Let's talk about arrays. What is an array? Well, an array is going to be a block of memory, a block of memory where it's divided into individual units and where each of these units have the same kind of data structure, have the same kind of data. So we could have an array on the stack. So here we are creating an array of length 5 with the elements 10, 20, 30, 40, 50.

So when we create this array, what we are saying is we are creating, we are reserving a block of memory that is going to have five integers. So it's 32 bits times five and we are going to have a pointer that's going to point to it. So if you look at the array address, what we see is some kind of a hex number and that is where our memory block is. So the only thing our program knows is hey, there is a memory block that's starting at this address ED98 and there's going to be some integers, since it's an integer array there. So since this pointer is essentially just a 64 bit number, since I'm on a 64 bit operating system, we could look at array 0 and array 1.

The elements at those take their address and when we take their address, we end up with a memory address and assign it to an unsigned integer 64. So when we do that, I can see that, hey, we have the first element's address as a decimal or as a hex, second element address as a decimal or as a hex. So we could in theory just pick a number that is going to correspond to any kind of a memory address and then try to access it. So for example, over here I am starting from the first and I'm multiplying 4 times size of ints, so 4 times 32 and that gets me an integer, a 64 bit integer. And I'm done casting it to an integer pointer saying hey, I'm going to use it as a pointer to a memory address.

And then I'm able to get to the last elements. It has got the memory address of eda8 and it's got the last element value 50. So in theory I can do these operations. I could have added array size times 32 or 100 to kind of go to another memory address. Then I would probably end up with a out of bounds error or a memory error as part of it.

So if we kind of try to do some pointer arithmetic and we add array 5 to an array and then take dereference it, then we have out of bounds access when we try to Access it. Well, who knows what kind of a memory we'll get for an array that's on the stack that we have declared locally? We can use sizeof to get the number of bytes it is. So we see it's 20 bytes in this case. And if we have an array on the stack, we can also pass it.

So when we are passing an array to a function, what we are really passing is a pointer. What we are passing is the 64 digit number. So we get the 64 digit number, it gets copied, and now it's called MyArray. And if you try to use size of, what we end up is sizeof is 8 bytes. Why is it 8 bytes?

Well, it doesn't really know the size of the array. It's actually just looking at the size of an integer pointer. And an integer pointer is a 64 bit number, 64 bits is 8 bytes. So we are actually just looking at the size of the pointer. So we try not to use sizeof for arrays.

So when we create an array, we really do need to kind of keep track of the array size as a separate element.

Of course, we can create an array on the stack where the compiler takes care of getting rid of the memory, or we can create an array on the heap. If we create an array on the heap, it's going to persist beyond the function that created it. Whereas when you create something on the stack, as soon as the function finishes, that memory is released back to the operating system. So if you want to create an array on the heap, the way we'll do it is using the new command. And now we are saying, hey, we have a new integer and we are reserving five sizes.

And we are also assigning the initial values 10, 20, 30, 40, 50. And we have our array pointer and the integer pointer, well, it's just a pointer. So if you look at the this using size of, we get eight bytes, we don't get the size of the array, we get the size of the pointer. Because all we have is a pointer. All we have is a 64 digit 64 bit number.

And if we create something on the array, we have to delete it. It's our responsibility as the programmer to delete it. And when we are deleting an array, use delete with the square brackets. There is however, an extension of C, it's called variable length arrays and it is not part of official C, but It's part of C99. So some compilers will do it.

And what that allows you to do is to create an array that is the sizes dynamically defined. So the size we are getting from the user we don't know at compilation time. And then we can still create it on the stack as part of it. And once we have created it, of course we can, we know how big it is and we are able to use it. It's not usually recommended to have non constant size arrays created on the stack.

If you don't know the size of the array at compilation time, you should be creating a dynamic array. And of course we also have arrays that are part of standard templates library. So and this is much more for compatibility and standard template library has a lot more other data structures to be used as part of it. So if you wanted to use the standard template library, we would say, hey, we are creating an array of integers, it's going to be size five and then we can give it the values. And if you wanted to look at the memory address, we can use it using data as part of it.

And because this is part of the standard template library, we have the dot size operator to be able to look look at it. And there are some common operators that are part of the standard template library. So if we have a constant integer 5 and we create an array, we can of course use index based address.

And in addition to having an index based address where we use the index to kind of go through all of the elements, we can also have a range based address access. And a range based loop is better. If you're going to do something with all of the elements so you don't create an extra variable, you say hey, for all of the integers X that is part of the array we are going to print out in this case the element. So there's a couple of good standard template library functions that are worth knowing. One of them is called max elements.

Max element is going to go through the array and find the maximum elements and return a pointer to it. So it's going to return an integer pointer. We give it the array and when we say we give it the array, what we are really giving is the 64 bit number saying hey, this is where the memory starts and we give it the ending location saying hey, the memory blocks finishes at array +n. And using that notation we are able to get the maximum element. Since it's an integer pointer, we dereference it when we want to print it.

And similarly we have the minimum element as well. We also have other functions.

We also have other functions that can be useful. One of them is O and O is going to return true if all of the items in the array satisfy a condition, once again we give the beginning of the memory address the end of the memory address. And in this case I have a lambda function that returns true if the number is divisible by two. So I can look at if all of it is even. We can also have other functions such as any of and this one is going to return true if any of them is greater than 25.

So as soon as it finds something that's greater than 25, it just returns true as part of it. And we also have the opposite, which is none less than 10, and that's making sure that none of them are going to be less than 10. So these are useful functions. We can use them or we can easily use their equivalent by writing our own for loops. Another thing that comes very useful from standard template library is the sort function.

We are able to sort an array once again. We give the beginning of the memory address, end of the memory address, and sort will sort the arrays. And then once an array is sorted, we can do binary search on it. So we don't have to write our own binary search routine. There is already one in standard template library.

We give it the beginning of the memory array, memory block, end of the memory block and the number we are searching for, and then it's able to tell us whether we have found 20 or not. Similarly, we can also reverse an array and we can kind of check that the array that we have, oh, where's my output is in there.

And another useful function is being able to swap two numbers. So we can swap array at index zero and array at index four. And afterwards, well, we can kind of print, print them. Or when we are using swap, you can also use once again pointer arithmetic, saying array plus one referring to element at one and array plus two referring to element at two. So now we are not printing because there is no flushing of the buffer.

So once we print it we we'll see that array 0 and 4 got swapped and then 1 and 2 got swapped. But this was after being returned, reversed. So now we have 10, 30, 40, 20 and 50 as part of it. So this is the end of a mini lecture on arrays. Just kind of getting comfortable with arrays as a block of memory that has a 64 bit pointer, 64 bit number that is pointing at there.

And we'll build on these things.