if it's not really mine to own, and I will read or write to that location as you ask me.

I can ask for the 100th member of the 1,000th member, the 6 millionth member, and all the calculations work off of well, here's where the array starts. Where would the 6 millionth member be if it were present? That piece of memory, if it's not allocated to this array and it's not really part of its proper bounds will receive no error checking, no nice out-of-bounds message.

It will just kind of do the calculation to figure out what place in memory would be referred to and read or write to those contents, causing kind of unpredictable, weird behaviors, right, that would be hard to figure out.

You can't easily change its size once it's allocated, so if I've done a new int bracket 10, I have a 10-member array. If I later want to put 20 things in there, then there is no way to take this piece of memory and stretch it to say, well, ad some space in the back. What I will need to do is allocate an array that's twice as long, copy over all the numbers I already have, and then, you know, delete the old one and update my pointer to point here.

So the only mechanism for changing the size of an array, once it's been allocated is to really go through this process of creating a new array of the size you want, copying over what you wanted to retain, deleting the old one, updating your pointer, which, as you can imagine, gets to be just a bit tedious.

So you can certainly use the raw array to do things, and we will definitely have to build things on it later, but once you have vector around, it provides so many of these things, right, in such a nicer form that dealing with the raw array appears too troublesome and too little value, right, to do so.

So vector does use array behind the scenes, but it hides these issues, right. It does track its size, it does do balance checking, it does grow and shrink on demand, exactly by doing the kind of things that I said were kind of a little bit tedious to do, it does on your behalf. So you've just sort of seen the picture. Your hand's up.

Student: You said that the type of pointer determines what it's pointing to. So why doesn't an array have, like, an [inaudible]?

Instructor (Julie Zelenski):That's a great question. So it turns out that there is actually – maybe there's one example where I was being a little bit disingenuous. It turns out a pointer to an integer and a pointer to a sequence of integers are considered the same in C++. So it actually doesn't distinguish the idea that you gave it an address and you're telling it what kind of data's at that address.

You are not telling it how much of those are there, how many, right? You're saying there's at least an integer there; there could be a whole sequence. So in fact it considers a pointer to an integer and a pointer to an array of integers to be type compatible.

There are certain quirkinesses about that, but you can sort of see what – at least from a type system point of view, that actually kind of does make sense, right? They both are saying that well, we have these two addresses. What's at the other end of them? They're our integers. And how many are there? It turns out we're not – we don't track for you, so the fact that there's 10, the fact that there's one, is up to you to know.

Way over here.

Student: Why don't you have to dereference array when you assign it?

Instructor (Julie Zelenski):So that is also an excellent question. It turns out there's a kind of a dereference sort of hidden in the bracket operator that the bracket operator has, behind the scenes, is actually doing a calculation that says take this address array, add to it sort of I slots' worth, and then dereference that.

So in fact there's kind of a star hidden in there, and it's just part of what the bracket operator does on an array. So you are correct, there obviously is a dereference happening in there, and it is sort of hidden in the brackets themselves is all I can say.

Student: So [inaudible] when you just declare an array [inaudible].

Instructor (Julie Zelenski):Yeah, so if you just make a – yeah, so all that – it's basically the same deal, it's just where did the memory come from, right? So if you say int bracket 10, it's on the stack. If you say new in 10 it got [inaudible] heap. So it's just a matter of does the memory live over here and it was kind of automatically allocated or deallocated by entering and exiting, or is it explicitly allocated with new and delete over in the Heap?

The heap tends to give you that flexibility of resizing it and moving it and stuff like that in a way that the stack doesn't, so that's kind of the preferred place to usually get an array from.

So mostly I'm telling you this because we are going to come back to that one, but I just – it feels like it kind of fits with kind of getting a picture about how memory, right – C++ is very much exposing to you the sort of low-level where it puts memory and how it accesses things, right, that pointers are giving you that direct manipulation for.

One of the more common uses of pointers is this idea of sharing. So if I were designing the Access database and I have this record for a student with their first name and their last name and their address and their phone number and probably a bunch more data – their student ID number and maybe their transcript so far.

All these things that model the information track for a particular student in there, I also have a bunch of courses – CS106b, Math 51, you know, Physics 41. And each of those courses, right, has a list of the enrolled students in it.

There is a student record that represents Michelle, right? And there's Michelle's record in the system. It could be that every class that Michelle isn't enrolled in copies her whole structure into it, so if I had declared this vector here as holding student T without the star, it would hold real student T structures.

Then every class Michelle was enrolled in would have a copy of her structure. I would say add to the students in this class Michelle's record, add Michelle's record to this class. Well, each of those, right, would make a full copy – her first name, her last name, her address, her phone number, you know, however big this record is, right, would be duplicated throughout the system in every class she was enrolled in.

That means that first of all, there's a lot of space that's being taken, a lot of redundancy in that system. It also means that we have a little bit of an update problem. If Michelle gets a new phone number and we'd like it to change across the system-wide, then I have to find every place where her record has been duplicated and updated to get the update to kind of flow through the whole system. So finding every course she was enrolled in and updating it in every place.

If instead there really is just one copy of Michelle's student record – I allocated it in the heap, I set it up, I put things into it, and then every course that she enrolls in I store the pointer to that same record, then there really is only one copy of the student T structure per student, but multiple pointers to it in different places.

When I need to update her phone number, there really is only one copy of her phone number. If I change the phone number in that one record, all the pointers are pointing at that one record, and all the updates effectively happened automatically, just changing in one place only.

So this is one of the most common needs for pointers, is just any kind of data structure where there needs to be some kind of sharing, where you have information, right, that's being tracked in different ways. You want to be able to look at your iTunes library by genre, you want to be able to look at it by album. You want to look at it by artist. That there really is only one record for each song in the database that's being managed there, but there's a lot of different views on that data, and those views could each use pointers to refer to the one copy, and then make for sharing of the data without that redundancy, and then this easy update of only being one place to do it.

So without pointers, right, you're kind of stuck, right? If you don't know about pointers, right, you end up kind of duplicating this or designing some more funky system where maybe you have a number here that tells you where to go look up them in some other place, you know. The pointer gives you a really clear means of just modeling there is a sharing relationship.

That's pretty cool. So what else can pointers be good for? Ah. Pointers are good for something. Now here's the thing you wanted to do. You wanted to take pointers, which inherently are a little bit confusing, and then take recursion, which we also know is pretty

confusing, and then put them together and make something that will make your head completely explode.

We're talking about recursive data. For some people, this is actually the thing that actually does make recursion suddenly make sense, though, is to see it – to see recursion echoed in a structural way as opposed to a code way.

So when I talk about recursion in the form of looking at a structure that itself kind of has embedded within it a smaller version. So I brought with you my physical embodiment of recursive data, which is my lovely Matryoshka doll. Ikea, $15. If you open it up, what's inside? Whoa, check that out – it's another Matryoshka doll. You open her up – hey, it's the – she's facing the other way. She was shy.

You open her up, there's gonna be something good, you can feel it. And then – little, tiny – this is the base case. She sits there. What you got there is like a doll which herself is a doll wrapped around another Matryoshka doll, which has kind of this self-referential kind of inside of it, right? There is another doll like the one you saw before, but a little bit smaller, a little bit tinier. Then at some point you get the one that actually is just so small that we're just done.

And so there are things like this in the real world, things like onions. You know, you peel off the outer layer of an onion, there's really kind of a smaller onion underneath, and eventually you get to that kind of onion core, that base case. We're going to look at this in terms of a structure that contains a pointer to the same structure we started with as a way of modeling something recursively.

Think about managing an address book. I've got a name and address and a phone for each of the people I wanna manage. I could certainly put these entries together in a vector and have kind of a vector of all the people in my address book. I'm going to choose a different way to organize this by virtue of making this structure have an additional field, a pointer to another entry.

So I've got three fields that represent, you know, Jason's entry in my address book, and then he has a pointer to more entries, or at least an entry, let's say – the entry that follows this one, which itself can be the name and address and phone number of Joel, let's say, with a pointer to another entry, and so on.

So each entry points to another entry. That's what makes for a recursive structure. It's kind of funny – I'm not even really done defining what the entry structure is, and I'm already referring to it. It's that feeling of kind of like the way a recursive function is written, where it makes a call back to itself before you're done completing your whole thought, saying that there is a pointer to another one of these.

And so a link list is a pointer to an element which itself has embedded within it a pointer to another element, and so on. And if I use that Null as my terminating sentinel at the end

to say that, well, Monte has no one following him, then I have this chain that I can follow from the front to the back to visit all the entries in my list.

Okay, this guy is called a link list. So it's a list because there's a, you know, collection of things that are there. They are linked together by virtue of pointers, and they have a very recursive formulation, if you think about it, right, that any individual entry is one entry in the address book followed by another list.

So the whole list has a thousands entries in it, but in fact if you kind of break it down recursively, you can say well, there's an entry in the front and it's followed by a smaller, shorter list – one shorter, in this case, 999 entries. And similarly, I can do that at every stage, which is to take each element and think about it as well, it's an element, but it points to a smaller link list behind it.

So I'm going to actually just do this code in the compiler. But it's all in the handout, as well as on the slides that I'll post. Just talk a little bit about doing some stuff with link lists.

So the first thing I'm gonna do is I have the entry, the name of the phone number with the pointer, and I wrote a little print entry that printed a single one. So we're going to see a lot of pointers in here, we're going to refer to all these elements by their pointers, and then we're going to allocate them out of the heap, kind of on demand, and wire them together by using the pointers to get our flexibility here.

The one routine that actually I'm gonna – I have pre-written here is one that will create a new entry in the heap based on information typed by the user at the console. So it prompts them to enter something. If they enter a return that's our clue that they have no more names. And otherwise, we make a new entry in the heap.

So here's our call to make a new entry [inaudible] out there, store the pointer to it here, and then this access – I should mention this syntax is a little bit – should be – may feel a little bit new to you – is that new one dot name, that what I'm trying to do here is take new one, dereference it, and write to the name field.

And so the syntax that kind of comes to mind here looks like this: take the new one pointer, dereference it to get to the entry structure, and then dot to get the name field out of it. And that is actually a sensible sort of construction.

The problem with it, if I leave it as-is, is that the precedence of these two operators is not quite what you'd expect. That actually, the dot has a higher precedence than the star, and so in fact the way the complier interprets this is as first take new one, act like it's some sort of structure that has a name field in it, and then retrieve that name field which you suspect to be some kind of pointer, which you can then dereference.

It's doing it exactly backwards. If we wanted to use that syntax as-is, we'd actually have to introduce these parens that said please first dereference new one. It is a pointer to an

entry. Get the entry structure that’s referred to on the other end and then dot name to get the field name out of it. And that would work just fine. Just because that's a little bit awkward to write every time, there's actually an alternate operator in C++, the arrow operator, that combines the star and the dot behind it to go reach out there.

So this says following the new one's pointer, reach into the structure at the other end and access the name field. So I assign the name, I assign the phone, and then I set the next field to Null as just a good practice to say well, I've just created a little entry all by itself. Rather than leave that pointer initialized, let's make sure we set its Null so we know it's just a cell by itself and it's not connected to anything yet.

Okay. Let me just write a little code to test that. Um, I'll just call it N and then I'll say get new entry. And then I'm going to call print entry on it, so I'm just gonna do that to see that I can allocate an entry and then do something with it.

I can say it's Julie and my phone number is – and then it prints out my name and my phone number. Okay. Try it again, now I can say Jason, Jason's phone number is – oh, Jason has lots of digits in his phone number, okay. So we've got a little bit of stuff going here.

Now what I'm going to do is I'm going to build a link list. I'm gonna use this get new entry function to give me individual cells, and then I'm gonna wire them together and build a link list out of it.

So I'm gonna start by having my list be empty so the standard sentinel value, Null, is used when you – to indicate that you have no more cells or you’ve started with a totally empty list. I'm going to keep asking for new cells until they don't have any more, and so get new entry returns Null if they have indicated they have no more.

So I'll just go ahead and break out of the loop when that happens. Otherwise, right, I have a new entry that they just gave me, and I'm gonna wire it into my list.

Okay, so let's think a little bit about how the pointers are gonna work on this. So here's my list pointer, and it points to a cell, which points to a cell, you know, which points to a cell and so on, that there's information here, you know – this is Jason, this is Joel, this is Brittany, okay.

I get a new cell, it is Jake. Back from my get new entry. And I wanna attach Jake into my list to join it with the ones I have. So I've got an existing list and a new cell to add to it.

There are two obvious places to add the cell. Typically, when you have a list, it seems like probably putting it at one of the ends, given that I'm not trying to maintain any sorting order at this time, is gonna be the easiest place to put it.

So tell me, think about this. Should I may Jake's first or last? Is there any reason to prefer one to the other? Is there any advantage to going with one way versus the other one?

Why not? Yeah, why not first? What's good about first?

Student: Because when you're making a new entry you can probably more easily decide what the next pointer's going to point to as opposed to going all the way back through the list and going to Brittany and assigning that pointer to something.

Instructor (Julie Zelenski):Yeah, that is exactly true. So there's something a little bit quirky about link list in this respect, which is because it's a chain, right, where you have a pointer to the front and then subsequent pointers get your way down at the end [inaudible] it's very easy to get to the front of the list, but it's actually a little more work to get to the end.

If I wanted to, if I had a thousand entries in my address book and I wanted to add each entry to the end, then I'd have to kind of work my way down to the end to kind of do that work.

So it turns out it's actually just a little bit easier – requires less housekeeping, less work to [inaudible] if I just – I already have a pointer to the front. Why don't I just go ahead and put Jake on the very front. So prepend Jake.

So the processes for this are going to be I have Jake in here. Typically when I am putting a new cell into a link list, there's going to be two pointers that need to be adjusted to splice it in. The new cell needs to have its outgoing pointer, so Jake needs to be attached such that if you got to Jake, it would lead away to some other cells.

In this case, I'm gonna have Jake point to what previously was the front most cell. And then I need to wire the pointer into that cell, so, you know, Jake has to kind of exist in context, he has to have something coming in and something going out. The one that's gonna come in is gonna be the one that previously was the front most – the list head or the front note of the list is now gonna point to Jake.

So the two assignments I need to make to get Jake into there is assign Jake's next field to what was previously the front most cell, and then make the front most cell point to this new one that we just got back. So we wired out, we wired in, and now Jake is the new leader of the list.

Okay, take a look at that. Like a little bit of code, but doing something pretty important. This is Jake's next field getting sent to what was the previous front most cell, and then updating the front most cell to point to Jake so that Jake will become the new leader.

So if you think of it, you know, on the very first iteration, it's always good to make sure those [inaudible] cases are gonna work fine. If at the beginning [inaudible] there's no cells in the list so I start with an empty list, I get a new cell from here and then I set that new cell's next field to be the list that basically says set its next field to be null, okay.

That's fine, there are no existing cells to attach. And now make the head cell to the list point to this new cell. So that made us a singleton list that has one node in it whose next field is Null, and then on a subsequent call – so we come back in to put the next node in, we'll make a new one. Its next field will point to what was previously our singleton cell, and then we update the front most pointer to point to the new cell.

So it should prepend each one onto the list. So if I type the names in A, B, C that when I go to print them back out I should probably get them in the other order.

So let's take a look at this. Let's write build list here, and call it. And then I'm – right now it's just gonna print entry, which is going to print the very first one, and then I'm gonna change that to print all of them. So if I put Julie and some phone numbers, I put Jason and some phone numbers, I put Marcia and some phone numbers, and then I say I'm done, then Marcia was the first one in the list so when I said print entry, I printed just the first one right now because she was the last one I put on there.

It's operating a little bit like a stack – last in, first out. Then it put them on the front, so what was ever the last one entered is the one that is the leader of the list at this time.

Okay. Well, I don't have time to show you much more today, so we're gonna have to do some more of this on Friday. But we'll come back and we'll look at this to do a little bit more work and start thinking about how recursion is an interesting partner with this kind of data structure.

[End of Audio]

Duration: 51 minutes