PART 28

As usual. Let's alternate some theory with the practice exercises to try to consolidate gradually and move forward. In this part, we will see some theoretical topics on topics that we have to know without trying to be very technical or heavy.

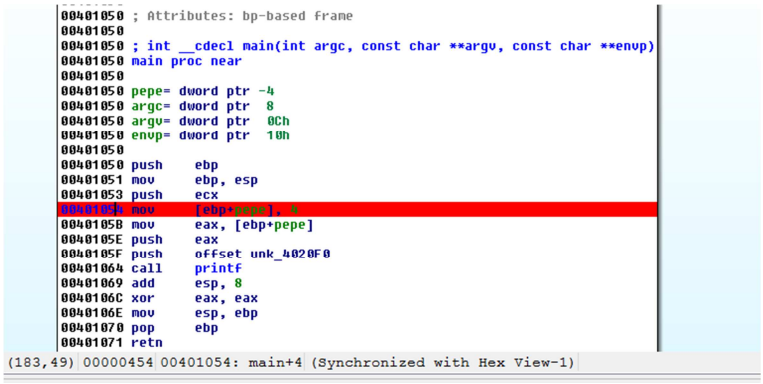
We have seen that when we define a variable for example:
int pepe = 4;

```
1   // ConsoleApplication6.cpp : Defines the entry point for the
2   //
3
4    #include "stdafx.h"
5
6
7   int main()
8   {
9       int pepe = 4;
10
11      printf("%x", pepe);
12      return 0;
13  }
14
15
```

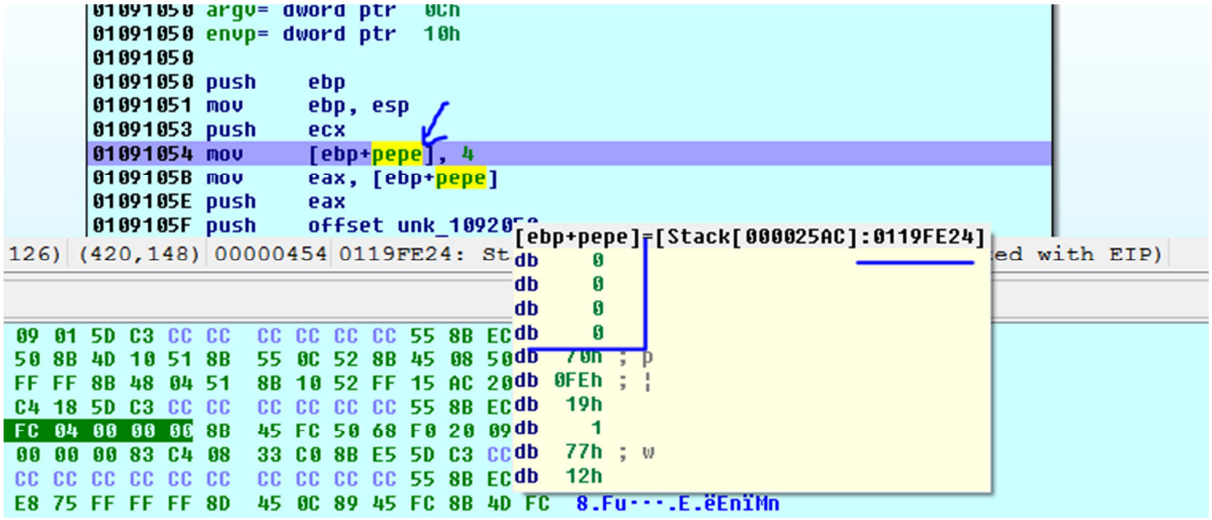
This will reserve the necessary space to store, in this case, an integer or 4 bytes in a memory location, and then when the value 4 is assigned, we will have the stored pepe value 4 in a memory address.

```
00401050 main proc near
00401050
00401050 pepe= dword ptr -4
00401050 argc= dword ptr  8
00401050 argv= dword ptr  0Ch
00401050 envp= dword ptr  10h
00401050
00401050 push    ebp
00401051 mov     ebp, esp
00401053 push    ecx
00401054 mov     [ebp+pepe], 4
0040105B mov     eax, [ebp+pepe]
0040105E push    eax
0040105F push    offset unk_4020F0
00401064 call    printf
00401069 add     esp, 8
```

There, we see that case. The int pepe variable and how it initializes it to 4, if we start the debugger and set a breakpoint here.

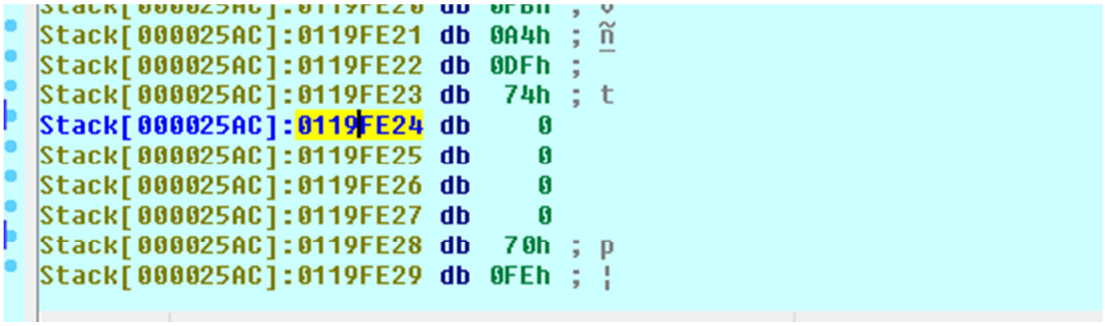


And we run the debug: LOCAL WIN32 DEBUGGER.



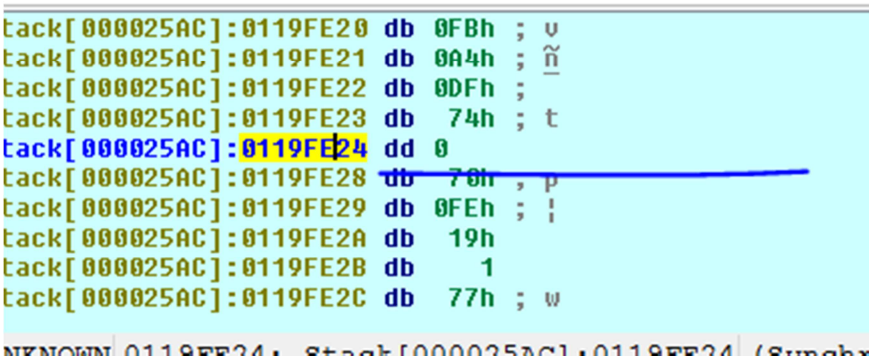
When it stops there, if we put the mouse on pepe, we see the memory address where the 4 bytes are reserved for the integer that is supposed to be stored.

If I click there in Pepe I will go and see (the address will not match yours)

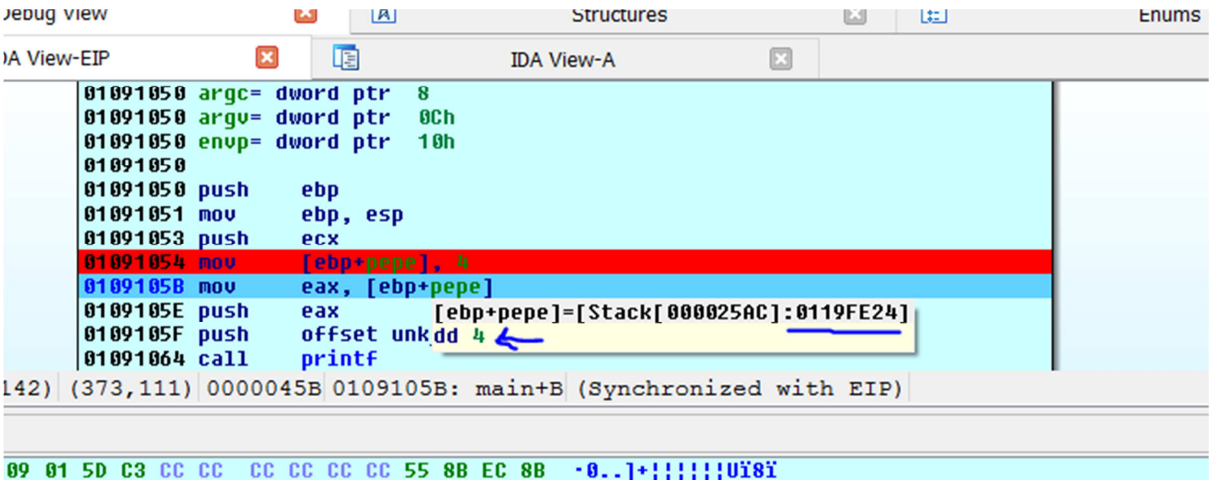


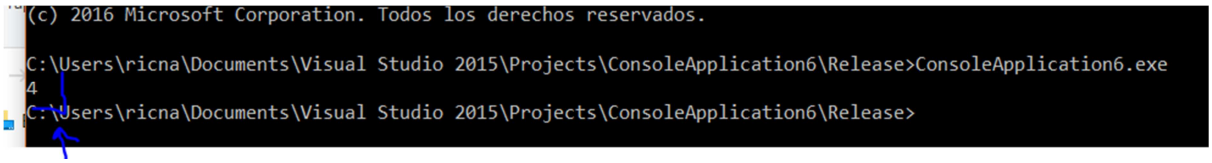
Since this is an integer, I can press D until I change to DD or DWORD.

View-EIP



There, it is marked as DWORD and with zero because it still does not save the value 4. If I execute the instruction that saves it and I look again.





It does not print anything until it does not finish and it closes. So, if I run it on a console, I see the 4 that is the pepe value. There, you have it like EJEMPLO1.exe.

What we see is that whenever there is a variable there will be a value and an address, in my case, the pepe value was 4 and the pepe address was 0x119FE24. Now, I will change the code, so that it does not only print the pepe value but its address.

```
1   // ConsoleApplication6.cpp : Defines the entry point for t
2   //
3
4   #include "stdafx.h"
5
6
7   int main()
8   {
9       int pepe = 4;
10
11      printf("%x \n", pepe);
12
13      printf("%x \n", &pepe);
14
15      return 0;
16  }
17
18
```

If we open EJEMPLO2 in IDA…

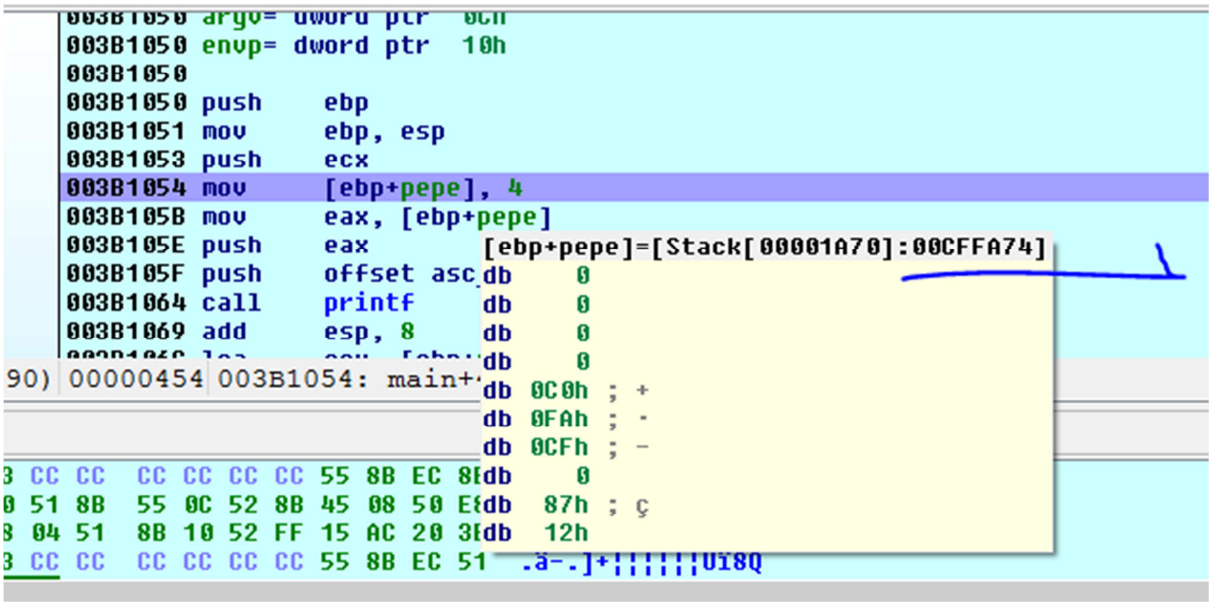
```
00401050 main proc near
00401050
00401050 pepe= dword ptr -4
00401050 argc= dword ptr  8
00401050 argv= dword ptr  0Ch
00401050 envp= dword ptr  10h
00401050
00401050 push    ebp
00401051 mov     ebp, esp
00401053 push    ecx
00401054 mov     [ebp+pepe], 4
0040105B mov     eax, [ebp+pepe]
0040105E push    eax
0040105F push    offset asc_4020F0 ; "%x \n"
00401064 call    printf
00401069 add     esp, 8
0040106C lea     ecx, [ebp+pepe]
0040106F push    ecx
00401070 push    offset asc_4020F8 ; "%x \n"
00401075 call    printf
0040107A add     esp, 8
0040107D xor     eax, eax
```

We see the two **print**. The first one using MOV passes the value 4 of pepe to EAX and then prints it, and in the second **print**, it finds the pepe address with LEA and prints it.

If I set a breakpoint on the mov…

```
00401050 ; int __cdecl main(int argc, const char **argv, const char *
00401050 main proc near
00401050
00401050 pepe= dword ptr -4
00401050 argc= dword ptr  8
00401050 argv= dword ptr  0Ch
00401050 envp= dword ptr  10h
00401050
00401050 push    ebp
00401051 mov     ebp, esp
00401053 push    ecx
00401054 mov     [ebp+pepe], 4
0040105B mov     eax, [ebp+pepe]
0040105E push    eax
0040105F push    offset asc_4020F0 ; "%x \n"
00401064 call    printf
00401069 add     esp, 8
0040106C lea     ecx, [ebp+pepe]
0040106F push    ecx
00401070 push    offset asc_4020F8 ; "%x \n"
00401075 call    printf
0040107A add     esp, 8
0040107D xor     eax, eax
0040107F mov     esp, ebp
00401081 pop     ebp
00401082 retn
(189,28) 00000454 00401054: main+4 (Synchronized with Hex View-1)
```

I run the debugger.



In this case, the pepe address on my machine will be 0xCFFA74. I go there and pressing D, I change it to DWORD.

```
003B1051 mov     ebp, esp
003B1053 push    ecx
003B1054 mov     [ebp+pepe], 4
003B105B mov     eax, [ebp+pepe]
003B105E push    eax
003B105F push    offset asc_3B20F0 ; "%x \n"
003B1064 call    printf
003B1069 add     esp, 8
003B106C lea     ecx, [ebp+pepe]
003B106F push    ecx
003B1070 push    offset asc_3B20F8 ; "%x \n"
003B1075 call    printf
.2) 0000046C 003B106C: main+1C (Synchronized with EIP)
```

I get to the LEA. If I put the mouse on pepe, I see that the value 4 is already saved when executing the LEA.

```
003B1053 push    ecx
003B1054 mov     [ebp+pepe], 4
003B105B mov     eax, [ebp+pepe]
003B105E push    eax
003B105F push    offset asc_3B20F0 ; "%x \n"
003B1064 call    printf
003B1069 add     esp, 8
003B106C lea     ecx, [ebp+pepe]
003B106F push    ecx
003B1070 push    offset asc_3B20F8 ; "%x \n"
003B1075 call    printf
003B107A add     esp, 8
0000046F 003B106F: main+1F (Synchronized with EIP)

EAX 00000003
EBX 00A1C000 TIB[00001
ECX 00CFFA74 Stack[000
EDX 74E95324 ucrtbase.
ESI 74E96314 ucrtbase.
EDI 74E96308 ucrtbase.
EBP 00CFFA78 Stack[000
ESP 00CFFA74 Stack[000
EIP 003B106F main+1F
EFL 00000216
```

I see that now ECX has the pepe address which is what it will print second if we run it out of IDA we will see how it prints, in each shot the address will change but it will print the value and the current pepe address.

```
C:\Users\ricna\Desktop\28\EJEMPLO2>EJEMPLO2.exe
4
bef748

C:\Users\ricna\Desktop\28\EJEMPLO2>
```

We see that when there is an integer, a buffer, a structure or the data type, we will have an address where it is stored (or where it starts if it is a buffer or structure) and a value that is the content located there.

POINTERS.

Since there are many types of data, there is one more data type that is used to store and handle memory addresses, it is called pointer.
A pointer is just one more data type, as int stores integer, char characters and float stores floating point numbers, so pointers store memory addresses.

For example, in the previous case what would happen if instead of printing the pepe address, I would like to save it, as it is a memory address, I should use another variable of type pointer to save it.

int * jose;

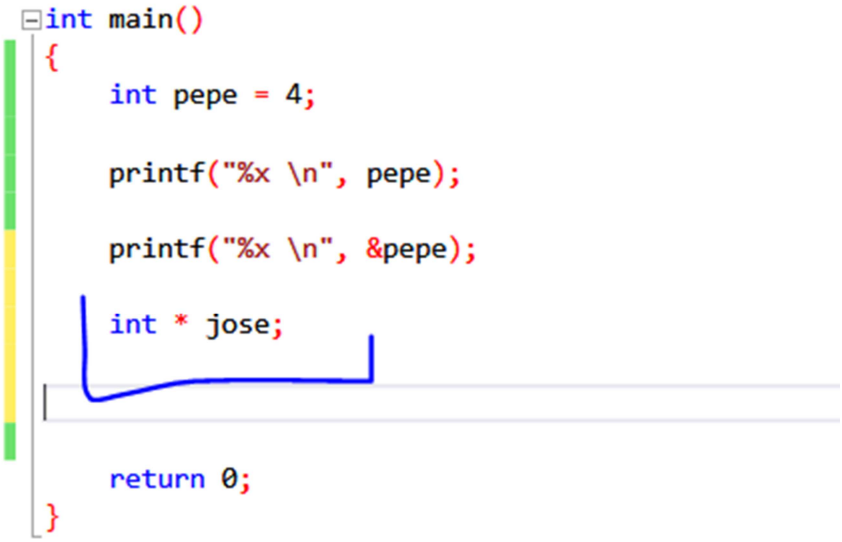
It's different from:

int jose;

The latter is an integer variable, while the former is a pointer type variable that stores memory addresses that point to integers.

Not only when defining a pointer, we define a variable that saves a memory address, but we say to that memory address, to what type of data point, if I try to save a pointer to another type of data, in this case, it will fail.

In the example above.



We see that jose would be a pointer to an integer but it has not yet assigned any value. I could assign it the pepe memory address whose value is an integer, so we meet both points: to keep a memory address and that that address points to an int.

```
1   // ConsoleApplication6.cpp : Defines the entry point for the console appli
2   //
3
4    #include "stdafx.h"
5
6
7   int main()
8   {
9       int pepe = 4;
10
11      printf("valor de pepe = %x \n", pepe);
12
13      printf("direccion de pepe = %x \n", &pepe);
14
15      int * jose= &pepe;
16
17      printf("valor de jose = %x \n", jose);
18
19
20      return 0;
21  }
22
```

There, we see that we assign to jose the memory address of pepe that as it is an int, itwill be OK. The types match.

```
00401050 push    ebp
00401051 mov     ebp, esp
00401053 sub     esp, 8
00401056 mov     [ebp+pepe], 4
0040105D mov     eax, [ebp+pepe]
00401060 push    eax
00401061 push    offset aValorDePepeX ; "valor de pepe = %x \n"
00401066 call    printf
0040106B add     esp, 8
0040106E lea     ecx, [ebp+pepe]
00401071 push    ecx
00401072 push    offset aDireccionDePep ; "direccion de pepe = %x \n"
00401077 call    printf
0040107C add     esp, 8
0040107F lea     edx, [ebp+pepe]
00401082 mov     [ebp+jose], edx
00401085 mov     eax, [ebp+jose]
00401088 push    eax
00401089 push    offset aValorDeJoseX ; "valor de jose = %x \n"
0040108E call    printf
00401093 add     esp, 8
```

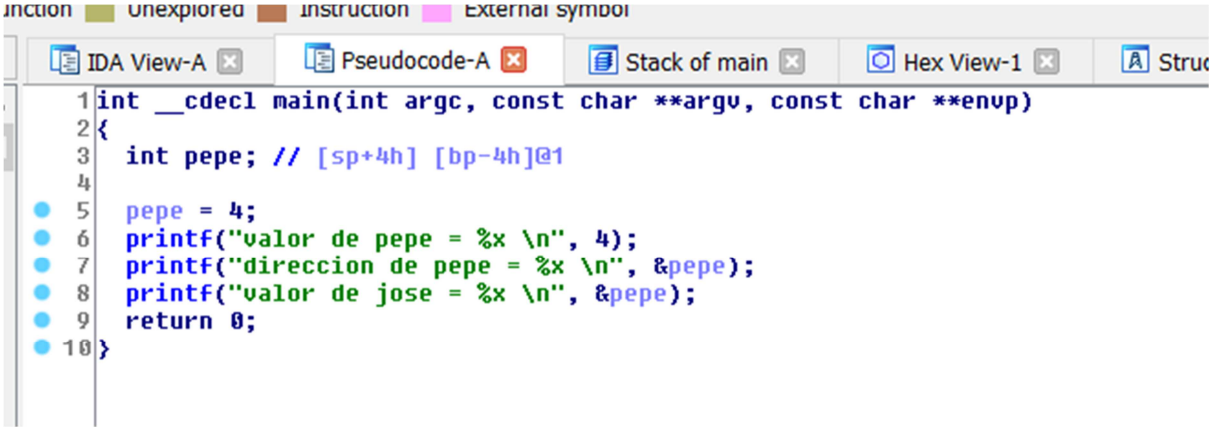
That happens there. It finds the address of pepe with LEA and saves it in jose as it is a pointer it serves to store memory addresses of variables and since that variable is an integer, it's all good, it will print the value of jose that will equal to pepe address.

```
C:\Users\ricna\Desktop\28\EJEMPLO3>EJEMPLO3.exe
valor de pepe = 4
direccion de pepe = 113fe5c
valor de jose = 113fe5c

C:\Users\ricna\Desktop\28\EJEMPLO3>
```

There we see and understand that a pointer is used for that, to store and work with memory addresses of variables.

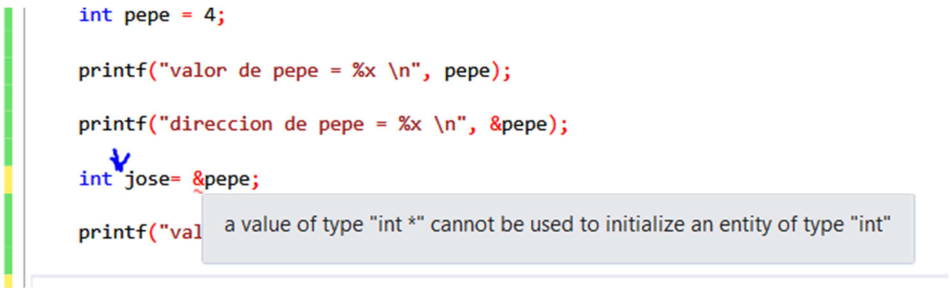
The pseudocode of the function by pressing F5.



It does not show us the jose variable because it uses &pepe instead of jose to economize directly. We will try to force it by changing the code in the following example.

When it comes to this point, one says, but well what difference is there to have to create a special data type that stores memory addresses? Couldn't it use an int and save there the address?

It would be something like this, removing the asterisk to the definition of jose, it will be an integer, and if I want to save the pepe address there.



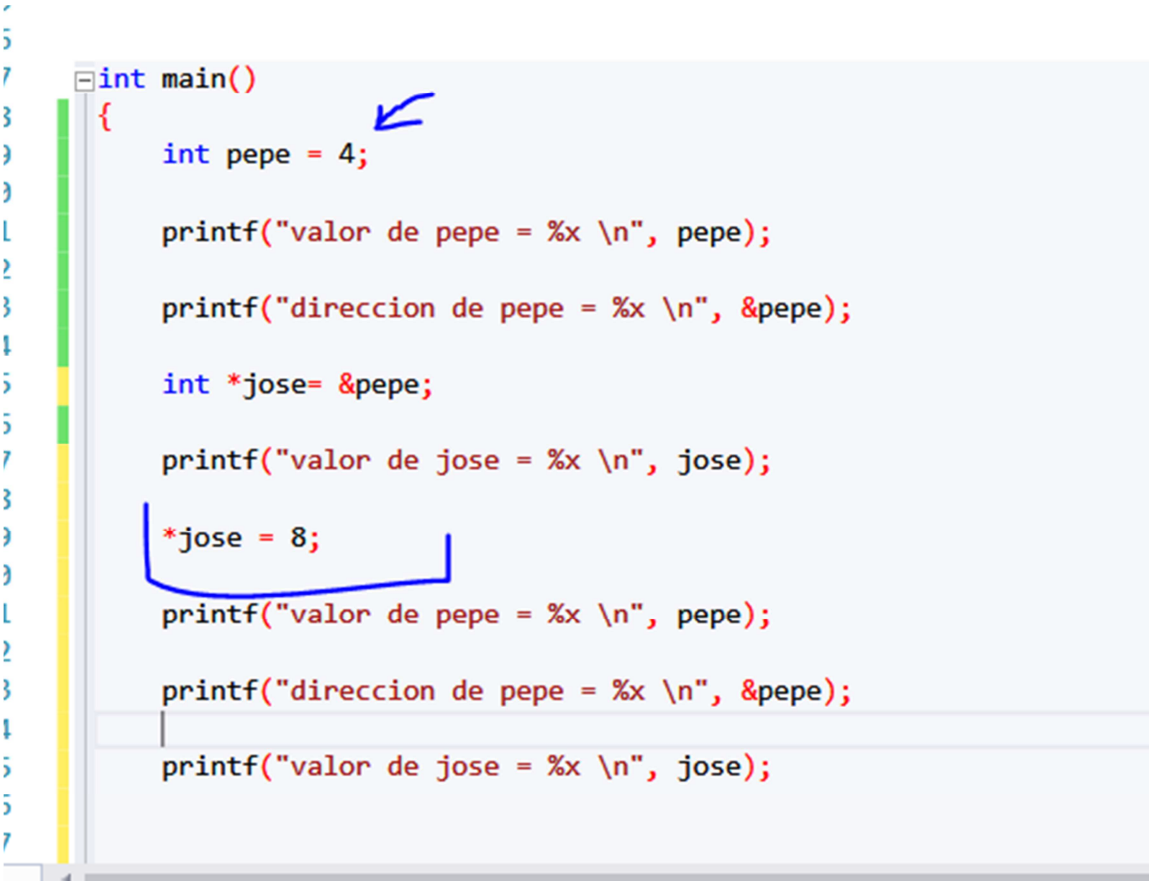
It does not let me. It gives me error, so, we have to use pointers, hehe.

The pointers also allow me to read, change and work with the values to which they point.

jose saves the memory address of the pepe variable and this value 4 with which they are related to each other, if I do.

*jose = 8;

The asterisk is used not only to define a pointer, but also to access the content it points to (in this case it is called indirection)



And as the value of jose is the memory address of pepe, jose is a pointer that points to the value of pepe or is 4 if I change it to 8, it will change the value of pepe also.

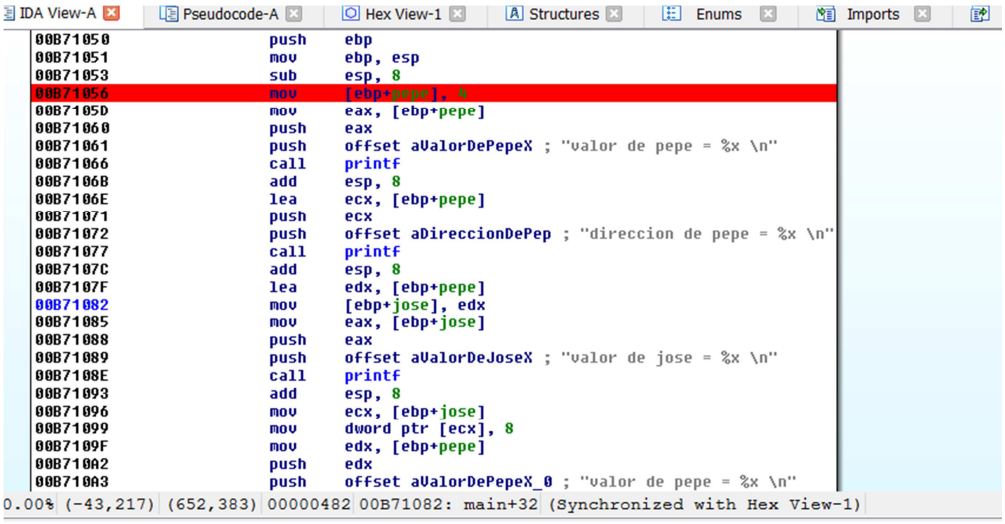
```
valor de pepe = 4
direccion de pepe = 10ffd98
valor de jose = 10ffd98
valor de pepe = 8
direccion de pepe = 10ffd98
valor de jose = 10ffd98
Presione una tecla para continuar . . .
```

We see that by changing the content of jose, we affect the value of pepe, which is logical, because jose has the pepe memory address as value and points there, now 8.

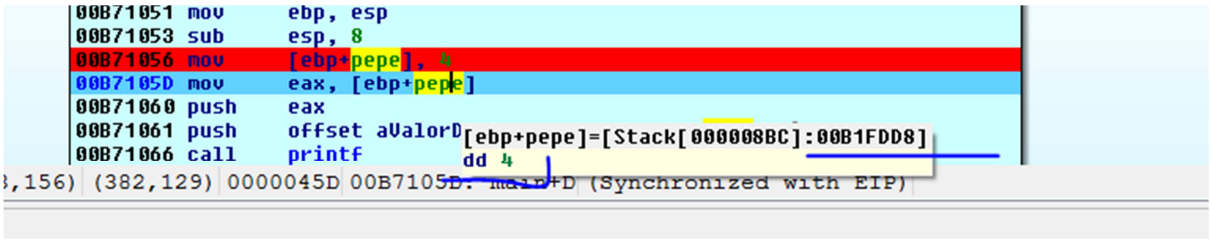
Let's look at IDA to see what it shows.

It is important and that is why I pass the compiled examples for you to understand this well and debug it and verify until you are sure.

We see that jose reads that it is the address of pepe and saves the value 8 in the content, if we debug it from the beginning.

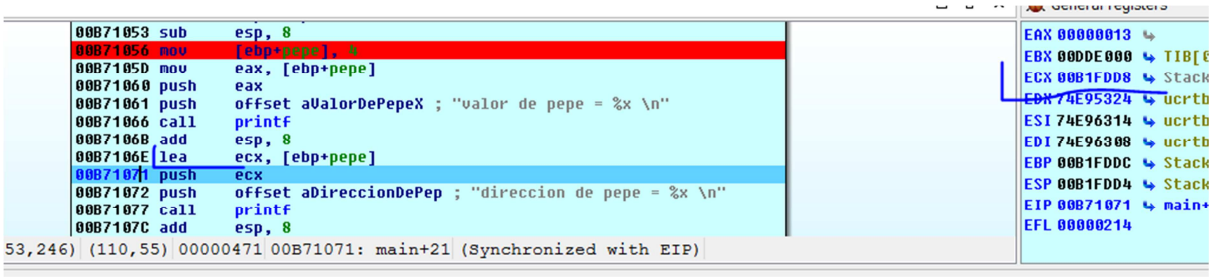


We run the debugger.

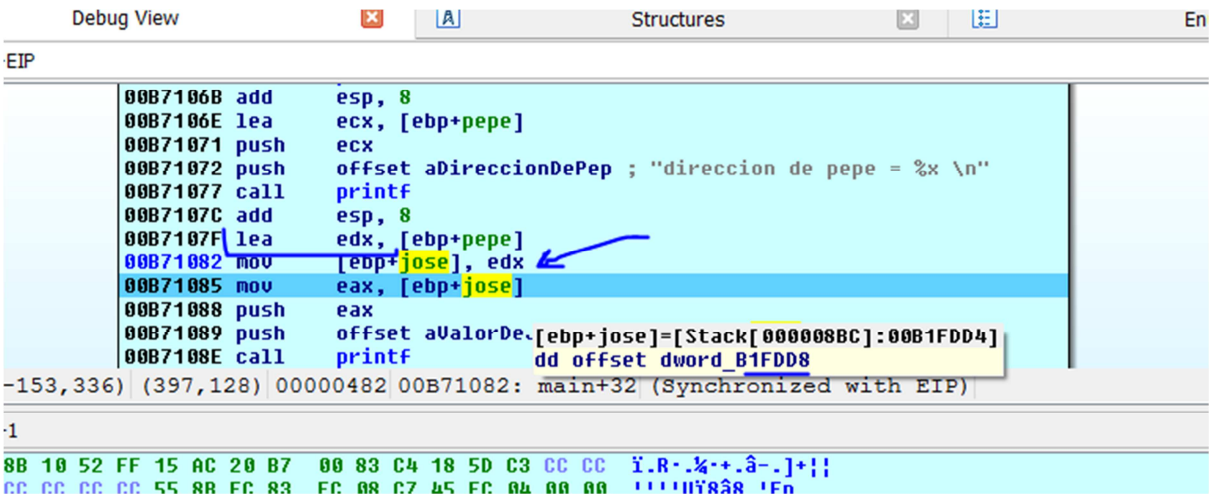


After saving 4, we see the address of pepe, in my case, 0xB1fd88 and the value of pepe 4.

If I keep tracing…



There, it gets the pepe address with LEA which remains in ECX and prints it, I still trace.

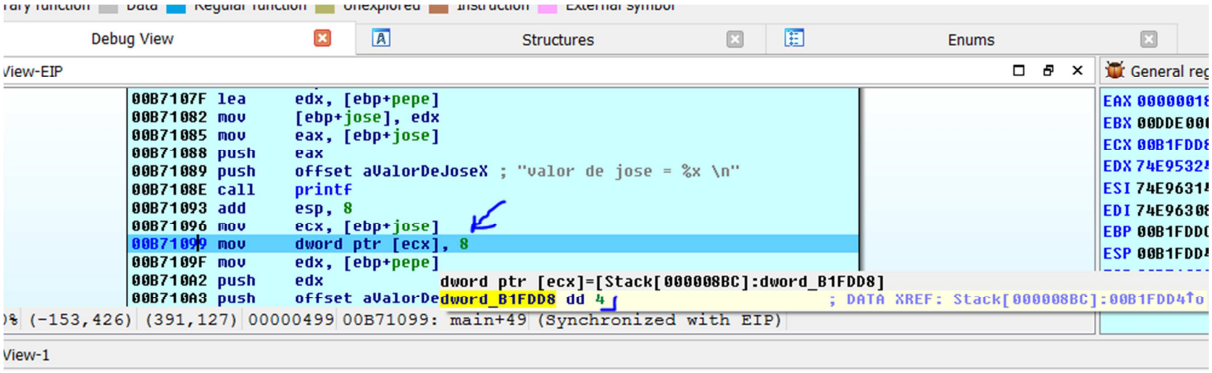


After saving the address of pepe in jose, we see there that it saved 0xB1Fdd4 that it will become the value of jose. (The same pepe memory address).

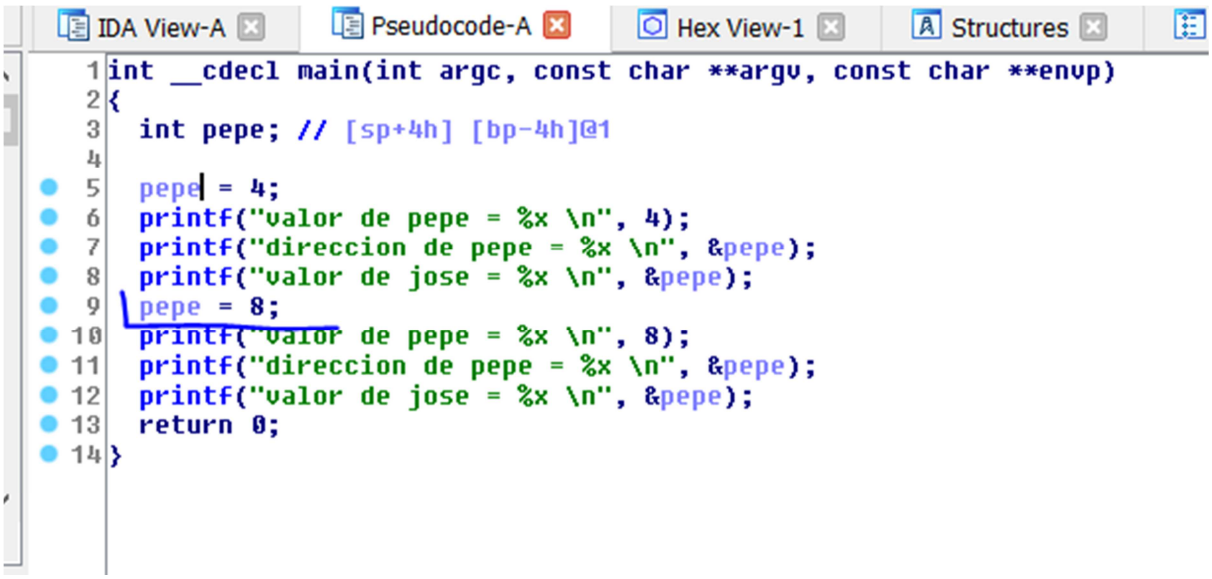
We see as we said that IDA shows that address as:

OFFSET DWORD 0xB1Fdd4

We had said that OFFSET in IDA meant a memory address and the DWORD that is next to it means that it points to a DWORD (int).



There, it passes the value of jose to ECX and saves the 8 in its content, where before was the 4. The content of jose is the value of pepe which it prints later.



We see that the pseudocode always works with pepe and does not show the jose pointer directly. Instead of using pointers, it modifies the pepe value directly which works but it is not the original code.

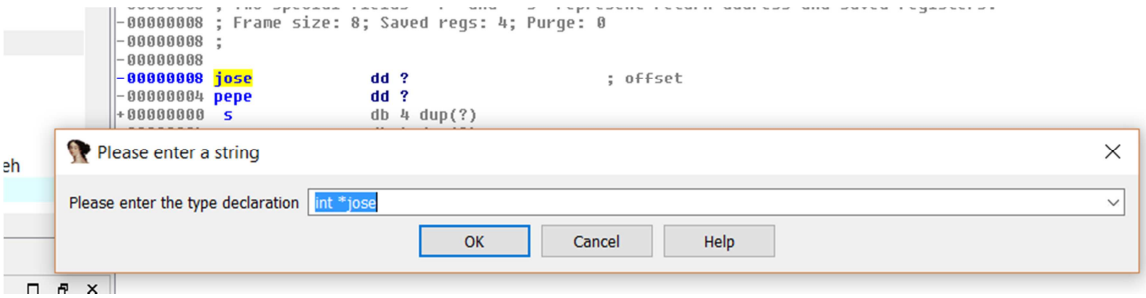
Obviously:

pepe =8

Equals to:

*jose=8

But it doesn't use the jose pointer.



Pressing Y allows us to manually define a variable, and I set it to be a pointer to an int.

```
-00000008 ; Two special fields " r" and " s" represent return address and
-00000008 ; Frame size: 8; Saved regs: 4; Purge: 0
-00000008 ;
-00000008
-00000008 jose            dd ?                    ; offset
-00000004 pepe            dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008 argc            dd ?
+0000000C argv            dd ?                    ; offset
+00000010 envp            dd ?                    ; offset
+00000014
+00000014 ; end of stack variables
```

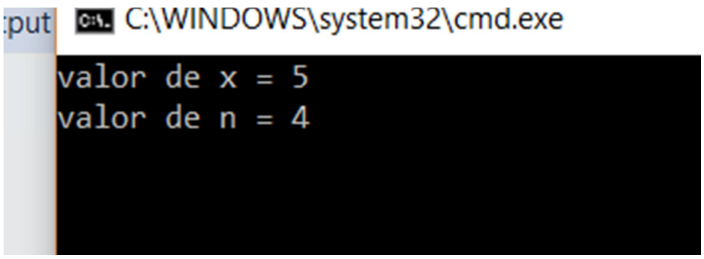
Even the pseudocode does not show changes, however the code it generates is an approximation and optimizes a lot to clarify, which cannot be taken as the revealed truth, hehe.

Let's see the following example that is a case where pointers are most useful, when you have to pass arguments to a function.

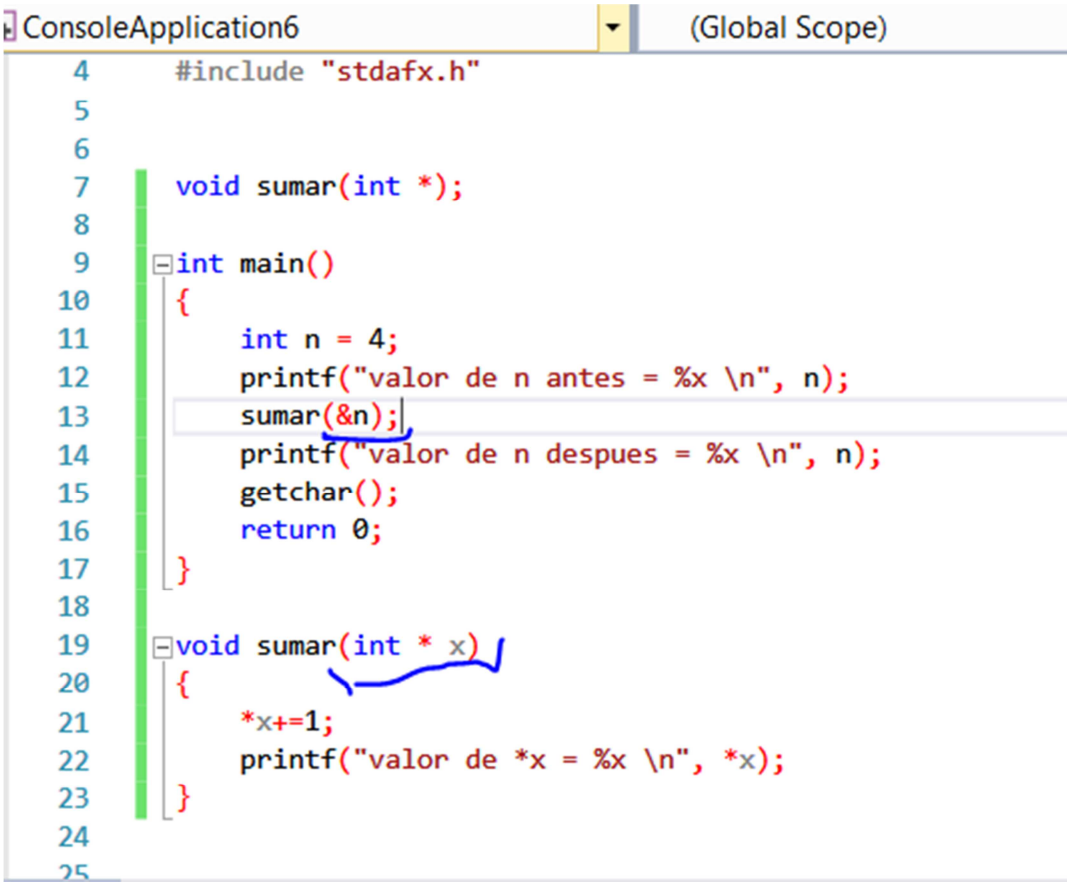
```
1   // ConsoleApplication6.cpp : Defines the entr
2   //
3
4    #include "stdafx.h"
5
6
7   void sumar(int);
8
9   int main()
10  {
11     int n = 4;
12     sumar(n);
13     printf("valor de n = %x \n", n);
14     getchar();
15     return 0;
16  }
17
18  void sumar(int x)
19  {
20     x+=1;
21     printf("valor de x = %x \n", x);
22  }
23
```

We see that we define an **n** integer that is worth 4 and we pass the value to an addition function, that function assigns the 4 to the **x** local variable and adds it 1, and prints 5, but when leaving **n** it is still worth 4, since the scope of validity of the n variable is the main function and the scope of x is the **sumar**

function, they are variables and different scope of validity and if you change one, it does not affect the other.



Now, how do I do if I want to work within functions and modify the value of n? Passing by value, as we did in the previous example is useless, but if we use pointers which serve for these cases, they allow us to store a memory address of a variable of main as in this case.



The **sumar** function is defined with a single x argument which is a pointer to an integer.

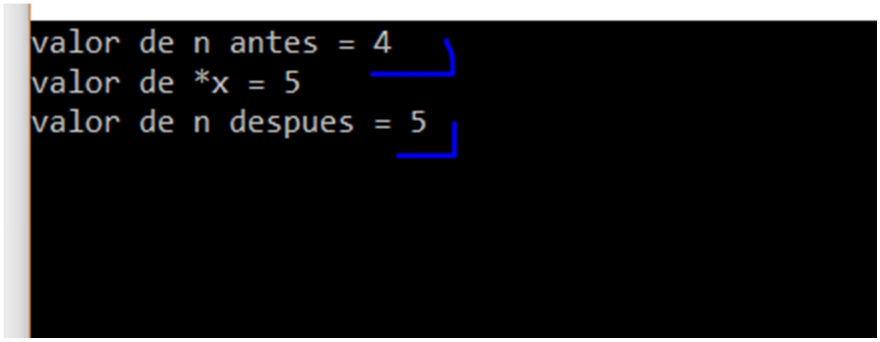
And when it is called, it is passed.

sumar(&n);

We know that a pointer stores memory addresses and in this case we pass the memory address of n that points to an integer, so everything is fine.

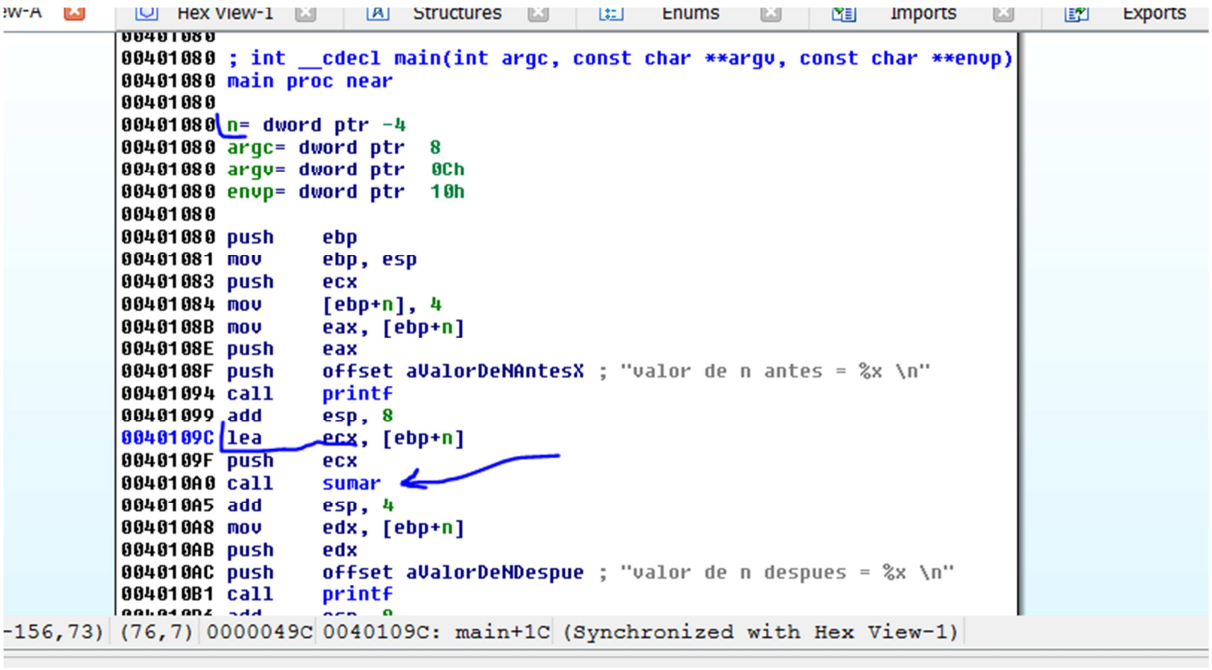
```
void sumar(int * x)
{
	*x+=1;
	printf("valor de *x = %x \n", *x);
}
```

We see that we increase the content of x, so as x had the memory address of n, we are affecting the value of n, which despite not being valid here, we are working with it, when using a pointer and when leaving its value, it will have changed.



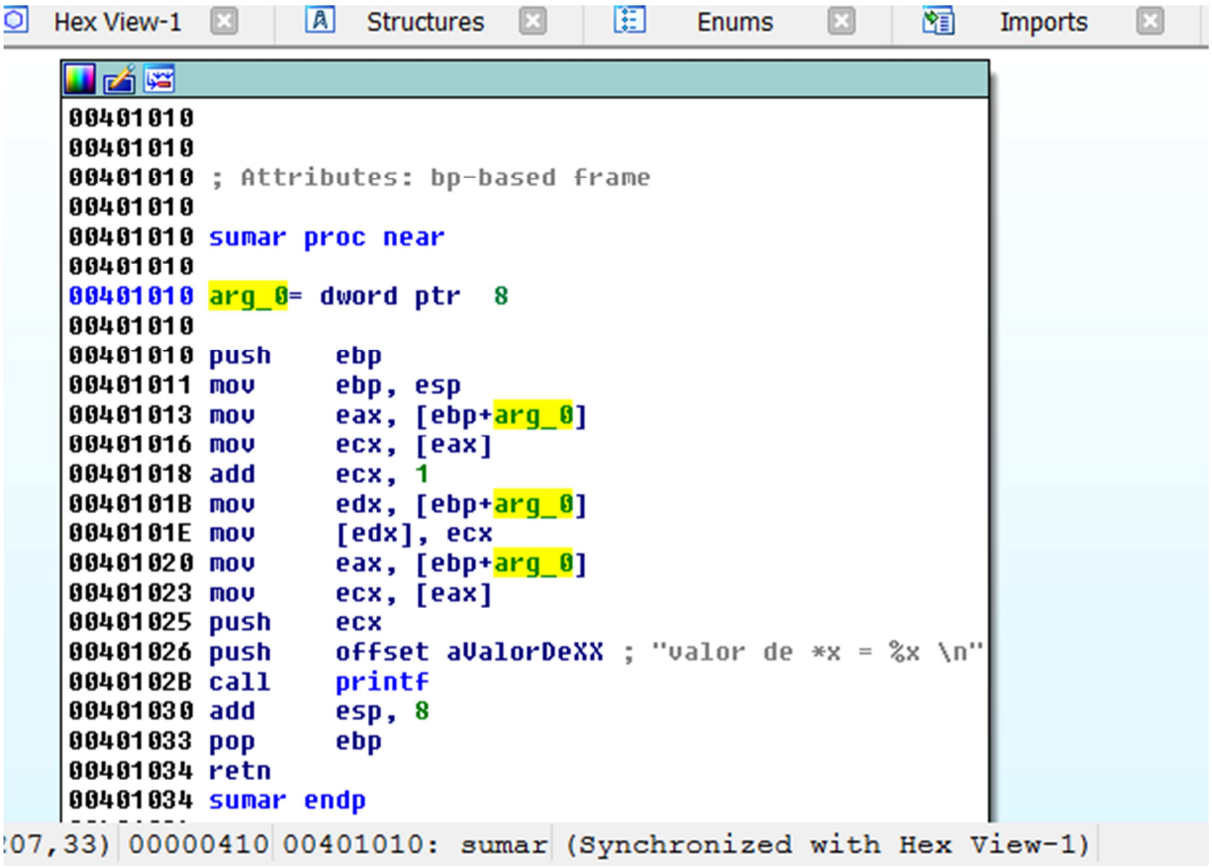
So n changed and we did not make any changes within **main**, nor is there any reference to any operation on it, because by passing the address of it as an argument and work inside the function with a pointer that received it, we access its value and we modify it equally.

Let's look it at IDA.

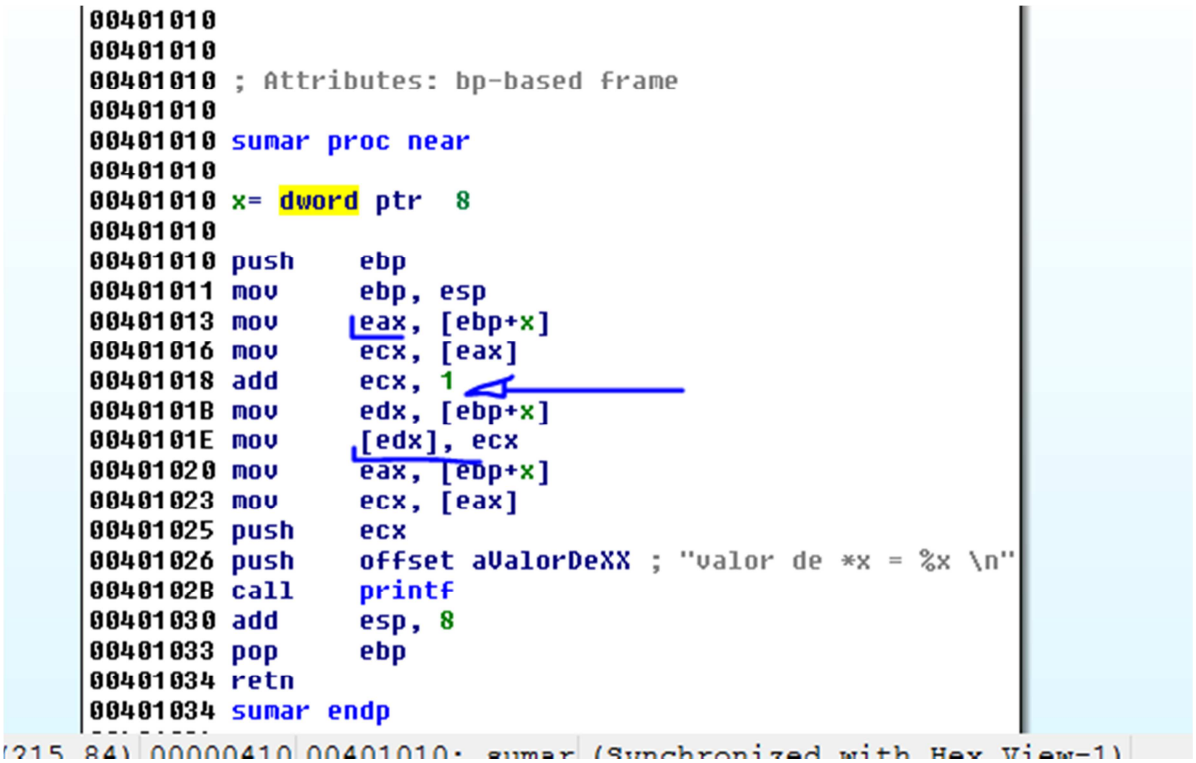


We see that it initializes n at startup when it is assigned the value 4, and then it passes the address of n as argument to the sumar function (and does not pass the value).

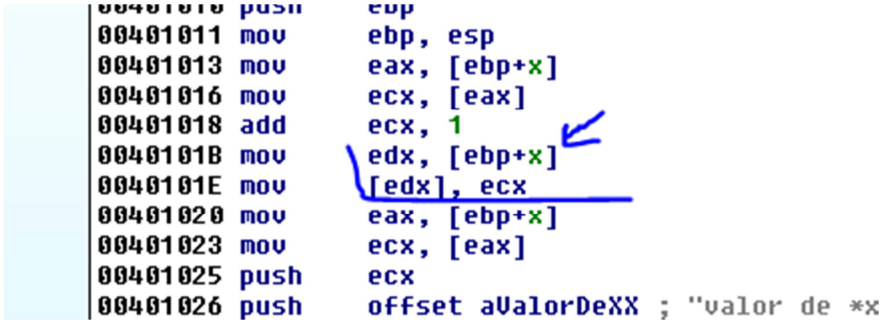
Let's see the function.



Arg_0 should be a pointer to an integer.



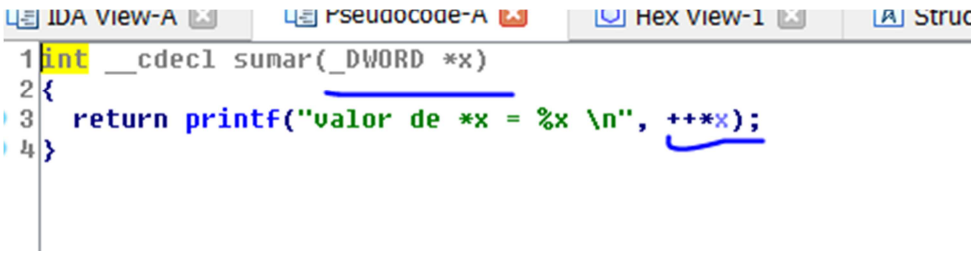
We see that it reads x there which is the memory address of n, and it passes it to EAX, then it reads the contents and increments it by 1 and saves it again in the EDX content.



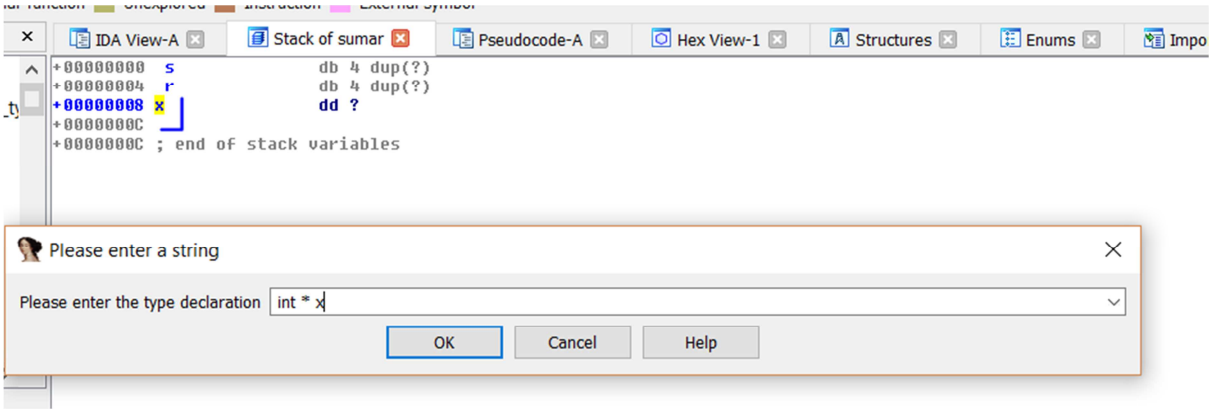
There the increased ECX value is stored in the EDX content which is the memory address of n.

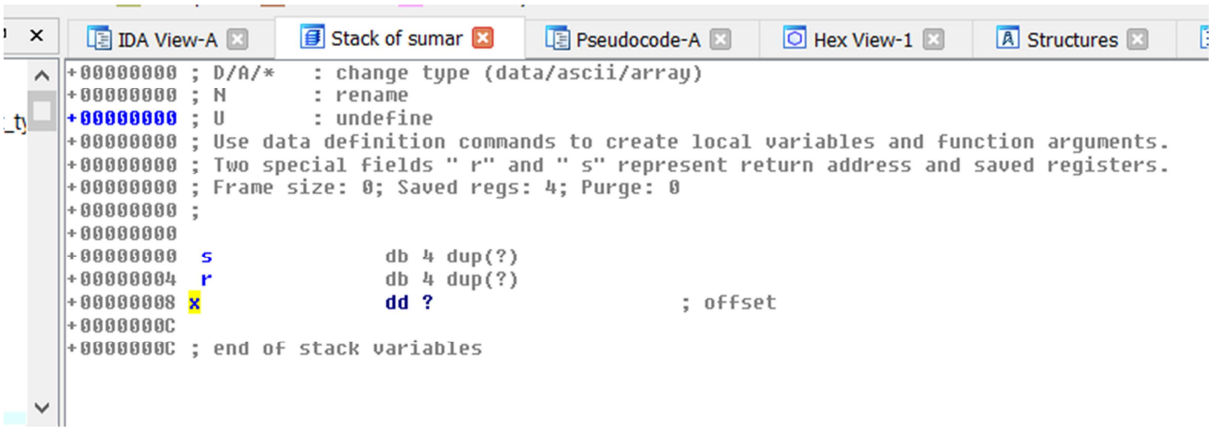
Therefore we see that passing addresses by means of pointers serves us to work with the values of variables of another scope, because if not, it could not change here.

Let's look at the code with f5.

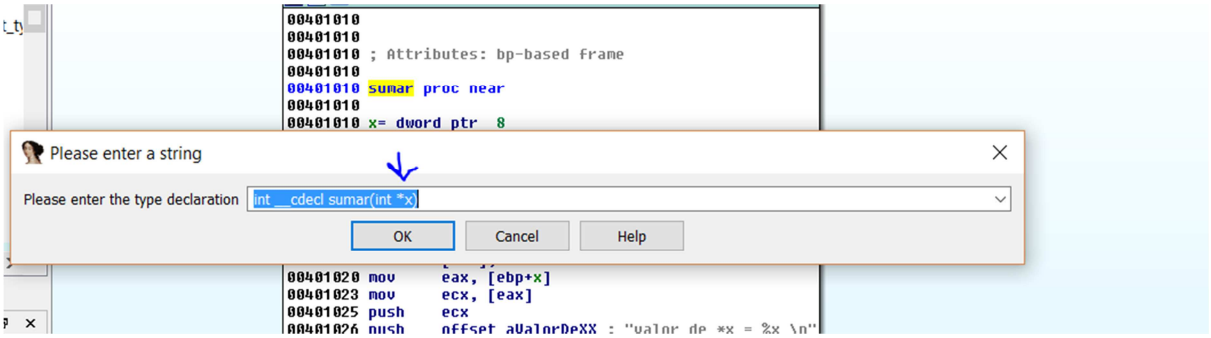


This time, it cannot optimize anything. X is a pointer to an integer (DWORD) there it shows. We can even change the data type of x in the disassembly.





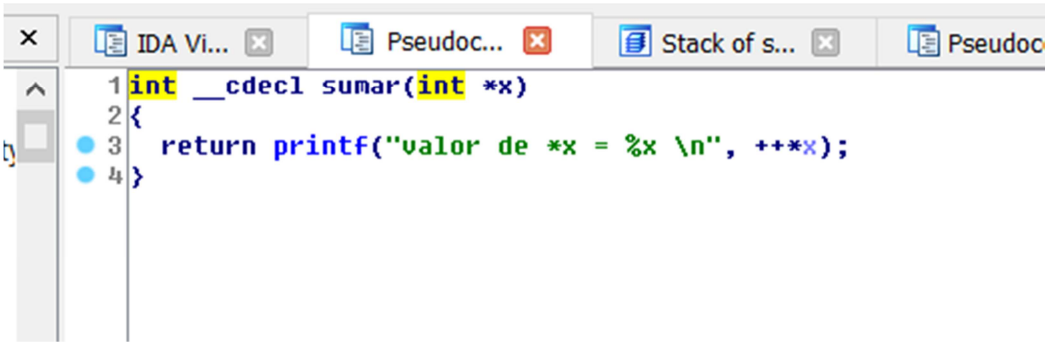
And let's see what it says when I do **set type** in the **sumar** function.



We see that the variable is now well declared as pointer.

Int * x

After that, pressing F5 is even better.

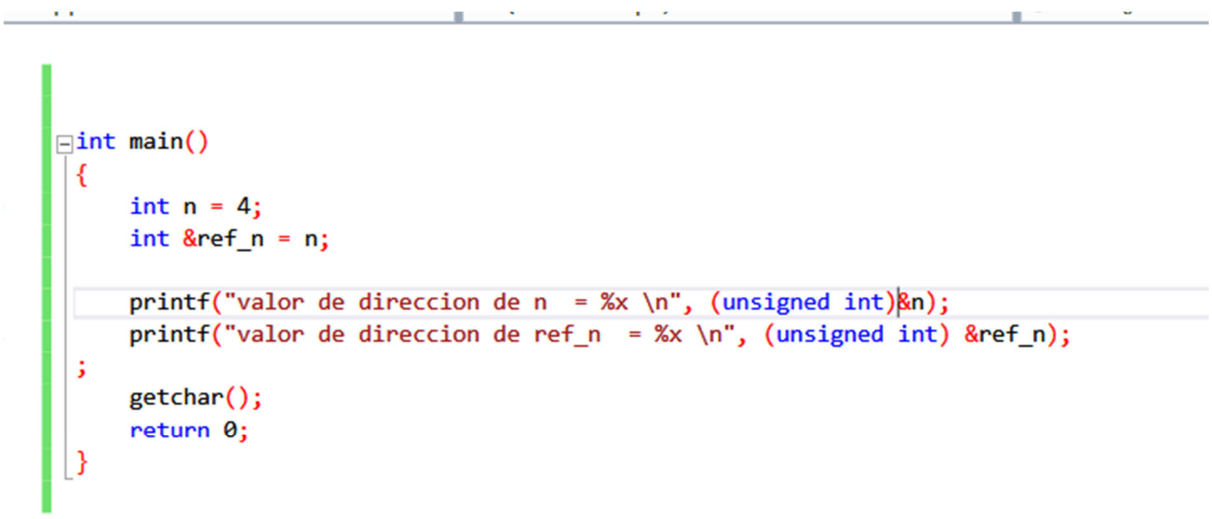


There is one more way to do this without using pointers, using what they call in C ++ references.

In addition to the pointers the C ++ language has another characteristic that are references, a reference is so to speak an alias or tag of a variable.

```
int n = 4;
int &ref_n = n;
```

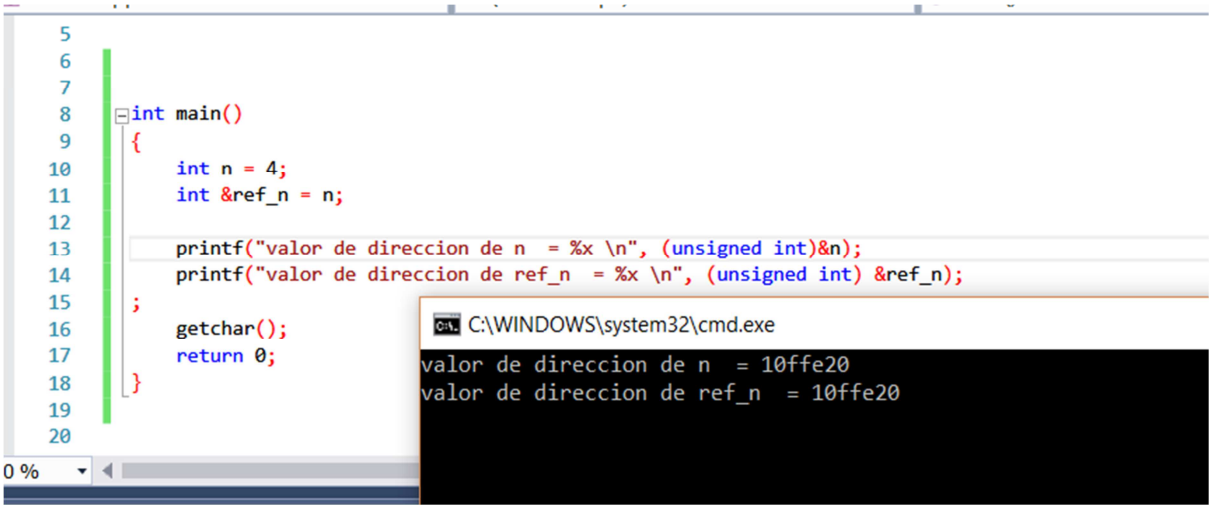
When I put & in front of a variable in the declaration, what I do is create an alias of the same variable that even shares the same memory address.



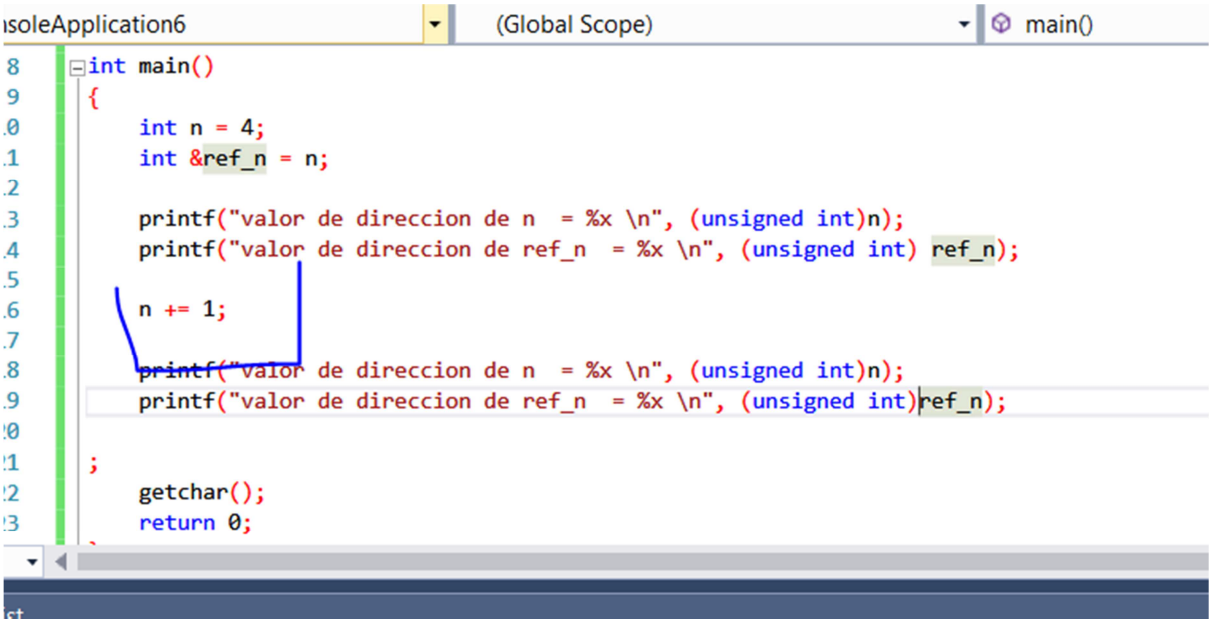
```
int main()
{
    int n = 4;
    int &ref_n = n;

    printf("valor de direccion de n  = %x \n", (unsigned int)&n);
    printf("valor de direccion de ref_n  = %x \n", (unsigned int) &ref_n);
;
    getchar();
    return 0;
}
```

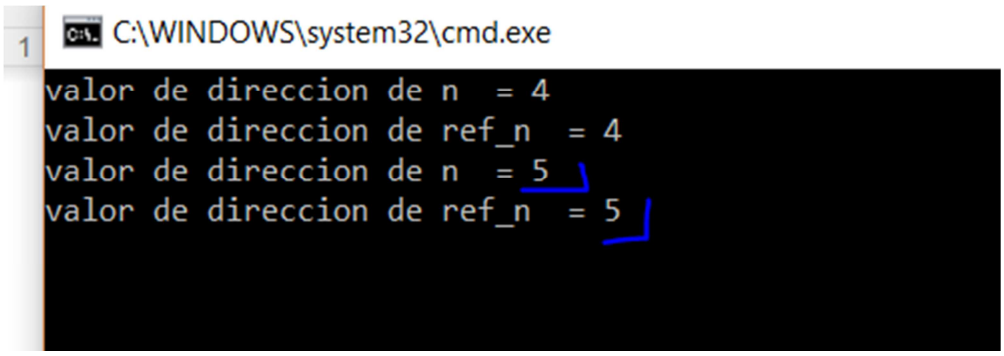
If I run it.



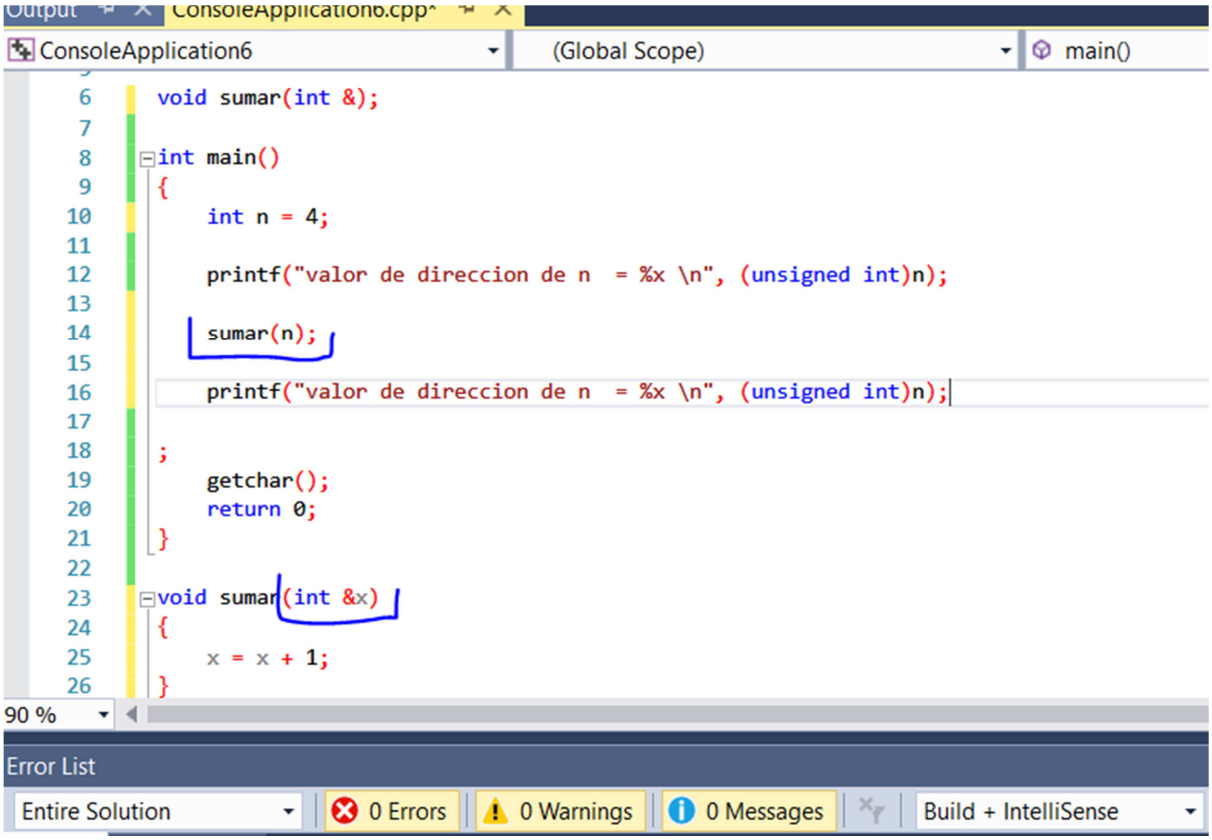
That way if I have two variables that share the same memory location, if I change a different one.



So when I change one, the other will change, too.



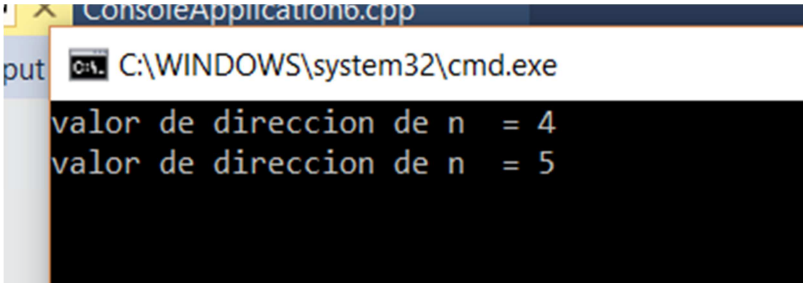
So that will serve me more than anything to write more comfortably the **sumar** function without using pointers.



We see that I passed the value of n but the function creates an alias of n that shares the same address, so modifying x is the same as modifying n.

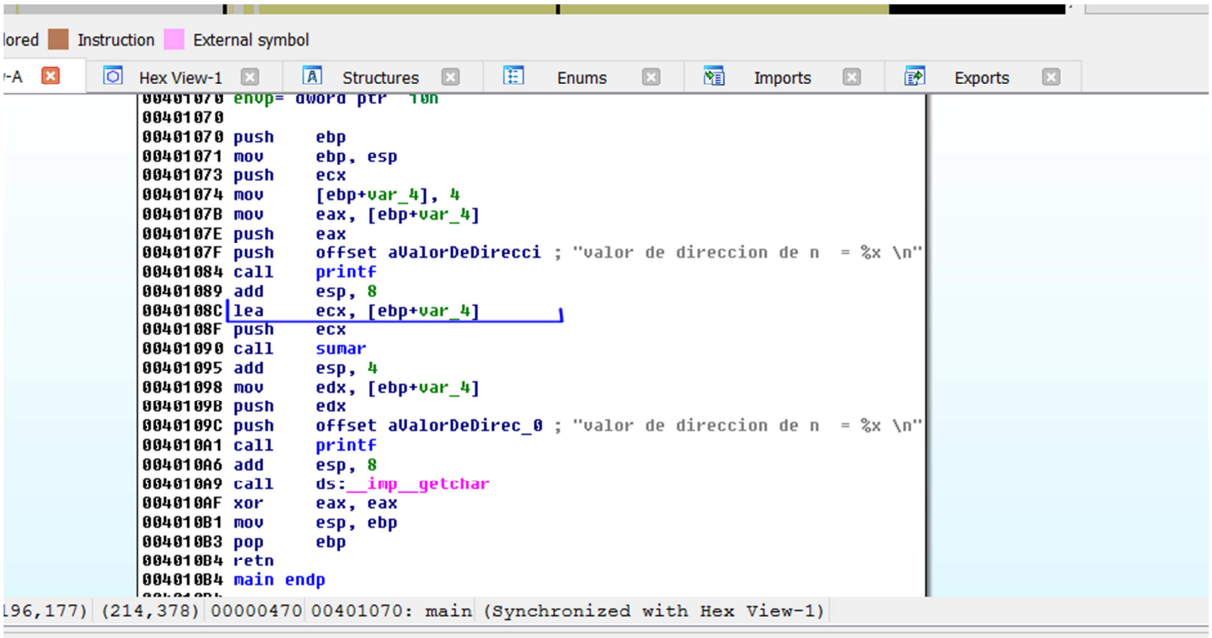
```
void sumar(int &x)
{
	x = x + 1;
}
```

We see that we do not have to deal with content or anything. Just change the value directly.

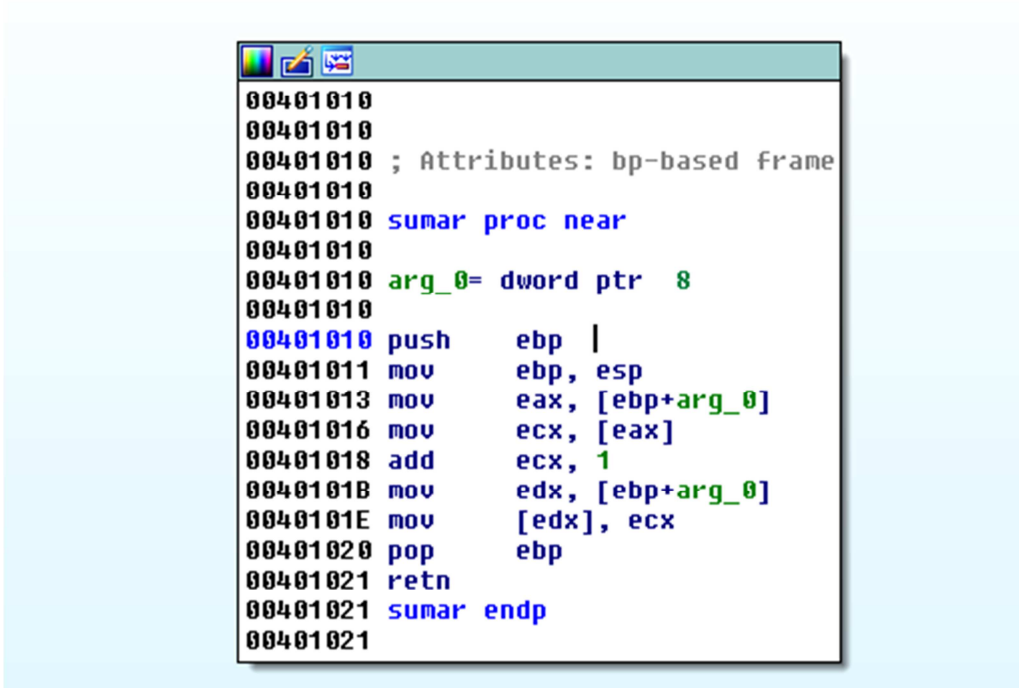


It works. Let's see it in IDA.

The joy vanished.



We see that all this is something to help write the source code more easily, but at low level still using pointers, we see that it does not pass the value of n but the address as before.



And inside the function is the same as before, a pointer. To which the content is increased.

So the alias or reference is very comfortable for writing code, but at a low level we only have to deal with pointers. :(

Ricardo Narvaja

Translated by: @IvinsonCLS