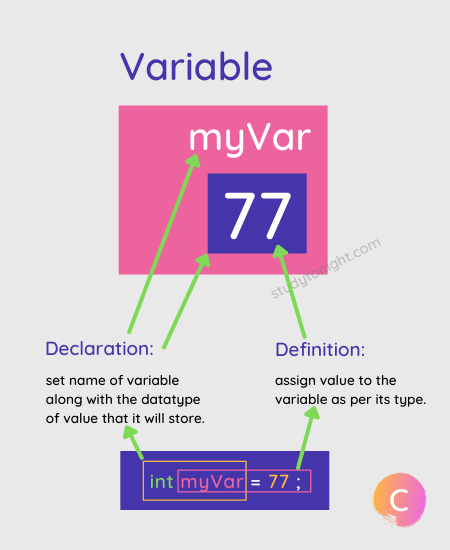
Now whenever we declare a variable, The generally taught concept is,

A memory space is allocated and the content is stored in it. The memory is then labeled as <variable name>.

Now lets understand the actual working here.

Yeah the image besides shows pretty much the declaration variable and its supposed memory allocation.

Lets get into the part where we have to access these variables. How does it happen?

Lets say the code is,



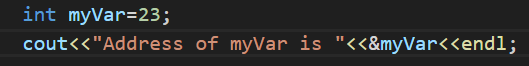
So how is this going to work. Well one theory can be, it searches for memory box named myVar and then accesses its value and prints it. Or

There is something else to it.

Note that previous assumption is wrong as there is no concept of labeling the memory units with names. The only way for compiler commands to access memory units is via their addresses but as a programmer since we don’t know the memory addresses we see that these are accessed via variable names. This is facilitated via a tool our compiler has that’s called symbol table that maps the memory address with the variable name.

**Address Of Operator(&)**

This operator helps us accessing the address of a given variable. Lets see the code.



The output of this code is as follows.



Note that how the addresses are stored in hexadecimal formatting.

**Pointers**

Treat pointers as datatypes. Like int stores integers, char stores characters, float stores decimal numerals, Pointers can be defined as datatypes that store memory addresses. Lets see few code lines to describe various aspects of usage and declaration of Pointers.

Here is how we declare a pointer.



Here is how we store addresses of the variables in pointers.



What happens above is we are storing the address of variable “myVar” in the pointer ptr.

Lets print it and see the output.



Lets see the output.



Lets see how using pointers we can access value stored in the address to which the concerned pointer is plotting.



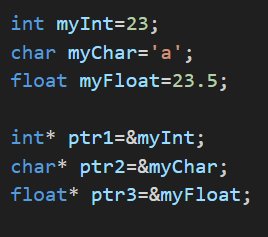
The output can be observed as follows.



Lets see a few more things. Lets decode Pointer a bit. The general code of pointer declaration is,

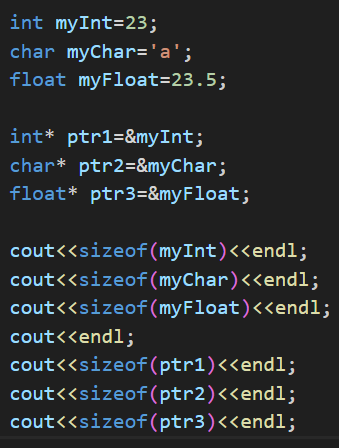
**<Datatype Of the Data Stored In Variable With the Pointer Will Point>\* <Name>**

That is,



As observable, as per the datatype of variables declared, we see the same datatype of the corresponding pointers.

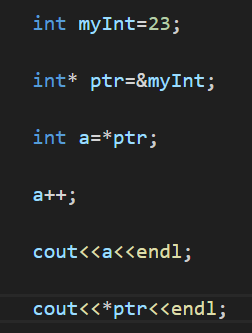
Lets talk about the sizes. Lets see the following code and it output.



What we observe is that, no matter what the datatype of a value stored at a particular address is, the size of a pointer variable remains 8 bytes.

Lets move on look at cases of Pointer Arithmetic, Semantics and Copying.

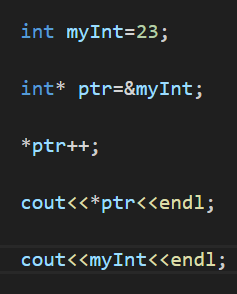
Consider the following code.



Guess the output of following code. We initailize myInt with 23, store it address in a pointer, then that value in pointer is copied to a variable “a” and its then incremented. Will the value of \*ptr change?

No. The value of \*ptr, matter of fact, myInt does not change. Its because a is storing the copy of \*ptr in another memory, so when changes are made to a, it has nothing to do with ptr or myInt.

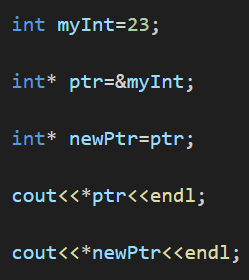
Examine another code.



Here we initialize 23 to myInt, and save the address of the variable to ptr. Then we increment \*ptr. We know the value of \*ptr will change as its being incremented. Can the same be said about myInt?

 As observable, both change. This is because ptr is linked to the address of the myInt variable, as a result when \*ptr is changed, changes are reflected to variable stored in the address to which ptr is pointing.

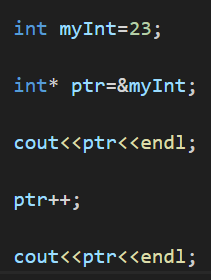
Lets examine a code to understand the copying of pointers.



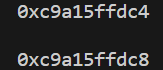
Yes that’s how copy the contents of a pointer to another pointer, as done in case of ptr and newPtr. Note the most important line in the code is the 3rd line. We know pointer variable store addresses and since we are creating a new pointer while trying to copy the contents of a pre-existing one, we see that we assign the new pointer the very address to which the old pointer was pointing to. As a result both old and new pointers point to the same memory unit,myInt, and any changes made in these pointer variables reflects in the main memory too. Though lets just see the output,



Lets delve into pointer arithmatics. See the examplar code.



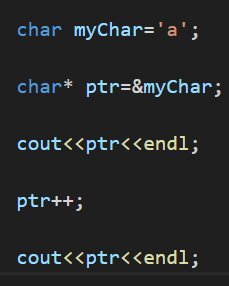
Everything seems normal,right? Here we declared a integer myInt=23, and we stored its address in the ptr pointer variable. Now note that till now, what ever operations we performed in this section were in reference to, \*ptr, which was nothing but reference to value stored in the address to which the pointer ptr was pointing. Now here we see that we are directly incrementing, ptr itself, so the question is what will be the output?

First print show the memory address of memory unit, to which myInt is mapped to. Second also seems like a memory address, but it has been incremented by 4, but we incremented ptr by 1 only, so what happened?

Well, when we wrote ptr++, it actually performed,

**Ptr=ptr + <Size of datatype pointed by ptr>**

In case of integers, the size is 4, hence we see increment by 4 in the output above. See another example to clear your doubts,



The output of the code will be,

Now there is one bad habit that we should always save ourselves from. It’s the initial declaration of pointer variable. See the code for better understanding.



Initialising a pointer this way is wrong. Instead do it like this,

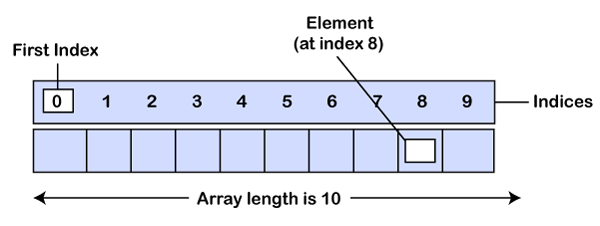


Advantage is that in place of pointing at some unknown, non-existing memory unit, we point at one which we have created.

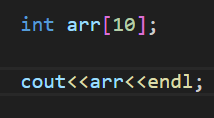
Let venture into something completely different. Lets talk about arrays. Lets first declare a array.



Lets see a box representation of the array.



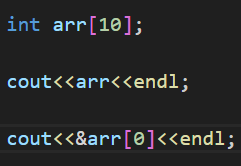
Lets execute this code and see what happens.



The output is a memory address. This memory address represents nothing but the index of the first element of the array, that is arr[0]. See the output.



Lets just confirm it via this code.

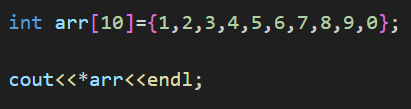


The output is,



As observable the addresses in either case are the same, hence our deposition was correct.

Now since “arr” returns address of memory unit pointing to first index of the new array created, lets see what happens when we execute “\*arr”.

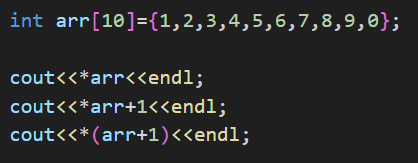


The output is,



Which is what we expected.

Let just see few set of commands and see what their output is to understand the function of pointers in a better manner.



The first code simply prints 1, with is at index 0.

The second code simply prints Elements at index 0 and adds 1 to it. So the output is 2(1+1).

The third code accesses index next to 0 ,since 1 is added to pointer, 4 is added address itself. Hence the element at index 1 is printed and the output is 2.

See it yourself,



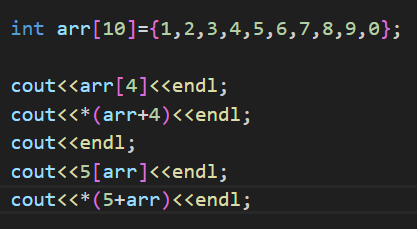
Lets now see the working formulae that explains the working of the 3 code.



The formulae states that arr[I] mean, arr+I enclosed in a pointer. As observable above. Another way it could be written is,



The formulae simply states I[arr] means I+arr enclosed in a pointer. Hence the two formulae are the same, just reversed. Lets see code to verify this formulae.



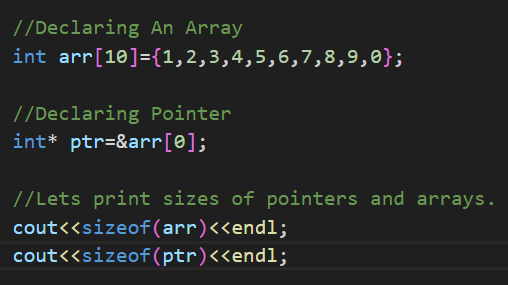
The output is as follows,



Well you can simply verify from the code that these do work.

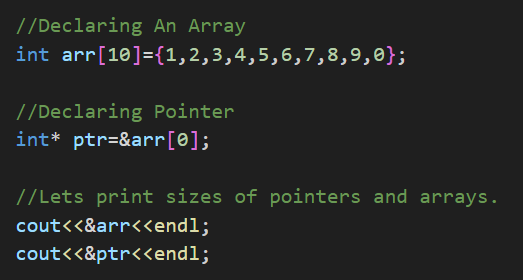
Lets differentiate between Array and Pointers.

1. The size of arrays and the size of pointers are different. See the code to understand better.

 The output are as follows. Clearly we are able to see difference in their sizes though they point at the same thing(arr and arr[0] addresses are the same).

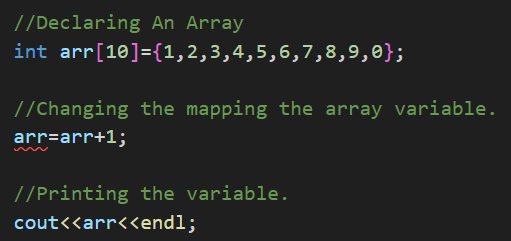
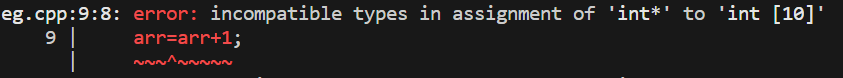


1. When we try and take out the address of arrays and pointers they point to two different addresses. See the code and see that.

Clearly the addresses printed out are different.

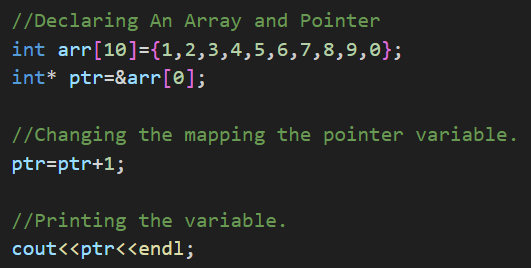


1. Symbol table does represent another difference between pointer and array.We know that when we declare arrays, the name of the array is mapped to 0 index of the array we just created. Note that we cannot edit the mapping or symbol table of the array name. Let see it with a code.



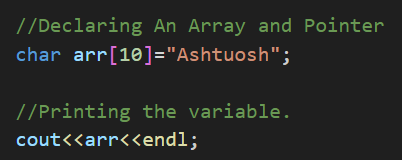
As observable error is generated.

Lets see the other code.

 As observable from the output no error is generated.



Lets move forward and talk about character arrays. We already know how to declare character arrays. Lets discuss them in context of pointers to have better understanding. See the following code,

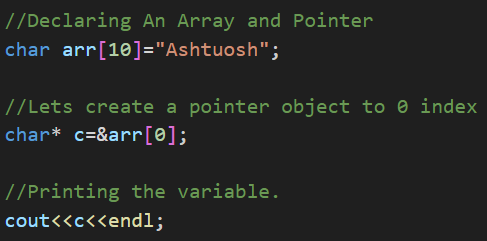


So if we have remembered our observations from integer arrays we may say, that arr will print the memory location containing ‘A’. Lets see the output.



Right so as observable, its not what we expected. Unlike in integer arrays, the very name of the character array prints all of its contents, rather than just the address.

Lets see another code,

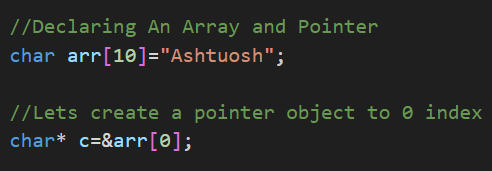
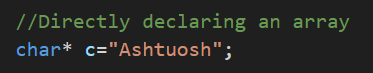


So what will c print? The address of arr[0] or something else? Let see.



Again the whole string. Lets understand what actually happened here. ‘c’ points to the location at which arr[0] is placed. But when ‘c’ is passed through cout, it starts from that particular address of ‘arr[0]’, print, will move to the next address of next element of array, arr[1], prints it, and goes on and on until it encounters the null object.Lets move forward.

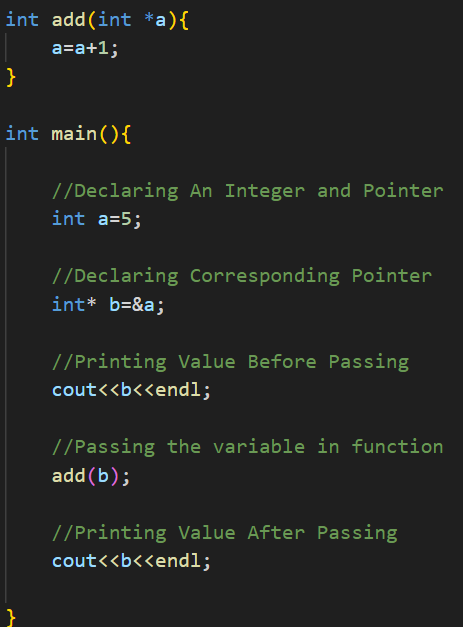
Now there are two ways in which we can declare pointers on a character array. They are as follows.

Lets understand which technique is right and wrong. In case 1, we see that firstly the elements are stored in temporary memory and then are copied to another memory but that is permanent.

Though in second case we observe that the elements of the character arrays always remain in temporary memory and are never transferred to permanent memory as a result its not optimized, hence its suggested to follow the first model rather than the second.

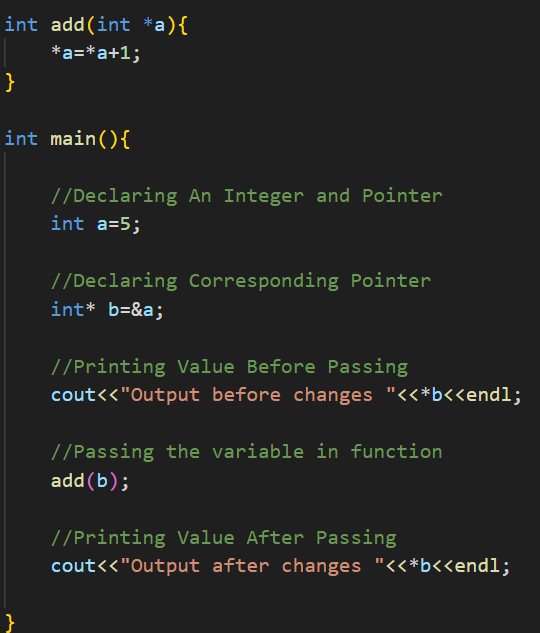
Lets move ahead and talk about how pointers and functions work together.Lets see the program below.

Lets see the output of the adjoining code.



As observable we see that there is no change in prints before and after the action of functions. This is because the variable ‘a’ in the functions ‘add’ is acting as a local variable, hence we see that changes made to ‘a’ here do not have global effects.

Lets now see another code.



Lets see the output.

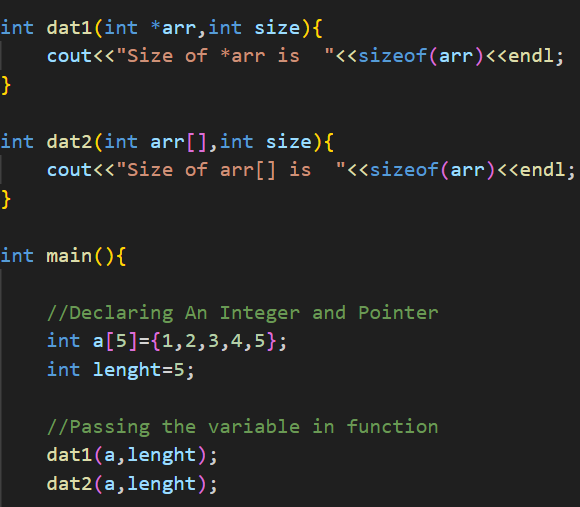


As observable changes are observed. This is because in the function ‘add’, we see that changes are being made to location in which the number stored are pointed by ‘\*a’, hence necessary changes will be seen.

Lets see how we pass arrays to a function via usage of pointers.

Below we see 2 ways via which we can pass a array into a function.

 Here we observe two methods via which we can pass arrays into function. Note that both are correct. Lets see code to verify this.



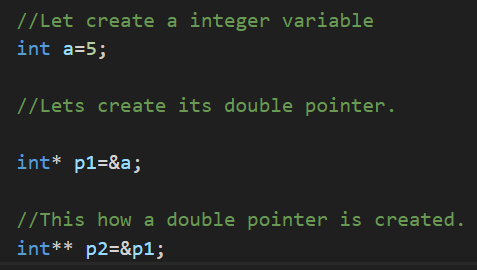
Let see the output here.



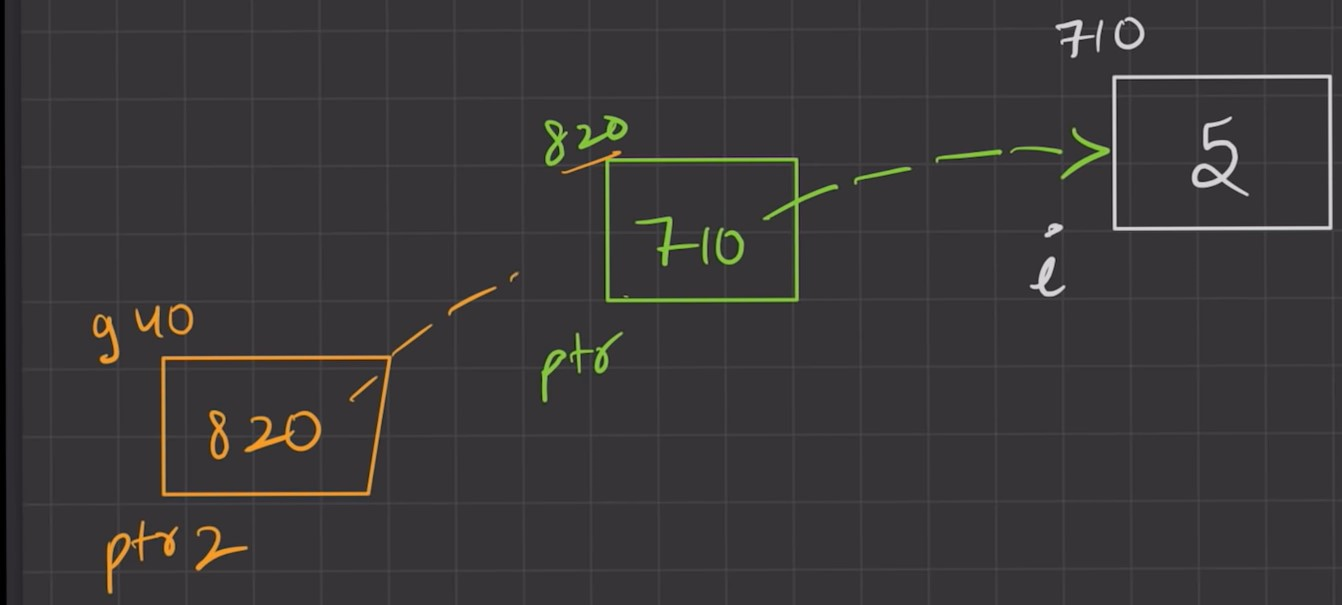
As observable both of them are nothing but two different styles of writing pointer that point to the same data structure.

**Double Pointer**

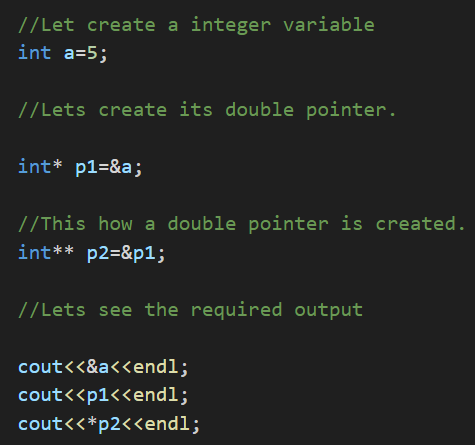
Lets see how to we declare it in the first case.



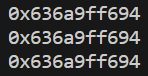
Lets see a diagram to understand who bis pointing whom.



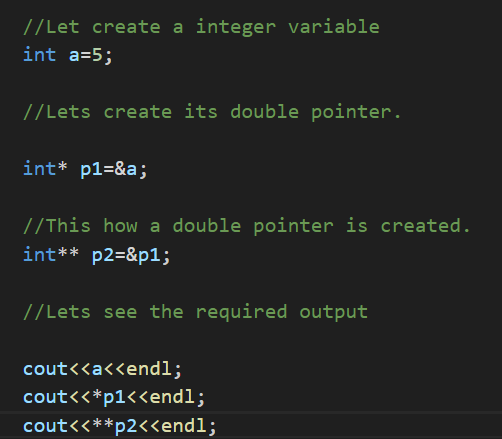
Lets see a set of code and their output. Lets see codes that print the address of the first pointer.



Let see the outputs.As observable below we see that all the output are the same.



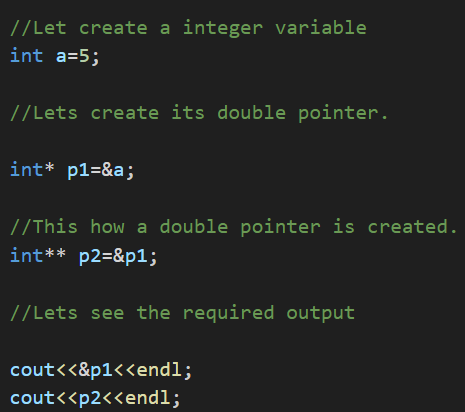
Lets see a set of code that returns what stored in ‘a’.



Let see the outputs. As observable below they all are the same.



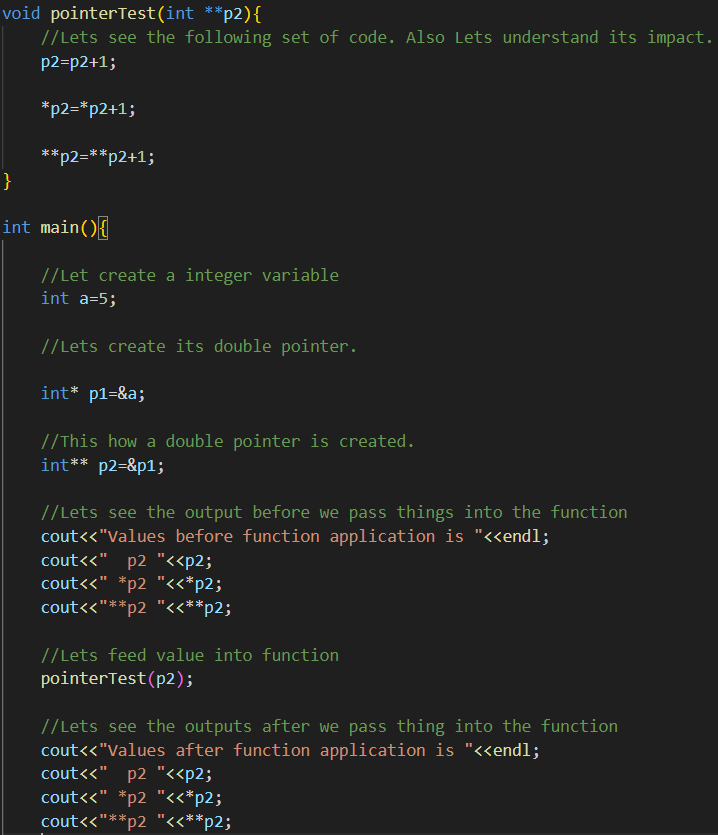
Lets see a set of code that print the value stored in p2.



Lets see output of the following code. All values returned are the same.



Lets now understand the functioning of double pointers in context of function. See the code below.



We have seen this exercise in case of first pointer. So we are seeing it again, but in the context of double pointer. We know in case of single pointer command 1 will make no change, while command 2 will, in the function. Is it the same in case of double pointers? Lets see. The output is as follows.