Week 6 Module 14 Pointers and Dynamic Storage

Introduction 1.2

This module we're going to look at C++'s pointers and how we can use them to expand the capabilities of what we do in terms of programming. And using heap dynamic variables using heap dynamic arrays. We're going to create some very useful code here and this code is going to be the basis for a lot of what we're doing in the future for both future programming and for data structures. It also leads into discussions on operating systems so this is a rather important topic that's a gateway to a lot of other things in the rest of the semester.

Pointers – Why? 1.3

So the first question is why do we need pointers at all? And a pointer is a way that we can store a reference to an object that's either created on the heap or created on the stack. Now the stack is what we've been using to create all of our variables since we started programming. When we say something like index that's a variable that's created on the stack and we're going to talk a lot about what the heap is and how we can use it a little bit later on. But for right now there's something else that we can create and that is a heap dynamic variable. When we work with pointers we can have a pointer either point to a stack variable or a heap dynamic variable. And we can use the pointer as a way to store a large amount of information that's not associated with the function. One of the problems is when we start a function all of stack dynamic variables are allocated. And when that function ends all those stack dynamic variables are deallocated. But what if we wanted to have a variable that existed without having a function being called. We want the variable to continue to exist even after the function ends. We can create those variables on the heap but the heap doesn't have a name there's no way to name a variable that's on the heap. And the only way to do this the only way to have a reference to it is to have a pointer to that variable that's on the heap. So when we get into heap dynamic variables pointers are going to be really really critical. Also we can use a pointer to link two objects together. So the usual scenario that I like to use is imagine that we have some sort of object like a person object and we'll talk a lot about object orientation in the next module. But for now you understand that we can have some data structure that represents a person. Well what happens if one person gets married to another person? This means that we have to have an association relationship between two person objects and in order to do that we have to have a pointer from one person to another person. I would probably call that pointer a spouse pointer.

What It Looks Like 1.4

In C++ we have to specify the data type. C++ is very type specific so we always have to specify the data type. And a pointer is going to be specified as pointing to a particular data type. So a pointer as we're going to see later on is really just a number that we store in main memory to represent where this pointer is pointing to. It's actually just a memory address but the problem with this is that all memory addresses are the same size. Unfortunately C++ doesn't let us take an integer pointer and make it point to a double you can't do that. You can't make an integer pointer point to anything other than an integer. So if we took an integer variable and we found out its address which we'll figure out how to do later on we could take that which is just simply a number it's an address and we could store it in a pointer. Now that has the effect of making the pointer point to the integer and that's exactly how we do this. The stored number represents the memory address of the item that we're being that's being pointed to. We can create pointers using a very simple construct it's just int star ptr. In the same way that we can create integers like int x we can create integer pointers by just saying int star and then giving it a variable name.

At that point on construction the pointer doesn't point to anything reasonable it's pointing to garbage in the same way that if we said int x we don't know the value of x. If we say int star ptr we don't know the value of ptr. It's pointing at garbage.

Getting Pointers to Point 1.5

So we'd like to use these pointers to point to something and we can do that but we need to have some address that we make it point to. In another words we need to find an address of a variable and take that address and store it in the pointer and we can do that by using what's called the address of operator. It's a new operator that we haven't seen before but if we place it in front of variable that we have existing we're going to get back the address of that variable. Now the data type for the address of an integer the data type for that would be a pointer to an integer which is perfect because now we know how to create pointers. So we have a little bit of code here which is just simply creates a variable called x. Sets its value equal to one hundred and then we have a pointer and the pointer is set to point to x. Now that allows us to work with the x value but not use the variable name x which we're going to see later on is very important. We can use pointer ptr and access x as if we're accessing it through the pointer which is exactly what we're doing. But we're going to need to get the address of x first and we do that using the ampersand or what's called the address of operator.

Accessing Data From a Pointer 1.6

Of course it would be pretty useless if all they could do was point to something. Really we want to be able to access the data that the pointer is pointing to and we can do that thankfully we have another operator known as the dereference operator and that's just the star. It's a unary star operator we place it in front of the pointer and what that means is we're going to follow the path that the pointer is pointing to. If you sort of imagine that the pointer is a signpost if you will on a highway. What we want to do is follow that sign to get to a destination and we can do that using the star operator. So here I have a little bit of code which I'm going to show you creates a variable x sets its value equal to one hundred makes a point makes the pointer point to x and that's the same as in the previous section. But now we're going to go ahead and print out the value that the pointer is pointing to. Now this is not printing out the address of the pointer. If we simply said cout arrow arrow or output operator or whatever we're going to call it. Cout ptr we would get the address that ptr is pointing to. And the easiest way to think about this is if we printed out x we would get the value that x was pointed sorry the value in the x variable. If we printed out ptr we get the value in ptr but unfortunately that value is a pointer value which is just an address. If we dereference ptr as we have here in the code we are going to see that the value that's being printed out is actually the value that's in x. Because what we did was follow the signpost that ptr is pointing to and that took us to the variable x and now we're printing out the value in the variable x. And of course this falls on both sides both an L value on an R value we could talk about this being printed out or we can do an assignment statement where we actually change that value. That last line of code on your screen is actually going to change the value of x as if it were to say x equals twenty. And now we're seeing star ptr are equals twenty. It's the same thing because ptr is pointing to x.

What if a pointer doesn't point to anything? 1.7

So when we first create a pointer just like when we first create any variable. We don't know the value that's in it. And that's a bad thing if we're talking about an integer. But it's a really bad thing if we're talking about a pointer because when we create a pointer it has some garbage value it has something in it it's not zero it's not unknown value. It's something and we don't know what that something is so effectively the pointer is now pointing somewhere where we didn't decide where it's pointing. And if we use that we have the potential for crashing our program or doing horrible horrible things inside the

computer. So we don't want to do that. So whenever we create a pointer or whatever we have a pointer that's not pointing at something valid we should make that pointer point to a special place. And that special place is known as NULL or null ptr. It's capital N U L L or null ptr all lowercase. But it's really important that we always make sure that the pointer is either pointing at somewhere valid like a variable or somewhere on the heap or it's pointing at NULL or null ptr. If it's ever pointing somewhere else then we're never going to be able to check to see if it is pointing at something valid because some arbitrary location in main memory will look exactly like a variable on the heap or a variable in the stack. There will be no way to differentiate between something that's valid and something that's invalid. So we use null ptr as a safety mechanism. Whenever we have a pointer and it's not pointing at somewhere that we know we make it point to NULL or null ptr.

Defining Multiple Pointers 1.8

So there's one little problem with C++ that's a little bit of almost like an eccentricity of C++ it is a little strange. If we define multiple pointers on the same line the star doesn't associate with the data type it actually associates with the variable name. And I know this makes absolutely no sense but if you take a look at this sample code you'll see that here we're creating three pointers and one x one integer. So the three pointers need to have their stars immediately preceding the variable name and this actually doesn't even matter if the star is before the space immediately adjacent to the int or if it's to the right and immediately adjacent to the name of the variable. But it's important to recognize that the star does not make the line create all pointers it makes only that variable a pointer. So in this case we have ptr one ptr two and ptr three are pointers to integers. But x is just an integer on its own.

Let's Get Dynamic 1.9

So let's talk about dynamic memory. We've already talked about pointers and having them assigned to static variables or stack dynamic variables. But now we're going to talk about what pointers are really useful for and that is creating heap dynamic memory. We're allocating heap dynamic memory and having a pointer point to that heap dynamic memory so that we can use it. Now heap dynamic memory when we create it it has no names. So we have to use pointers in order to access that heap dynamic information. And there's no way to find anything on the heap once it's created and allocated you'll be given a pointer to that memory location and then you have only that pointer to use to access it. If you ever lose track of that pointer you've lost track of that memory. And that's a bad thing it's called a memory leak. So pointers can point to heap dynamic memory heap dynamic memory is allocated whenever you ask for it and it remains allocated until you destroy it. Which means that it can survive a function call what we can do is create this heap dynamic memory or allocate this heap dynamic memory in function A. And then function A can end and we go on to do function B and heap dynamic memory is still allocated. What we can do is pass a pointer between function A and and its parent function and then back to function B. And the variable still remains allocated in that same memory location and the only thing we need to do is keep the memory location keep the pointer. Now the unfortunate fact here is that while there's a lot of capability in heap dynamic memory there's also a lot of danger. If you don't deallocate your heap dynamic memory then you have what's known as a memory leak. What this means is that for every variable that you allocate on the heap you must remember to destroy that variable. It is not destroyed automatically and this is a huge downside to C++ and languages does that have support for heap dynamic memory. There are newer languages which prohibit the use of of certain features like heap dynamic memory. And there are other programming languages that do so in an more automated way like newer languages like Java will allow you to create heap dynamic memory but it keeps track of all of it and destroys it on its own time when you're not using it anymore. So C++ leaves a lot in your hands you have to do a lot of the work and it's very useful but you have to be very very careful. Memory

leaks themselves won't cause your program to crash at least not immediately but over time a small amount of memory that's wasted by your program will add up. And if your program runs continuously for months and months or even weeks and weeks. It's quite possible that you're going to run out of memory eventually and then your program will crash. But it's insidious it's going to take a really long time before that happens.

Well, That's New 1.10

So how do we create new memory on the heap? How do we create these heap dynamic variables? Well there's a function that we can call it's actually an operator technically in C++ and it's called new. All we do is say new and then the data type that we want to be created and we get back a pointer to that new variable that's created on the heap. It doesn't have a name it will never have a name. So the only way that we can refer to it would be via a pointer. Again once memory is allocated on the heap it's not deallocated until you deallocate it or the program ends. If the program ends all the memory for that program is deallocated. If you ever lose track of a pointer of a pointer to a heap dynamic variable or if you ever lose track of the dynamic variable it immediately becomes garbage on the heap and you can't use it anymore. And that's your memory leak. But for creating a new variable the only thing they have to do is call new and tell it the data type that you want in terms of integer float double or any of the other data types. And it is going to give you back a pointer to that data type so you just need to store that pointer. And then you have all the access that you need by the dereference operator you can make copies of the pointer that doesn't make copies of the data necessarily. And you can use the pointer to access the variable that's created on the heap. Remember that it doesn't get destroyed until you specifically say it should be.

For every new, there must be delete 1.11

For every new there has to be a delete. If memory is allocated it's going to have to be destroyed and once you destroy it you can only destroy it one time. So you can destroy it once and only once. Now here's the double edge sword that we have the problem with. We have to destroy it once and only once if we don't destroy it that's a memory leak and if we try to destroy it twice that's double delete. Memory leaks we know they're going to take a long time to show up as problems. The program will continue to run. It'll just take up more and more and more memory eventually running out of memory and then it'll crash. But double delete is much more critical. As soon as you do a double delete the first time you delete the object the first time you delete the integer or whatever it is you're doing. The memory is returned to the operating system and the operating system says it can be reallocated to someone else but that second double delete tells the operating system that it can take back a memory that it already owns. And the operating system has a real problem with this it says but I already own that memory something's wrong with your program and your program will immediately crash. Your program will end and on a Windows machine for example you'll get a pop up box that says Microsoft is sorry but your program has performed and illegal operation and that's the end of your program. Now we can delete a memory location or we can delete a heap dynamic variable I should be specific and say by just calling the delete operator. So we say delete ptr that does not do anything to the pointer itself. Delete ptr doesn't change the pointer all it does is return the memory that's being referenced by that pointer returns it back to the operating system. The pointer still points to the same location it's just you don't own that location anymore. So again it's important to then take the pointer and make it point to NULL or null ptr right after you do the deletion so you don't accidentally double delete. If you accidentally do a delete on NULL or delete on null ptr It has absolutely no effect. There's no crashing there's no problem. The operating system and C++ just simply says that has nothing to do so I'm done. And deleting null ptr is perfectly safe. So there are certain circumstances that we're going to see over the course of the rest of

the semester that you'll see where we delete null ptr and that's perfectly ok. What's not ok is deleting the same pointer twice.

What About Arrays 1.12

So now that we understand that we can create variables on the heap. There's not really a whole lot of benefit in creating a single variable on the heap but what there is a huge benefit is creating heap dynamic array. If you remember back to our discussion on arrays we said that the size of a stack dynamic array would have to be known at compile time and has to be static. So when we created our arrays back in the previous modules you created them with a known size like twenty five or fifty it was some constant values literal value. But what if we didn't know what if we didn't know how many elements we needed to create until we got to the point that it was time to actually create them. So for example we might ask the user how many things are you going to tell us how many objects do you need to store in this array? Or we might pretest how many objects we're going to put into an array. We might read in a file and see how many integers there are before we're even loading them into the array. And if we do that then we have to create a heap dynamic array because a stack dynamic array does not allow us to create it with a dynamic size. It has to be a static size. So working with a heap dynamic array means that we are going to have this since it's created on the heap. It means we're going to have to have a pointer to point to that array. Since we're working with pointers another big benefit is that if later on we decide that now we don't have enough space in this array and we need to make it bigger that's not really possible by the nature of the way that arrays work. But what we can do because this is just a pointer is we can temporarily create a new array that's slightly larger. Copy over our values into that new array we can delete our old array and make our old pointer point to our new array. And that effectively resizes our array because the pointer will still be the same pointer. But now the size of the array will be significantly larger and that's an important factor we're going to see this later on. We're going to create objects which store information and they're going to expand on how much information is stored based on the inputs. But for now what we want to see is just a simple format for asking the user how big the array should be remembering that number and then creating an array. And we can create arrays on the heap just by specifying the data type we call new again we specify the data type and then we use the array operator the square brackets operator and tell it the size. But now the big difference is the size doesn't have to be static the size can be decided at run time. When we create the array of course we have a new operation we're going to have to have a delete operation and the delete operation is a little bit different just slightly different. Because when we are working with arrays we're going to have to tell C++ that we don't just want to delete one element we want to delete the array. And we do that with the square brackets operator after delete. So this would be delete square brackets array. Now don't get ahead of ourselves and start thinking that we can delete individual objects inside of an array. We really can't do that it's not really safe to do that. What we want to keep in mind is that if we're deleting a single object we use delete ptr. If we want to delete an array we use delete square brackets arr.

What can we do with heap-dynamic arrays? 1.13

So what can we do with heap dynamic arrays? Thankfully everything that we can do with normal arrays. All the normal arrays we've been working with are actually just pointers to stack dynamic arrays. So even when we created those old stack dynamic arrays that you did in previous modules you actually had a pointer to the base of a stack dynamic array. Well now we have the pointer to the base of a heap dynamic array and we can do everything that we could do with the old stack dynamic arrays. There's really no difference heap dynamic arrays work just the same way as stack dynamic arrays. We can still use the square brackets operators to access each of the individual elements. We also now if we wanted

to could use the star operator dereference operator to access just the first element. So we see that there's really no difference between a heap dynamic array and a stack dynamic array.

Pointer Arithmetic 1.14

One of the fun things that we can do with pointers is pointer arithmetic and this is especially useful if we have a pointer that's pointing at an array. C++ allows us to manipulate pointers using standard arithmetic operators like the plus sign and the minus sign and the plus plus and the minus minus sign. And what we can do is add on to the pointer. Now C++ understands enough about what the pointer is pointing to to make some interesting extrapolations. What it does is for example if we added zero to the pointer that would obviously move it no positions. But if the pointer is pointing at in an array then we can move the pointer up one element of the array by adding one to it and that's possibly very helpful. If you want to use pointer arithmetic to access the elements of the array you can make a temporary pointer and then progress through the array in a loop adding one to the pointer each time and that would take you through all the elements of the array. You don't have to worry about how large a single element is. If this for example is an integer each element is going to be four bytes. If it's a double each element is going to be eight bytes. But that's not something that you have to concern yourself with because with pointer arithmetic the pointer data type is known because we can only have an integer pointer pointing to an integer. So when we add one to that integer pointer it's going to move up one element which means either four bytes for an integer or eight bytes for a double. If we add five to it it moves of five elements. But the important thing to recognize now is if we change where the pointer is pointing to then the zero element or the square brackets zero element the first element in the array will now be different. So if we have an array with the number of elements and we add five to the beginning of the pointer to the pointer at the beginning. It's going to move up five elements which makes the zero element now what is actually the fifth element in the real array. So that has to be taken into consideration you should always remember where the beginning of the real array is. If you want to move up just one element of the time we have the plus plus operator and the minus minus operator and those can just move us forward or backwards an individual element of time. So something like ptr plus plus is a really useful tool for accessing the array.

A Real Example of a Growing Array 1.15

Hello everybody I wanted to show you a simple example of how we could create a dynamically resizing array or rather how we can dynamically resize an array. And I wanted to do that by starting out with just a simple array its size is 10 and it holds some stuff you can see the code on your screen. I hope we're looking at just an array that holds 10 things and we've gotten to the point where we'd like to add one more element. So we'd like to add that one more element into the array and unfortunately because the array is already full we don't know how it's possible that we could add any more elements into this array. So what we're going to do is write a very simple function which is going to resize the array and we're going to give it the array itself as well as the current size and the new size and we're of course going to have to go ahead and write that function. So it's not going to return anything and it's going to be called resize array and it's going to take the array as a pointer now we have to recognize that the array will change. So it's important that we pass that array by reference or rather that pointer by reference because in this case the array is going to have to be resized and the only way that we can resize an array is by destroying it and creating a new one in its place with the old data. So we have to take into account I'm going to call this current size we have to take into account that that array pointer will actually change which means it does have to be passed in by reference and we're going to take in new size. So we're going to create this function which takes this array the current size and the new size and the purpose of this function is to grow the array into a new size and we can talk about how to shrink

the array we can do that as well but that's not the core of what we're trying to do here. So I'm not going to focus on that so when we look at this the first thing we should recognize is that we're going to have to create a new array. So we're going to need a temporary pointer and that temporary pointer should point to an array of size new size so we're going to create on the heap a new array that is pointed to by temp so temp is going to point to an array of new size and then we've got to go through and copy over all the elements of the array into the new array. So one by one we're going to have to go through the old array and copy the elements and we can do that just by saying temp at I is equal to ARR at I. Okay so we've got now two arrays the original array ARR and new array pointed to by 10 and they effectively hold the same data and the only difference is that ARR is completely full and temp now has a little bit of extra space left. So temp has some free space which we can recognize as the new size of the array and then we're going to just take care of deleting the old array so now we've done a new operation we know that if we don't do a corresponding delete operation we're going to have a memory leak or garbage on the heap. So what we can do to take care of that is delete the array ARR which cleans up the memory space pointed to by arr now obviously that means everything in there is deleted but that's ok because we've made a copy of all that into temp. And then here's where it's really important that we pass this array in by reference we're going to make ARR equal to temp that's effectively resizing the whole array so right now those couple of lines of code there what we've done is created a new array temporary array of the new size that we want we copy over each of the individual elements and then we delete the original array and make the original array point to the temporary array. The effect of that is that we've resized the array now obviously this is a very involved operation and it's going to have to go through each element of the array in order to make the copy. So we don't just want to go up by a small amount what we'd like to do is say that the new size should be something like double the size so down here in main. I'm going to say that the new size should be the size times 2 and then I can say size equals new size because we've now updated the size of the array and we can put our new element into the additional position. So ARR at size plus plus is equal to one more and that has the effect of taking our original array which was only ten elements and now storing the same information in a new array that has 20 elements and then storing the position 10 which of course would be the eleventh element the element that we're trying to store. So we have the capability now not just to create a static sized array we have the capability to create an array that can grow and we can do that on the heap.

Someone's Done This Already 1.16

Now if you're thinking that what you saw was a lot of work. It was and somebody has already done it all for you. Somebody took all the operations that need to be done to expand and an array and keep it up to date and bundle that into what's known as a vector and the vector is a component of the standard template library which has a lot of components which we'll talk about throughout the semester but the vector inside the STL is an array that dynamically grows whenever we needed to. So the key functions inside the vector are the size function obviously which tells us how many elements we've put into the vector. So that's nice because we don't need to take care of how recording how many elements are inside the vector and we use the push back function to add elements into the vector. That's automatically resizing so we don't have to worry about how many things we can put into the vector. We just keep putting things into the vector. Now of course we're going to have to pound include vector and we'll have to create it when we created the notation for creating it is a little bit odd we have to tell it what data type we're going to be storing inside the vector so in this case we're restoring int. Now that doesn't mean that we could store other data types so if we get to the point where we want to store floats or doubles or chars or any other data type we can literally store anything in the vector whatever data type you want. So we create this vector called V and we're going to push back one hundred elements each of which is the value of the element position times one hundred just for the sake of

showing something. And then we're going to go through that vector and print out each of the individual elements. So you can see that we're still using the square brackets operator the same way that we were using on the arrays we can still use the square brackets operator on the vector class and now we have something that takes care of the size. So the vector class is one that we're going to really want to get familiar with because we're going to be using a lot throughout the semester. We're also should understand how it works internally and we will in a homework assignment or in a laboratory assignment. But the the vector is a really useful data storage class inside C++ and this is what we'd like to use going forward for storage of a lot of data we just use it whenever we don't know exactly how much data we're going to store but there's a lot of it.

But Wait… There's More! 1.17

Ok so we've seen the vectors and we understand how vectors work. There's one more thing that was added in C++ in 2011. Obviously C++ is dynamic it's still being updated there's a lot of work being done to it but this was added and it was actually taken more from Java and some other languages because C++ realized that they were missing this capability and it's called ranged for loop. The ranged for loop has a really strange syntax but that's because it was borrowed from other languages and what you see is I have the same code from as from the previous section and you can see that I've changed that last for loop now and this is what's called a ranged for loop because we're progressing over the entire range of the vector. So this is a very common occurrence we want to do something for each element inside the vector. So there's going to be some number of elements inside the vector in this case we have one hundred elements which we loaded up in a previous line and now we want to do a task for each element inside the vector. And in fact other languages call this a For Each loop but C++ is called the for loop so we specify what data type we're going to get out of the vector in this case it's an integer we specify a variable name for the individual element as we go through the loop in this case I am using I. And then we specify a colon and the name of the vector that we'd like to progress over. And from that point inside the for loop we have access to this variable I which is a different element of the vector each time we go through the loop. So in the first iteration through this loop I is equivalent to v square brackets zero and in the second iteration loop I is equivalent to V square bracket one and so on and so forth until we get to the end. But it saves us having to worry about which element we're specifically working on and it also saves us from having to check to see that we haven't gone off the end of the vector. So the format for this for loop while being a little bit strange actually can save us time and protect us from making a lot of mistakes.

Conclusion 1.18

So we covered a lot of material in this module we covered pointers and we covered how to make a pointer point to something how to get the address of a variable how to access information via a pointer. We also talked about heap dynamic variables and heap dynamic arrays. We talked about vectors and we talked about the range for loops we covered a lot of material and this does take quite a bit of time to acclimate to using heap dynamic variables and making sure that we don't have memory leaks and making sure that we don't have double deletes. But we'll get used to them over the course of the semester and you'll be experts by the end.