Week 5 Module 9 Introduction to Static Arrays

Motivation 1.2

Hi there. I hope you're okay. Today we're going to talk about our first data structure in simplest loss. But before we start doing that, let's take a second look at one of our programs we've implemented in one of the previous models. If you want a more detailed review, you can just click on the link below. But I hope you recall that we calculated an average of a sequence of elements.

Actually, we wrote a program that computes the average of student grades. The program first asks the user to enter how many grades. How many students are in the class of the student, the user responded; let's say four. And then he was asked to enter the grades one after the other, so seventy one, eighty six, sixty nine, ninety four, and eventually the program just responded with the average of these grades, which is seventy nine point twenty five.

So, if you recall when we implemented this program we basically summed up all the numbers and divided it by the amount of the numbers, that's how we calculate the average. Let's take a closer look at our implementation.

Computing the Average Implementation 1.3

Okay. So we start by reading the number of students from the user, and then we ask the user to enter the grades. Then we do number of student iterations in order to first read each grade but then, we also want to accumulate the sum. So, for that we have our accumulating variable sum that is first initialized to zero before starting to iterate and then after each time we read a grade, we also add it to this accumulation variable, we add it into sum.

So, when this 'for' loop ends we have the total sum of the grades in our sum variable. So we can just divide sum by the number of students and get the average. After doing that, all we have left is just outputting this average back to the user.

Above The Average 1.4

Okay. So now let's try to make this program a bit more complicated: in addition to just calculating the average, let's also figure out which grades are above the average.So, the program would interact something like that. First it would ask the user to enter the number of students in the class, the student would say four. Again, the user would be asked to enter the grades. So seventy one, eighty six, sixty eight, ninety four, and then the program would not only say that the average is seventy nine point seven five. It would also say that the grades above the average are: eighty six and ninety four.

Seems very similar, but then if we think about it a bit more; let's see how we can implement this requirement. So, we would need to know what grades are above the average. We can do it really in a single pass over the elements, just as we have for calculating the average when we wanted to sum them up because only after figuring out the value of the average we can know which grades are above the average.

So it seems like we would need two passes over the data in order to find the grade that are above the average. So one solution could be let's ask the user to re-enter the grades after we calculated the average but that is not user friendly at all. And so, another solution could be maybe to store the grades, to store these datas in our memory, so we can pass over them more than once. It could be very impractical to store them in different variables because first, we don't know how many variables we would need and it would

be very difficult to manage these variables. For that we would need some kind of a data structure, which is a single entity that allows us to store a collection of elements in it and to access and modify these elements as we need. We would use the ray data structure, a static array basically, which can be thought of as a sequence of variables that are of the same type; in this case are integers. Let's see how it works.

Above the Average Implementation 1.5

Let's see how we can get it done. So let's have our main place for the variables, so let's have an integer X, a double Y. That's a sign X. with four, Y. with seven point three. And then our memory, our run time stack would look something like that. We have our X, let's say it's physically stored in the address one thousand, it's assigned to be four since it's four bytes long, then it would end at thousand and four. Then we would have Y; Y is double so it's eight bytes long, that would take it to ten twelve. Y is assigned to seven point three. Okay, so all of that we already know. We can create variables of different types, assign each one of the stored in the runtime spec, everything is as we use.

Now let's create our already, which is a sequential collection of variables. So if you do something like that; int arr five that's a single line of code that basically declares a sequence of five integers. So they are in the memory one after the other: one, two, three, four, five. And there are ten twelve, ten sixteen, ten twenty. They're all integers. So each of them is four bytes long: ten twenty eight, ten thirty two. So in a single line of code, we basically declared a sequence: a collection of five integers. And then let's say I want to write the value ten over here. So this array's basically zero based index; each one has a local address, a local position. And if I want to assign this thing here to be ten, I'll just have arr index two. I use a square brackets, equals ten. If I want to write a value fourteen here, I'll just go arr four. That's the index that's a position where I want to have the number fourteen in and put fourteen right over there. So as you can see, it's very easy to declare a sequence; to declare a multiple collection of variables in a single line and then we can also have our uniform naming mechanism using indexes to access the different position, the different cells or slots of this array.

Basic Properties of Arrays 2.1

Let's recap the three basic properties of arrays. So let's say we have an array, so the first thing is obviously that the elements are all stored one after the other in our memory. So, the elements are stored continuously in the memory. Second thing is that all of the elements are of the same type. For instance, if they're ints, they're all going to be four bytes long but they all must be of the same type; we can't have an array that one element is an integer, the other is a double and so on. So either all of the elements are integers, or all the elements are doubles, or all the elements are chars or whatever, but they all must be of the same type. And the last thing is that we access our array using a zero-based index system so they have their local addresses starting at zero, one, two, three, and so on.

Basic Array Properties 2.2

So, these three properties basically imply another very important property; let's take a look. For example if we have an array of six integers, in the memory we would have six continuous integers one after the other. Let's say it starts at address one thousand. If we want to assign let's say arr zero with four, that won't be an issue for the compiler since it knows that the array starts at address one thousand, then that's exactly the location where this four should be written.

Okay, so what happens if we want to assign arr two with let's say five. How would the compiler know where in our memory to write this five? So since it knows it starts at address one thousand, how could it

calculate; how could it figure out where arr two is? So, this is obviously index zero, one, two, three, four, and five: these are the indexes of the array. Arr two should be two jumps of four bytes after this one thousand. Actually the compiler can figure out a formula that the address of arr two is one thousand plus two times the size of each element in the array, basically one thousand plus eight; that is the address of arr two, that is where five should be written. More generally, we can say that instead of one thousand, that would be the address arr begins, plus two times instead of four, let's just say the size of each element.

More generally, let's say since we know that all the elements are continuously in the memory they're all of the same size and it's a zero based index system. We can figure out a formula that the address of, let's say arr I. Let's do a general formula here, would be this base address where the arr starts.
So the address arr begins, plus in this case I jumps of the size of each element, which is I times size of each element in arr. So the compiler can use this formula in order to figure out where each element is located in the memory.

Basic Array Properties Cont'd 2.3

This formula that the compiler uses makes very weird behavior. For example, if we do, let's say, arr five equals seven, that's regular behavior where the address of five would be, once again, one thousand plus five times four that would be one hundred and twenty, one thousand four, thousand and twelve, thousand sixteen and twenty and that's where the seven would be written in.

But let's say the programmer says arr eight equals, let's say, ten. So that's obviously a logic mistake because our arrays only of size six and we're trying to access arr eight, which is basically the ninth element in this array. There are no nine element, there are only six elements but then the compiler would just use this formula.So the address of error arr eight would be one thousand, as our base address, plus eight times four which takes it to a thousand and forty, and ten will be written over here. So the compiler doesn't care that we go out of the range of our array, it basically goes follows this formula and writes to the memory using this formula and it's our responsibility as programmers to keep ourself inside the boundaries of the array. Know that in address ten forty, there could be other variables it can be even memory that is not associated to our program; it could be associated to some other application that runs simultaneously to our program And it's very dangerous to write to locations that are not logically related to our program, so it's our responsibility to keep in bounds.

Another quite weird behavior we can say arr negative two equals, let's say, twenty. Again the compiler would follow the same formula: substituting negative two to be one thousand, the location is one thousand minus two times four, which is basically ninety nine two, that where the twenty will be written. So we kind of have the eight index and the negative two index, which are invalid indexes. But the compiler does allow it; it's not, it won't be in the air. So, again be safe when you use in indexes for the arrays. It's your responsibility to keep inbound of our range.

Syntactic Notes 2.4

Okay, so the arrays that we're talking about are actually called static arrays. There are also other kinds of arrays which are called dynamic arrays. We'll talk in more detail about them in one of our future models. But static arrays are arrays that are stored on our runtime stack, just like other variables. For example, if we have, let's say, main with an integer, an array and a double, since they're all stored on the runtime stack X. is stored first, after that comes our five element array, one, two, three, four, five, and right after that comes our Y.. So basically the size of this array must be known at compile time so the stack can be organized; the compiler could know where exactly this Y should come. That means that we have to supply the physical size, the number of bytes basically, that the array takes in our memory at compile time. We can say something like, let's have an integer X. and then in an array of integers and the double; we must say what's the physical size of this array.

Let's sum up a few syntactic notes regarding static arrays; so a few important things. So, the most important thing I want to say here that: the array's physical size must be known at compile time, basically means it must be a constant and it must be given at declaration. We have to say int arr six, which is a constant int literal: it is known at compile time, it is given a declaration. Or we can have, let's say, const iny X. equals seven and then create an array of size X.. Again, it's a constant and it is known at compile time. We cannot do, let's say, int arr without giving any size or we cannot have, let's say, an integer N equal seven and then create an array of size N, again it's not a constant; it's a variable, that's also not legal. So, when we create, when we declare this array we have to supply the size of the array as a constant at the declaration line. I have a few more syntactic notes so I'll show you later, but for now let's go to implement our 'Above Average' problem.

Above the Average 3.1

Okay. So now that we have the syntax of the arrays in C++, let's go back to the problem of calculating the grades that are above the average. So, as you recall, we have our class with four students, we have the grades, and we know that, not only, that the average is 79.75, we also know which grades come about the average. In order to implement that, we would obviously store the grades in an array, but if you recall we have to provide the physical size of the array at compile time. So, when we write the code, basically we have to know how many grades, or what the maximum amount of grades we're gonna have. Think it would be a very legitimate assumption to assume that a class won't exceed, let's say, sixty students. So, let's update a bit this problem. And the program first would ask the user to enter the number of students and the grades, saying not more than sixty. And then continue as we said before. Let's go implement it.

Above the Average Implementation 3.2

Okay. So let's start with implementing this program; so first, let's ask our user, cout 'please enter the number of students.' And we should also mention that we're expecting to get a number that is less or equal to sixty as we said we are assuming that the number of students is less than sixty so we can use a static array physically of sixty elements. So, let's say, no more than… you know instead of writing sixty hard coded, I think it would be better if we'll just defined constant variable. So const int max class size, I'll name it, and I'll set it to sixty and then I would just say, no more than, and I'll just take the max class size value. And that would be our announcement.

Okay. Now let's read the number of students; so let's create a variable num of students. And let's read whatever the user enters into this variable. So now we have to… Okay, so let's create an array of physical size of sixty so we can store the elements, the grades in it. So, int, let's call it grades list, and it's going to be the size of sixty. Okay so, now we have the number of students, the array and we should start reading the grades into this array. So for that we would obviously use some kind of a loop. Let's take a for loop for that, actually when we implemented the program of just calculating the average, we simultaneously both read the input and summed all the grades up in order to calculate the average. But now since we're storing the grades in a data structure, I think we can split this task into a few steps. So, first step would be reading the grades: so we'll probably have a form for that. Second step would be calculating the average, and then as we pass over the data one more time, will print grades above the average. So we're going to have three steps and not only like two; one for reading and calculating the average and then to print the ones that are above. We'll also split it to a reading step and the calculating average step that are separate.

Okay, so let's start with reading the grade. So, let's have like a variable for a current grade: so curr grade and each iteration we're going to read the current grade, and we'll also want to store it in our array. So for that, we would probably iterate over the indexes of the array; they basically said the location where we put this grade in. So we'll have an ind variable, initially ind would be set to zero, each iteration we would

increment or increase this ind. And we'll just put our current grade inside our grades list at the ind position. This is where our current grade would go to.

So, basically each iteration we read a grade and we store it in our grades list in the ind position. We start with index equals zero and each iteration we increase our index for the next element. So first time, firstly iteration, we insert our grade into grades list zero, second iteration grades list one, third iteration grades list two and so on. We'll keep on doing that while our index is inside the valid range for our array, which is no more than a number of students. So as long as index is less the number of students, we want to keep on going and reading. Let's try to trace only this piece of code that we have up to now so we'll make sure we get exactly how this works, why we stopped or controlled this loop exactly like that. Small details that, I think, by tracing we would get it with a better understanding.

Above the Average Implementation 3.3

Okay. So let's take a look at our run time stack; let's assume it starts at address one thousand, first variable we have is number of students and then we have our array which has sixty elements in it, so we have sixty integers one after the other. So, that goes up to here; it starts at a thousand and four. First index would be zero then one, two, three, four, five, and so on up to, it's going to be fifty nine, right? Since it starts at zero, when it has sixty elements it would go up to fifty nine. After this array we have our current grade variable and our ind variable; they're all integers so each of them is four bytes. And let's start executing the program.

So first we ask the user how any students are going to be in the class. Let's assume the user entered four, so number of students would be four and then we start reading the grades. So let's start executing the for loop, ind would be initialized to zero. We read the first grade, seventy one, and then we assign our grade lists. This is grades list with sign index zero of this array to be seventy one. Ind increases, we read the next grade, eighty six, and assign grades list one. In this case, this element here to be eighty six. Again ind the increases, we read the next element, sixty eight, and assign grades list two in this case to be sixty eight. Ind increases, by the way it is still, ind is still less the number of students; three is still less than four. We have one more iteration: we read another grade, in this case it's going to be ninety four, and we set grades list three in this case to be ninety four. Once more we increment our ind, this time when we check the boolean condition, ind that is no longer less the number of students; four is not less than four. So we break out of this reading step. Basically, we read the entire sequence of four elements into our array data structure, right? All the rest of the elements in the array starting index four to fifty nine remain with the same value they had before that, we didn't change it. So, the logical size will name it is only four. Physically, this array is of size sixty; logically, it has only four elements in it and these elements are in indexes zero to three. Okay, let's go on to the next step where we calculate the average.

Above the Average Implementation 3.4

Okay, so let's calculate the average. For that we would need to go over the elements in the array and to sum them up. Let's create a sum variable. We'll start by setting it to zero and then iterate over the elements and add each of them into sum, so we'll use a for loop. Again index would start zero, it would increase each iteration and we just keep on going while ind is less than number of students. So this basically is the way we pass over an array; we start our index with zero, we move it, increase each iteration and go up to the logical size of that array. Each time here we just want to add the current element into sum, so sum plus equals, lets access our grades list at the index position. This code here will then just add all the elements in the array up into are some variable. After doing that we can calculate the average. So let's have a double variable; let's call it average and we can just set the average to be sum divided by the number of students.

One thing we should be careful though is when we divide sum and num of students, since they are both integers this division is basically integer division; it's div. So, we should cast sum and number of students in order to turn it into real division. So let's cast some to a double and number of students to a double and then this would be real division and not just integer division. So after we have the average calculated, we can output to the user that 'the class average is' and then just print the value of the variable average; that basically calculate the average here. Let's try to trace this, see if it all makes sense.

Above the Average Implementation 3.5

Okay. So now we have our sum variable, we set it to zero and we start iterating over the array. We first add seventy one into sum, that would make it seventy one, then our index would be two. I'm not changing the index value but the index is going to change again, zero, one, two, three, four, as we iterate over the array. So, a second iteration index is one, we're adding eighty six to our sum, that would make it one fifty seven. Then we're adding the sixty eight, it would make sum two twenty five and eventually we'll add ninety four that would make sum be three nineteen, I think. Then we have our average variable and average is three nineteen divided by four, which is a seventy nine point seven five. We just print this value as the average. Now, we'll need to make another path over the numbers in order to figure out which ones are above the average. Let's go and do that.

Above the Average Implementation 3.6

Okay then. Now let's make one more pass over the data in order to print the ones that are above the average. Let's first start with announcing to the user that 'the grades above the average are' and now let's start iterating over our array.

So once again, we'll have our index starting at zero, incremented each iteration and it goes as long as its value is less than number of students. And now we have to figure out if we want to print this current element or not, so we'll just if the grades list, our current element, grades list ind that's the element we're currently looking at, is as we asked it to be above the average, so greater than the value of the average. Then, let's just print this current element: cout grades list. And let's also space after each print. Yeah, so that should go over the entire set of elements and print only the ones that are above the average, spacing them. After we do all of that, let's just break the line so cout endl, and that should be it.

So we have our first iteration to read the grades. Second pass, in order to calculate the average and eventually, last pass over the elements in order to print the ones that are above the average.
Let's just execute it, make sure it all behaves as we expect it to.

Build success succeeded. Please enter a number of students. That would be four. Oh, we didn't ask the user to enter the grades. We'll fix it in the second; it's now waiting for the grades. So let's have seventy one, eighty six, sixty eight, ninety four. And then, it just says that the class average is seventy nine seventy five, and the grades above the average are eighty six and ninety four which is great.

Let's just add this message here before starting to read the grades. Let's ask the user enter the students' grades, separated by a space. I think that should do it. Let's test it. Okay. Building. Number of students is four. Here is our message so let's put seventy one, eighty six, sixty eight, ninety four, and so the average is, again, seventy nine seventy five and we also have the numbers that are above the average. Great.

Syntactic Notes 4.1

Okay so I want to tell you two more little things before we end this model. So first let's recap what we already know about static arrays. So we know that the elements of the array are stored continuously in the memory, we know that all of the elements are of the same type and we know that we can access them using a zero based index system, meaning that the compiler basically can figure out the address of a

certain element just by knowing where the array starts, what index we're looking for and what's the size of each element in the array. By these the compiler can just figure out where each element is located. We also said that when we declare our array, we must supply its physical size as a constant. In this case int arr of size six, this six must be given at compile time.

I want to say two things, two more things about static arrays, two more syntactic issues that I want you to be aware of. First is that the arrays name is a legal C++ expression. Basically saying that if we create an array of, let's say, size six the name of the array ARR is legal C++ expression. We can even print the value of this expression we can do cout arr, not arr zero, one, two, obviously, but we can cout the value of the arrays name. The value of this expression, you can try to guess, but I just tell you that its value is the address in the memory where the array starts. So, if this array starts at address one thousand, this cout would just print one thousand. So that's one thing I wanted to say.

Another thing, another syntactic feature that we have for arrays is way to initialize array. So initialization of array and this thing is only at declaration. So, typically we can create an array of size six if we want to assign it with values we just go arr zero equals five, and arr one equals seven, and so on. But I want to show you a different syntax for array initialization; we can do something like that int arr six equals and then inside curly braces, I can just list the values so I can have five, seven, six, two, I don't know, one, fifteen, whatever. That would not only create our array of size six but it would also initialize the elements of the array to be the ones we gave in this list here. I just have to say again, that this is valid only at declaration; we can't create let's say an array in arr six and later on do something like arr are equals, I don't know, some list of values. That is not legal.

Another thing I want to say here, you don't have to put a list of all of like, in this case six values. If you have less than six values, let's say if you give only three values in your list then these three values would be the first three values of the array and the rest of the array would just be filled with zeros.