Pointers, Arrays, New and Delete, in C++

Memory for programmers can be a very abstract thing, hardware can seem very far away, but not so much in C and C++. Here it is said in jest, that segfaults are the way we learn. That is only half a joke. What I really mean by that is: in C/C++ memory management of arrays is really what we are doing and so there is no escape. We must play with memory and if we want to be good at our jobs we want to be good at playing with memory and arrays.

Thomas Edison did his lightbulb with a million failures to figure out the one design that worked, and sadly this is the only real way to learn in c/C++ how to deal with the memory. So that is kinda what we are doing today with this lesson.

Arrays are where we will begin, and that is because we already know arrays, from one of my previous videos, so if I just open a blank program and create an array of five elements using the square brackets that's cool but that doesn't do much for us. We can output this array with a little loop. And that is pretty much what i did with python3 in the one video.

**Int main(){  
 Int arr[]={1,2,3,4,5};**

**For (int i =0; i<5;i++)**

**std::cout<<arr[i];**

**Return 0;**

**}**

But if we change our array to like try to assign more objects, like we say:

**for(int i=0; i< 39; i++)**

**arr[i]=i;**

We get us a little undefined behavior from that output. Undefined behavior is basically exactly what we never ever want from a computer. You notice you will never get a wrong amount of money from the ATM. They pay the people mucho denero to never never get any thing from the computer that they didn't tell the computer to put out. This is why your processor runs so well, missiles fly straight, your video-games usually dont glitch and the stock market can count on a trade being closed. In C/C++ you must be careful because the compiler will very rarely stop you from doing things that are undefined, thus debugging, debugging, more debugging and making sure that we never give up control of our memory to random chance.

So that seems a bit harsh, but that is C, and that is the way it will always be. Some things in programming are very cool and relatively easy, this not so much. Good programmers find memory management a tedious and difficult task, so feel free to be frustrated.

Getting rid of this loop we can start to do some things, if we take our array, and just output the array itself, you can see we get this really weird output that doesn’t make any sense at all.

**std::cout<< arr << std::endl;**

So what the number is, is the actual location in the computer that the array begins at in C we like to call it a pointer, but that is weird that is not what we want, so now if we put our square brackets in there, right like we are trying to access it as an array.

**std::cout<< arr[ ] << std::endl;**

You can see we get a compiler error, that is because when you are using your square brackets you need to give the compiler a position in the array for it to use. So now if we say.

**std::cout<< arr[1] << std::endl;**

Now you can see that we get our second, not first, position in the array. This is because if you remember from that python episode, in computer science arrays begin at 0 and not 1. Just to prove that we can change in our source code. We change our second array point to like 356, then save and compile, and there it is, and now if we say [0] and we save and compile, now we get the beginning of our array.

So now that we have our understanding of how to work with arrays, we can move on to pointers. If I take and make a pointer, call it arrP, for array pointer. We do this with the asterisk symbol.

**Int \*arrP;**

Now that we have a pointer we can run with it, no not really, you can’t run with anything in the computer because you might drop it and computers are expensive… Ok lol no more corny jokes, but I can assign it to be the address of my array like this,

**arrP=arr;**

Now, it kinda follows that we now can say like before.

**std::cout<< arr << std::endl;**

This of course will be our array pointer as before, but if we also do this.

**std::cout<< arrP << std::endl;**

And we save, compile and run, we can see we have both our pointers that are the same. What's more if we take our array and our pointer and we add some braces to access a member, you can see we Now that seems simple enough, but with everything in Science and Math, it is simple in principle, it is just very complex in application right.

The reason why all this is important because it connects to the lesson on functions, because I am about to show you, how you can pass arrays between functions. You might think oh well you just put an array in the function parameters, but you have to remember that the function parameters are going to pass a copy of your data to the function. So while you can pass the entire array in C++ this is just not how such things are done.

You pass the function a pointer. Now we are going to just define inline, I know I showed in the lesson on functions to declare the function above main and define it below, but today we are just going to define and declare inline, meaning at the same time. In short, to save time I am just gonna do the whole function right here, no separation of declaration and definition. And it is a void because it is not returning anything.

So:

**void printFunct(int \*arrayPointer, int count){**

**for(int i=0; i<count; i++){**

**std::cout<<arrayPointer[i]<<std::endl;**

**}**

**std::cout<< std::endl;**

**return;**

**}**

Now we can see in our function that I am actually just using this thing here called arrayPointer as the array. Just to prove my point a bit. I can write another function that takes the array and gives us the pointer. This function will be of type int \* because we are returning the array.

**Int \*giveArrayPointer(int \* array){  
 Return array;**

**}**

Now notice in that last one that in our return statement that is all we are doing we are passing the thing we just got back, and you would think it would be like ‘**return \*array**;’ right, but it isn't, because we already told the computer in our parameter that this item, object, thingy we are calling array is already a pointer.

So now if we use this return array function to send it back the pointer from our array, right next to these other ones in main like this.

**Std::cout << giveArrayPointer(arr) << std::endl;**

We can see that we get the same thing from it, to the point that we can do the same thing with that pointer we made earlier.

**std::cout<< giveArrayPointer(arrP) << std::endl;**

The point of all of this is to show that we can use the pointers to move our array around and we can basically treat our pointer as an array.

So we can now use our print array as you can see on all three of these examples.

**printFunction(arr, 4);**

**printFunction(arrP, 4);**

**printFunction(giveArrayPointer(arrP), 4);**

Now if we do all this we get our array three times.

This allows us to pass arrays around without having to copy the entire array, and this is the way we work with arrays in c/c++. If we want we can re-assign the value of the items in the array with a function like this.

**Int \*alterArray(int \* array, int count){**

**std:cout << “Please enter numbers for an integer array: ”<< std::endl;**

**for(int i = 0; i< count; i++){**

**std::cout<< “>>: “ << std::endl;**

**std::cin>> array[i];**

**}  
 Return array;**

**}**

We can now use our functions together again in main.

**printFunction(alterArray(arrP), 4);**

And now we can put our own numbers in the array during the operation of the program and you can see we can change the numbers in the array, I said that position 2 was like 356 or something, and it prints as that first, but after that you can re-assign the value of that position during runtime.

Now that is all cool and all, but notice that we are only changing the value of the item in the array. Say for education’s sake that you wanted to change not only the values in the array but the actual number of items in the array. We could make a huge array and then keep track of how much we are actually using, and that can actually be fine for simpler stuff, we can even do it in a function, like this.

**Int \*createConstantArray(int count){**

**Int array[5000];**

**std:cout << “Please enter numbers for an integer array: ”<< std::endl;**

**for(int i = 0; i< count; i++){**

**std::cout<< “>>: “ << std::endl;**

**std::cin>> array[i];**

**}  
 Return array;**

**}**

Notice that we are just taking the number of the items we want, and not an array pointer. We are creating a new array, so if we try to show this pointer it will be different than the other array, right.

In main we can say:

**std::cout<< “Hi, please enter the number of items for your constant array: <<std::endl;**

**Int count;**

**std::cin>> count;**

**createConstantArray(count);**

If you capital K know that you will be getting arrays that are only so deep, you can use this, with smaller arrays of smaller items that can save time from allocating and deallocating stuff from memory. It works ok for some implementations, not often, but you should not totally forget that constant arrays exist after I show you the next part.

Now we work with something new, specifically the c++ keyword **new**. **New** goes with **delete**, you might say they are opposites. We call new to add **new** memory to a program, and **delete** to give it back to the operating system. So if you need to dynamically allocate memory, you use new. What that means is if you need to change the size of memory you are using during the running of the program, you use new. We will be working with these operators more eventually but we will keep it short for now, this is already a long video.

We are going to delete everything in main for now so we can have a clean slate, and the work from there, we can just leave the functions there of course because if we don't do anything in main with them we can just ignore them.

So im just gonna write this simple program in main.

**Int main(){**

**Int \*arrP;**

**Int count;**

**std::cout << “Please enter an integer number as an array index: “;**

**std::cin>>count;**

**arrP = new int [count];**

**//now that we have our array we can process with it**

**for (int i = 0; i < count; i++){**

**arrP[ i ] = i;**

**//iterate over array and do stuff**

**}**

**Delete[ ] arrP; //arrP is an array we use square brackets with delete**

**}**

There that is the basics of getting new memory in c++, but just to be more familiar I want to write a function that appends to our array, append means add to the end of. If we had an array a[3] ={1,2,3}; and we wanted to append it with {5,6}, we would get the a[5] = {1,2,3,5,6}, that is if we just do it, with pen and paper or something, programming this into the computer.... If we just gave just the number items we wanted to append, and our items, we might try to do this, a[3]={5,6,3}, or we might get a[5]={5,6,0,0,0}, or any number of things. This is gonna be a pain because we will have to take two integers and our pointer **append(int, int, int\*);,** because we have to keep track of where we want to insert our new Items. Because how can the computer know where to insert it unless we tell it. We could just take our old array and the new size we want, and just reallocate that array, but then we wouldn’t have any way to keep our old data, because we wouldn’t know where to stop collecting items for storage if we did try to store them. Thus we take the new and old count and the pointer.

So:

**append(int currentCount, int NewCount, int\* arrP){**

**//create a new array to store all items**

**newArrP = new int [currentCount + newCount];**

**//first copy old items**

**for (int i = 0; i<currentCount -1; i++){**

**newArrP[ i ] = arrP[ i ];**

**}**

**//now insert new items**

**for (int i =currentCount -1; i<newCount-1; i++){**

**std::cout << “Enter Array Item: ”;**

**Std::cin >> newArrP[ i ];**

**}**

**//now call delete on the old array**

**delete arrP[currentCount];**

**//now return the new array**

**return newArrP;**

**}**

So that was fun, but programming time over, lecture time now.

In the programming world we would call this code here in this last function fragile and confusing, if we were being nice. For instance if we had somebody who did not understand that our **newCount** is the amount of numbers we are adding, not the total count of the new array, they would add far too many entries. Equally bad is what would happen if you somehow failed to catch the array in the calling code, that is, in **main**: you call the function without assigning the return to an array in that calling code. If you did that you would just lose your array entirely.

What is the solution then, well what if you could tell somehow where the end of your array is? What if the last item in the array always said, hey! Go no further! Well they have those, we call those null terminated arrays. Null means nothing or not applicable, it means no, this is that type of thing, but that thing is not here, that thing has no value, or its value is zero. A null pointer for instance points to nothing. We use a type of null terminated string all the time, when you use a **std::string** you are actually passing a null terminated array of characters. But null terminated arrays are only a stopgap measure, what if we just had items that pointed to the next item, and the last item had a null pointer… well we would have a list!

More on lists in a minute, when working with lists we use new and delete a lot.

New should always be matched by delete, failure to do so will result in memory leaks and other issues that can lead to undefined behavior or a runtime crash.

Memory leaks are when your program fails to deallocate memory and a crash caused by this will always result in undefined behavior. That is, the computer gets some memory from ram, does its thing, then gets more memory, then its thing and gets more memory… but you are never giving any memory back, you are never deallocating, eventually your program goes to get more memory and there is none to be had, and bam crash and undefined behavior. As we stated earlier undefined behavior could also be described as a programming failure, so we wanna avoid that right.

Thus when you use new you need to be careful to use delete as well.

At some point in this series, the second to last episode probably, I am going to show you how to make a linked list in the computer memory. The list structure is what it sounds like, it is a collection of objects in computer memory, like our array, but with larger and more complex objects. Say we have a list of names and desk numbers grouped together under an object called profile, for a classroom of school kids, we know there wont be too many profiles so we know we can cycle through the list fairly quickly, so to save time we just code in a link between each profile, and keep a copy of the first item, then we can add new items, delete them and sort them, by rearranging the links. I will show you how to work with that list, how to sort it, insert and delete items, all that. That is in basic terms what you will learn with an AA degree in computer science that you can’t get by just playing on your computer and reading manual pages.

The use of new will be to make new items in the list, delete to delete items, and the list will be in many parts accessed like we were just doing with the array. Sorting will require some complexities and so there will be much ado on that matter. While you can do all these sorting algorithms with an array, and you can add and insert items in a list and save time in computer operation because the list object contains a direct pointer to the next object. This is so you don’t have to call a function to change array position, and you don't have to iterate over an array with pointers, you can just iterate to the next item in the list, because each position in the list has a direct link only to the next item. This makes sorting algorithms and the like much faster. So that's all I have for now, this was a long lesson. Now homework:

For homework I want you to make a program or function that takes an integer and uses that as a size for a dynamic array, then fills that array with a fibonacci sequence. Bonus points if you can code in some error checking and prevent the entry of values that are too large. That is all, have a good night.