PART 41

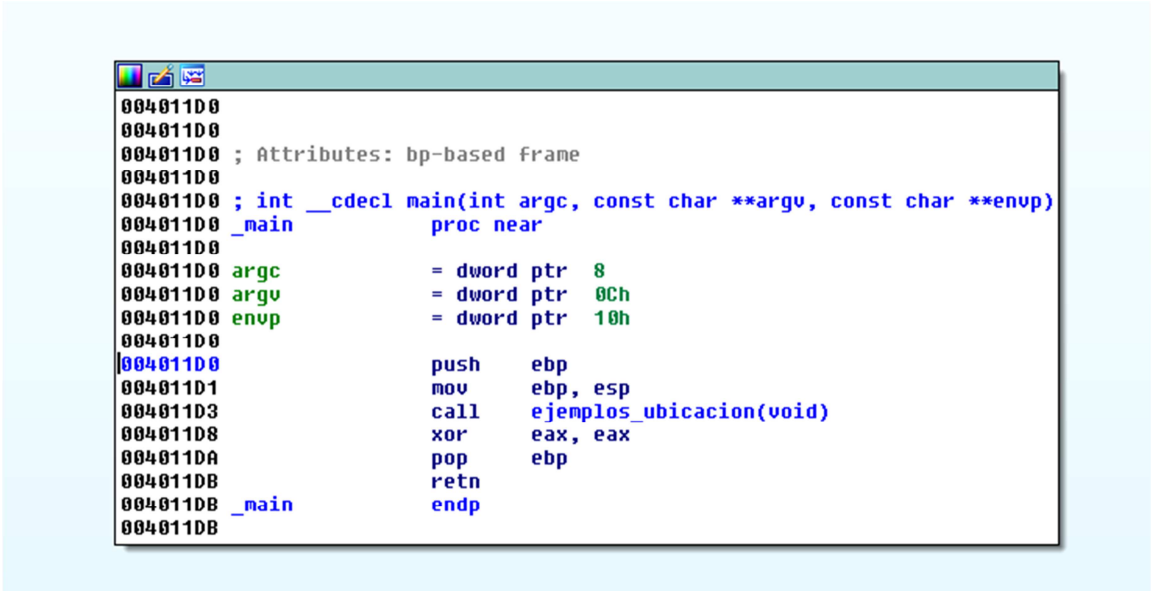
We'll continue practicing and seeing examples, in this case, it is a code that has different ways of handling and locating strings.

```
#include "stdafx.h"
#include <iostream>


char string[] = "Donde se ubicara?";
char string2[20];


void ejemplos_ubicacion()
{

    char mensaje_en_stack[]="hola reverser en stack";
    char mensaje_en_stack_sin_inicializar[100];
    char *mensaje_en_data="hola reverser en data";
    char *mensaje_en_heap;



    mensaje_en_heap = (char *)malloc(strlen(mensaje_en_data)+1);

    strcpy(mensaje_en_stack_sin_inicializar, "hola reverser en stack sin inicializar");

    strcpy(mensaje_en_heap,mensaje_en_data);

    memcpy(mensaje_en_heap+strlen("hola reverser en "),"heap",4);

    strcpy(string2,"Donde se ubicara?");

    printf("direccion mensaje_en_stack = 0x%x\n", mensaje_en_stack);
    printf("direccion mensaje_en_stack_sin_inicializar = 0x%x\n", mensaje_en_stack_sin_inicializar);
    printf("direccion mensaje_en_data = 0x%x\n", mensaje_en_data);
    printf("direccion mensaje_en_heap = 0x%x\n", mensaje_en_heap);
    printf("direccion string = 0x%x\n", string);
    printf("direccion string2 = 0x%x\n", string2);

    getchar();


}
```

There are some characters and then, it prints the addresses of each one to see where it is located.

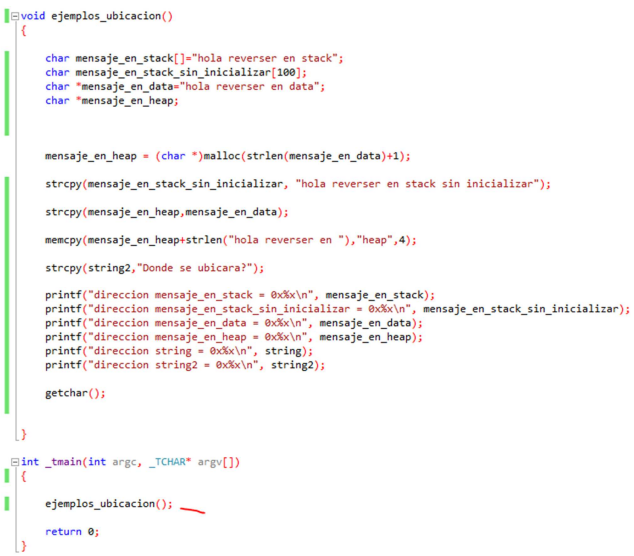
Here, we aren't looking for some vulnerability. We are just seeing addresses.

Let’s open the executable in the LOADER. We make it load the symbols. We’ll see the MAIN function.



```
004011D0
004011D0
004011D0 ; Attributes: bp-based frame
004011D0
004011D0 ; int __cdecl main(int argc, const char **argv, const char **envp)
004011D0 _main          proc near
004011D0
004011D0 argc           = dword ptr  8
004011D0 argv           = dword ptr  0Ch
004011D0 envp           = dword ptr  10h
004011D0
004011D0                push    ebp
004011D1                mov     ebp, esp
004011D3                call    ejemplos_ubicacion(void)
004011D8                xor     eax, eax
004011DA                pop     ebp
004011DB                retn
004011DB _main          endp
004011DB
```

There, we have the MAIN. It only has a CALL to the **ejemplos_ubicacion** function. Nothing else. Let’s activate the DEMANGLE NAMES –NAMES.



```
void ejemplos_ubicacion()
{

    char mensaje_en_stack[]="hola reverser en stack";
    char mensaje_en_stack_sin_inicializar[100];
    char *mensaje_en_data="hola reverser en data";
    char *mensaje_en_heap;


    mensaje_en_heap = (char *)malloc(strlen(mensaje_en_data)+1);

    strcpy(mensaje_en_stack_sin_inicializar, "hola reverser en stack sin inicializar");

    strcpy(mensaje_en_heap,mensaje_en_data);

    memcpy(mensaje_en_heap+strlen("hola reverser en "),"heap",4);

    strcpy(string2,"Donde se ubicara?");

    printf("direccion mensaje_en_stack = 0x%x\n", mensaje_en_stack);
    printf("direccion mensaje_en_stack_sin_inicializar = 0x%x\n", mensaje_en_stack_sin_inicializar);
    printf("direccion mensaje_en_data = 0x%x\n", mensaje_en_data);
    printf("direccion mensaje_en_heap = 0x%x\n", mensaje_en_heap);
    printf("direccion string = 0x%x\n", string);
    printf("direccion string2 = 0x%x\n", string2);

    getchar();


}
int _tmain(int argc, _TCHAR* argv[])
{

    ejemplos_ubicacion();

    return 0;
}
```

There, it is. Let’s see what happens inside the function. We see it doesn’t have args, only variables.

void __cdecl ejemplos_ubicacion()

```
00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; void __cdecl ejemplos_ubicacion()
00401010 void __cdecl ejemplos_ubicacion(void) proc near
00401010
00401010 mensaje_en_heap = dword ptr -88h
00401010 mensaje_en_data = dword ptr -84h
00401010 mensaje_en_stack_sin_inicializar= byte ptr -80h
00401010 mensaje_en_stack= byte ptr -1Ch
00401010 var_4           = dword ptr -4
00401010
00401010                 push    ebp
00401011                 mov     ebp, esp
00401013                 sub     esp, 88h
00401019                 mov     eax, ___security_cookie
0040101E                 xor     eax, ebp
00401020                 mov     [ebp+var_4], eax
00401023                 push    esi
00401024                 push    edi
00401025                 mov     ecx, 5
0040102A                 mov     esi, offset aHolaReverserEn ; "hola reverser en stack"
0040102F                 lea     edi, [ebp+mensaje_en_stack]
00401032                 rep movsd
00401034                 movsw
```

If it had args, they would be in the parenthesis. Besides, seeing the stack representation…

```
-0000008F                db ? ; undefined
-0000008E                db ? ; undefined
-0000008D                db ? ; undefined
-0000008C                db ? ; undefined
-0000008B                db ? ; undefined
-0000008A                db ? ; undefined
-00000089                db ? ; undefined
-00000088 mensaje_en_heap dd ?                       ; offset
-00000084 mensaje_en_data dd ?                       ; offset
-00000080 mensaje_en_stack_sin_inicializar db 100 dup(?)
-0000001C mensaje_en_stack db 23 dup(?)
-00000005                db ? ; undefined
-00000004 var_4          dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

If it had args, they should be below the Return Address (r) but there is nothing, only variables.

There, I see the CANARY saved in var_4 to start reversing.

```
00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; void __cdecl ejemplos_ubicacion()
00401010 void __cdecl ejemplos_ubicacion(void) proc near
00401010
00401010 mensaje_en_heap = dword ptr -88h
00401010 mensaje_en_data = dword ptr -84h
00401010 mensaje_en_stack_sin_inicializar= byte ptr -80h
00401010 mensaje_en_stack= byte ptr -1Ch
00401010 CANARY          = dword ptr -4
00401010
00401010                 push    ebp
00401011                 mov     ebp, esp
00401013                 sub     esp, 88h
00401019                 mov     eax, ___security_cookie
0040101E                 xor     eax, ebp
00401020                 mov     [ebp+CANARY], eax
00401023                 push    esi
00401024                 push    edi
```

Let's go back to the stack representation.

```
-0000008B                db ? ; undefined
-0000008A                db ? ; undefined
-00000089                db ? ; undefined
-00000088 mensaje_en_heap dd ?                       ; offset
-00000084 mensaje_en_data dd ?                       ; offset
-00000080 mensaje_en_stack_sin_inicializar db 100 dup(?)
-0000001C mensaje_en_stack db 23 dup(?)
-00000005                db ? ; undefined
-00000004 CANARY         dd ?
+00000000  s             db 4 dup(?)
+00000004  r             db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

We see that as **mensaje_en_stack** as **mensaje_en_stack_sin_inicializar** are stack buffers. One of 100 bytes and the other of 23 bytes.

There, we see the two cases.

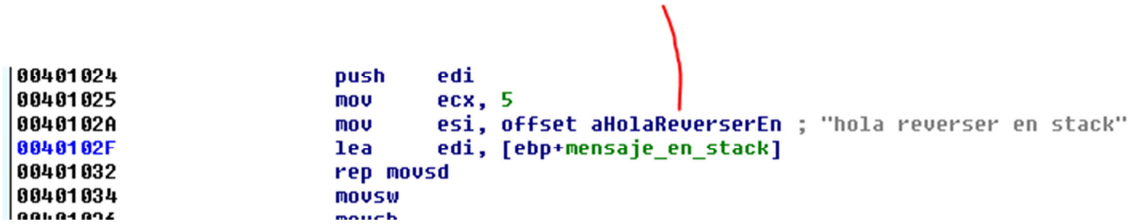
1) char mensaje_en_stack[]="hola reverser en stack"; #initialized

2) char mensaje_en_stack_sin_inicializar[100]; #not initialized

In case 2, it reserves 100 bytes because it doesn't know what will be saved there. It could be something entered by the user and not constant. The other reserves the space for the "hola reverser en stack" string that has a determined length.

It is 22 bytes long + the terminating null. 23 totally.

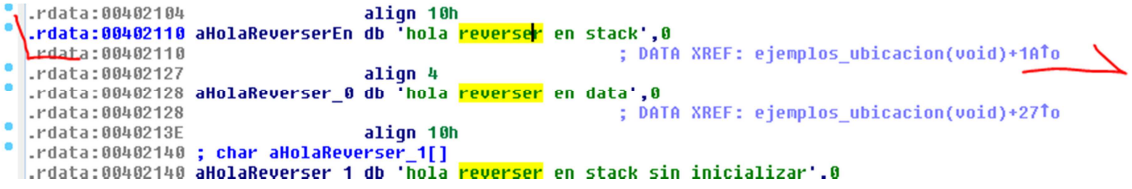
There, we see when it copies the string in the stack and initializes the variable. First, it gets the string address with OFFSET and moves it to ESI. Then, it will copy it to EDI that has the buffer address in the stack.



rdata

Declares an initialized data section that is readable but not writable. Microsoft compilers use this section to place constants in it.

In the **rdata** section, Visual Studio, when compiling, saves the constant data and the string is saved there. As the section is not writable, it won't change. Then, it copies it to the stack.



The LEA moves the buffer address in the stack and copies the string there with reps movs, initializing the variable.

In case 1, the variable was initialized while in case 2, it wasn't.

Logically, in case 2, this non-initialized buffer is there for something and the program will use and fill it some time.

```
00401063        push    offset aHolaReverser_1 ; "hola reverser en stack sin inicializar"
00401068        lea     ecx, [ebp+mensaje_en_stack_sin_inicializar]
0040106B        push    ecx ; char *
0040106C        call    _strcpy
```

It will do it there, it looks similar to case 1, but now, the program uses a Windows API to copy, it takes the OFFSET of the “hola reverser sin inicializar” string from the **rdata** section and copies it with strcpy.

strcpy(mensaje_en_stack_sin_inicializar, "hola reverser en stack sin inicializar");

Obviously, the compiler as in the stack variable initialization as in case 1, it won’t use Windows APIs. It will use instructions like reps movs while in case 2, it is program code that can use APIs.

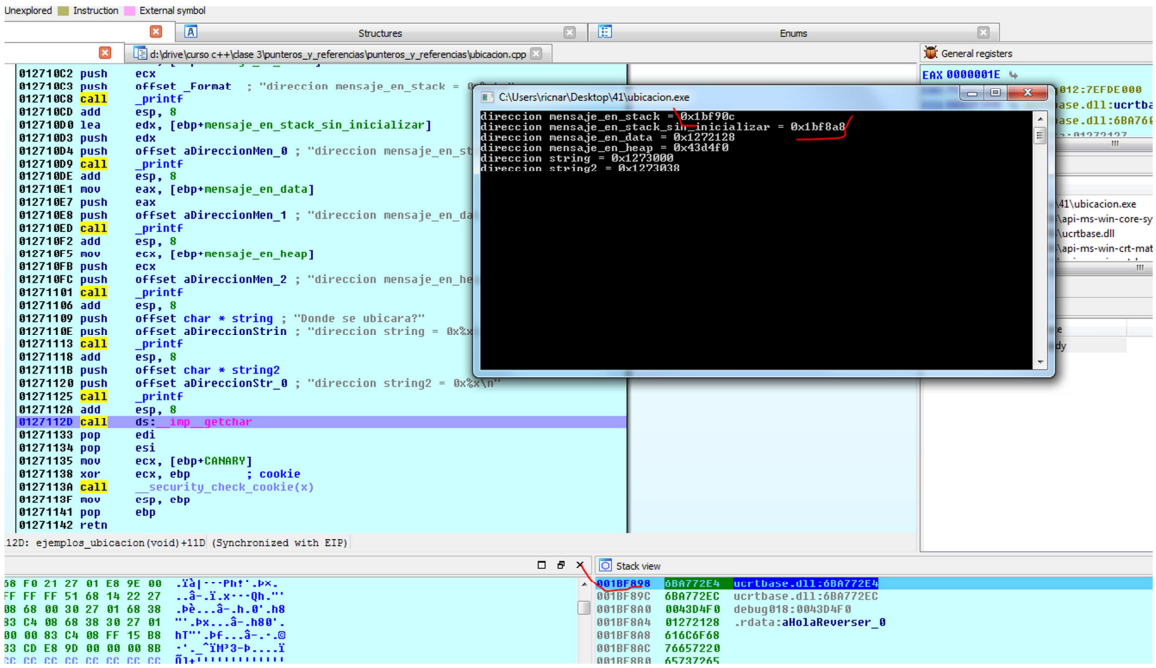
It is a coincidence that this case 2 has a determined string of a constant length, but it could be a string entered by the user that its length can change. There, we’ll have to check that it doesn’t overflow the buffer with a longer string.

Obviously, from these two strings we’ll get their addresses with LEA and they’ll be printed.

```
004010BC        add     esp, 8
004010BF        lea     ecx, [ebp+mensaje_en_stack]
004010C2        push    ecx
004010C3        push    offset _Format ; "direccion mensaje_en_stack = 0x%x\n"
004010C8        call    _printf
004010CD        add     esp, 8
004010D0        lea     edx, [ebp+mensaje_en_stack_sin_inicializar]
004010D3        push    edx
004010D4        push    offset aDireccionMen_0 ; "direccion mensaje_en_stack_sin_iniciali"...
004010D9        call    _printf
```

```
004010DE        add     esp, 8
004010E1        mov     eax, [ebp+mensaje_en_data]
004010E7        push    eax
004010E8        push    offset aDireccionMen_1 ; "direccion mensaje_en_data = 0x%x\n"
004010ED        call    _printf
004010F2        add     esp, 8
004010F5        mov     ecx, [ebp+mensaje_en_heap]
004010FB        push    ecx
004010FC        push    offset aDireccionMen_2 ; "direccion mensaje_en_heap = 0x%x\n"
00401101        call    _printf
00401106        add     esp, 8
00401109        push    offset char * string ; "Donde se ubicara?"
0040110E        push    offset aDireccionStrin ; "direccion string = 0x%x\n"
00401113        call    _printf
00401118        add     esp, 8
0040111B        push    offset char * string2
00401120        push    offset aDireccionStr_0 ; "direccion string2 = 0x%x\n"
00401125        call    _printf
0040112A        add     esp, 8
0040112D        call    ds:__imp__getchar
00401133        pop     edi
00401134        pop     esi
```

If I set a BP there, I choose the local debugger and run the program in debugger mode.



The printed addresses belong to the stack. I can see the strings in those addresses.

```
Stack[000023E0]:001BF90A db  9Dh ; Ø
Stack[000023E0]:001BF90B db  6Bh ; k
EIP
Stack[000023E0]:001BF90C db  68h ; h
Stack[000023E0]:001BF90D db  6Fh ; o
Stack[000023E0]:001BF90E db  6Ch ; l
Stack[000023E0]:001BF90F db  61h ; a
Stack[000023E0]:001BF910 db  20h
Stack[000023E0]:001BF911 db  72h ; r
Stack[000023E0]:001BF912 db  65h ; e
Stack[000023E0]:001BF913 db  76h ; v
Stack[000023E0]:001BF914 db  65h ; e
Stack[000023E0]:001BF915 db  72h ; r
Stack[000023E0]:001BF916 db  73h ; s
Stack[000023E0]:001BF917 db  65h ; e
Stack[000023E0]:001BF918 db  72h ; r
Stack[000023E0]:001BF919 db  20h
Stack[000023E0]:001BF91A db  65h ; e
Stack[000023E0]:001BF91B db  6Eh ; n
Stack[000023E0]:001BF91C db  20h
Stack[000023E0]:001BF91D db  73h ; s
Stack[000023E0]:001BF91E db  74h ; t
Stack[000023E0]:001BF91F db  61h ; a
Stack[000023E0]:001BF920 db  63h ; c
Stack[000023E0]:001BF921 db  6Bh ; k
Stack[000023E0]:001BF922 db    0
Stack[000023E0]:001BF923 db    0
Stack[000023E0]:001BF924 db  63h ; c
Stack[000023E0]:001BF925 db 0F9h ; ¨
Stack[000023E0]:001BF926 db 0EAh ; Ü
Stack[000023E0]:001BF927 db  77h ; w
Stack[000023E0]:001BF928 db  30h ; 0
```

If I press A to convert it in ASCII string…

```
Stack[000023E0]:001BF909 db  75h ; u
Stack[000023E0]:001BF90A db  9Dh ; Ø
Stack[000023E0]:001BF90B db  6Bh ; k
Stack[000023E0]:001BF90C aHolaReverser_3 db 'hola reverser en stack',0
Stack[000023E0]:001BF923 db    0
Stack[000023E0]:001BF924 db  63h ; c
Stack[000023E0]:001BF925 db 0F9h ; ¨
Stack[000023E0]:001BF926 db 0EAh ; Ü
Stack[000023E0]:001BF927 db  77h ; w
Stack[000023E0]:001BF928 db  30h ; 0
```

And the other one.

IDA View-EIP | d:\drive\curso c++\clase 3\punteros_y_referencias\punteros

```
Stack[000023E0]:001BF8A4 db  28h ; (
Stack[000023E0]:001BF8A5 db  21h ; !
Stack[000023E0]:001BF8A6 db  27h ; '
Stack[000023E0]:001BF8A7 db    1
Stack[000023E0]:001BF8A8 aHolaReverser_4 db 'hola reverser en stack sin inicializar',0
Stack[000023E0]:001BF8CF db 0D7h ; Î
Stack[000023E0]:001BF8D0 db    1
Stack[000023E0]:001BF8D1 db    0
Stack[000023E0]:001BF8D2 db    0
Stack[000023E0]:001BF8D3 db    0
```

We are OK until now. Let's see the other strings. Let's stop the debugger and go back to the loader.

```
00000090 ;
00000090
00000090                db ? ; undefined
0000008F                db ? ; undefined
0000008E                db ? ; undefined
0000008D                db ? ; undefined
0000008C                db ? ; undefined
0000008B                db ? ; undefined
0000008A                db ? ; undefined
00000089                db ? ; undefined
00000088 mensaje_en_heap dd ?                    ; offset
00000084 mensaje_en_data dd ?                    ; offset
00000080 mensaje_en_stack_sin_inicializar db 100 dup(?)
0000001C mensaje_en_stack db 23 dup(?)
00000005                db ? ; undefined
00000004 CANARY         dd ?
00000000  s             db 4 dup(?)
00000004  r             db 4 dup(?)
00000008
00000008 ; end of stack variables
```

The other two variables are pointers (offset) and just need 4 bytes each one (dd)

Let's see the **mensaje_en_data** pointer first.

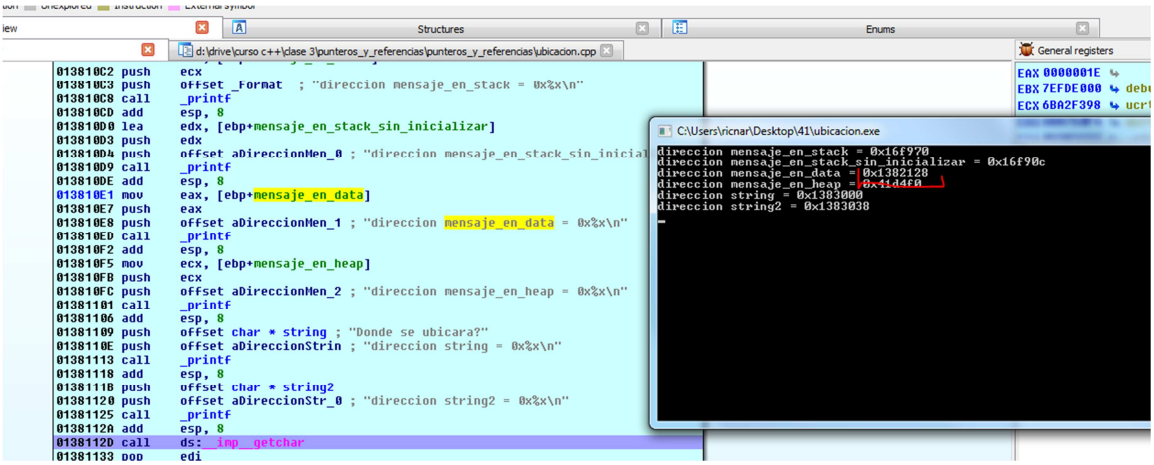
```
01271034                movsw
01271036                movsb
01271037                mov     [ebp+mensaje_en_data], offset aHolaReverser_0 ; "hola reverser en data"
0127103A                mov     eax, [ebp+mensaje_en_data]
0127103D                push    eax ; char *
0127103E                call    _strlen
01271043                add     esp, 4
01271046                add     eax, 1
01271049                push    eax ; Size
```

In this case, the variable is a pointer that saves the address it gets with OFFSET in the **mensaje_en_data** stack variable

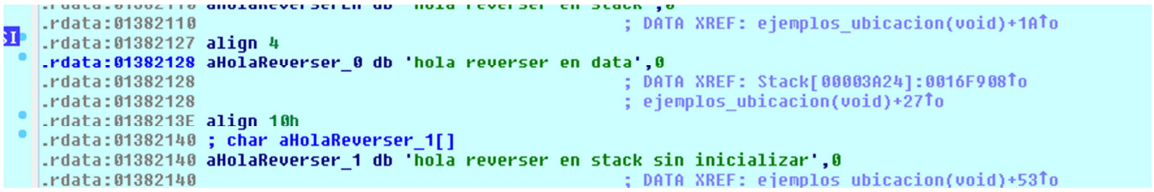
The string is located in **rdata** and saved there. What we handle in the stack is the pointer to it while in the previous ones the full string was copied in the stack filling a buffer in itself.

```
012710DE                add     esp, 8
012710E1                mov     eax, [ebp+mensaje_en_data]
012710E7                push    eax
012710E8                push    offset aDireccionMen_1 ; "direccion mensaje_en_data = 0x%x\n"
012710ED                call    _printf
```

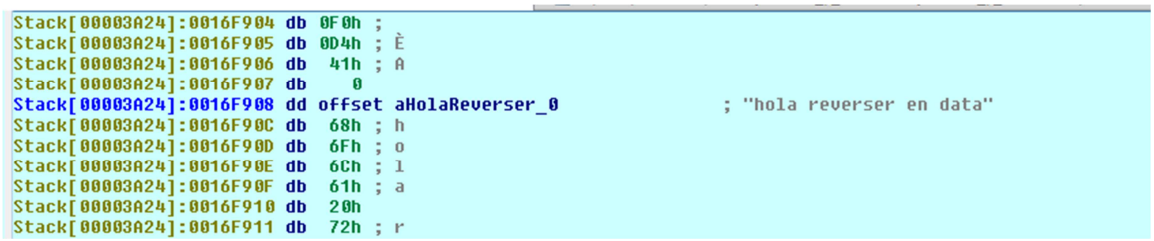
Now, it doesn't need a LEA to find the address because the variable value is a pointer and it is the address that is moved to EAX and pushed to print its value. Let's see if we debug it as before.



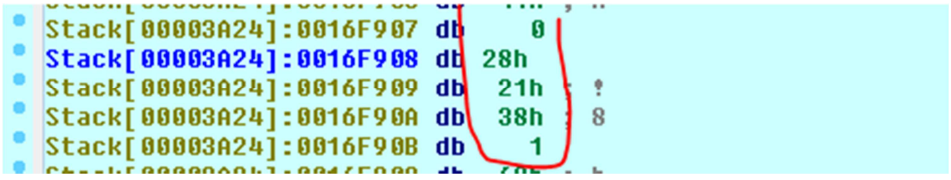
If I go to that **rdata** address, the string will be there.



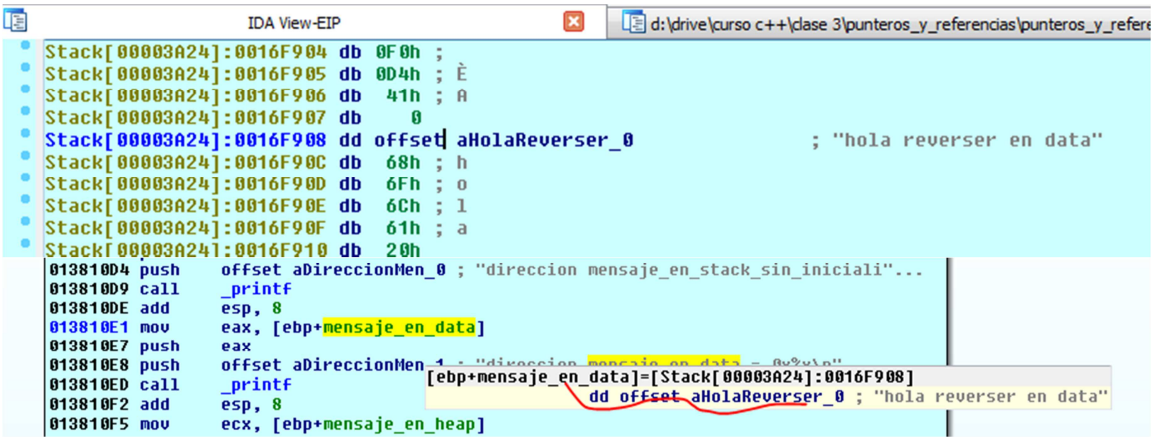
And, in the stack variable, that address (OFFSET) is saved.



There, the OFFSET is. If I want to see the numeric value, I press D.

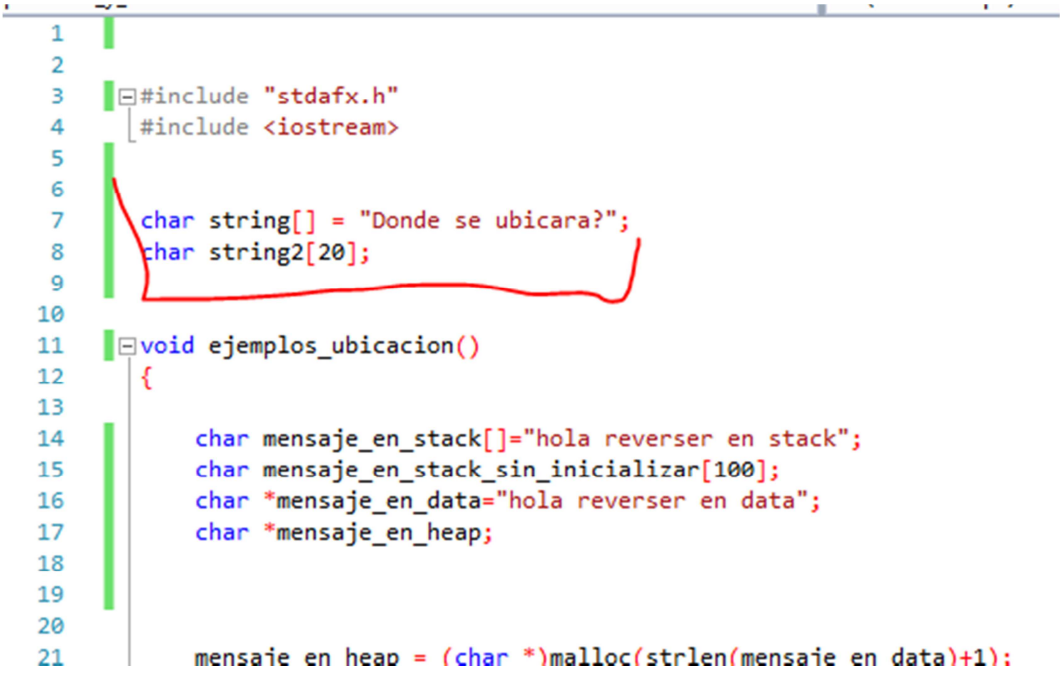


If I press D some times when it becomes a DWORD, it detects that it points to the string and changes to its OFFSET.



Hovering the mouse on there, we see that variable has the string address (OFFSET) saved because it is a pointer variable that saves addresses.

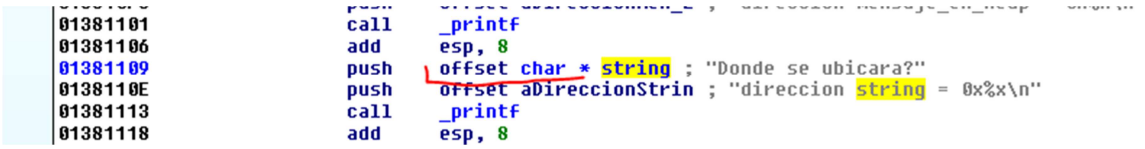
Let's leave the pending stack variable for the end and see the global variables. For those who don't know, the global variables are located in the source code outside the all functions, generally above all.



```
1
2
3   #include "stdafx.h"
4   #include <iostream>
5
6
7   char string[] = "Donde se ubicara?";
8   char string2[20];
9
10
11  void ejemplos_ubicacion()
12  {
13
14      char mensaje_en_stack[]="hola reverser en stack";
15      char mensaje_en_stack_sin_inicializar[100];
16      char *mensaje_en_data="hola reverser en data";
17      char *mensaje_en_heap;
18
19
20
21      mensaje_en_heap = (char *)malloc(strlen(mensaje_en_data)+1);
```

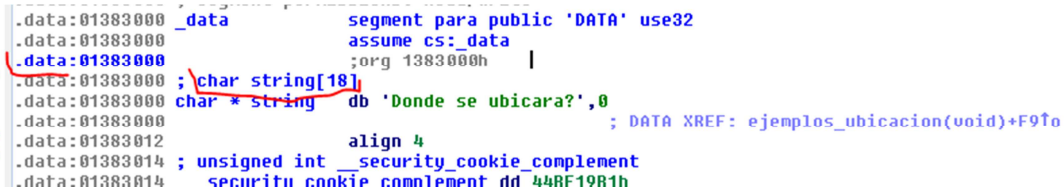
First, let's see the string variable that is initialized and it is global.

If I see the code, I realize that string is used here.

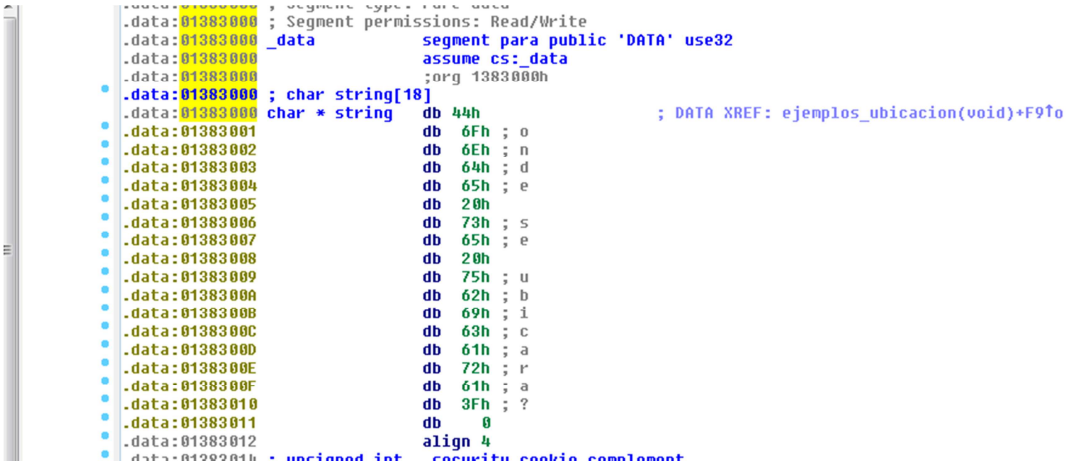


It pushes the string offset.

We see something confusing. We are in the data section that is used for the global variables.



We have the **char string [18]** buffer definition taken from the symbols. It knows that it is a string of that length that will be there. There, it is the "Donde se ubicara" saved string. If we press D, we see the bytes. If we press A, we go back as it was before.



Normally, when there is a second definition is because there is some reference to some API. From that API, it knows what kind of variable it needs and places it as definition too.

```
.data:01383000 ; Segment permissions: Read/Write
.data:01383000 _data           segment para public 'DATA' use32
.data:01383000                 assume cs:_data
.data:01383000                 ;org 1383000h
.data:01383000 ; char string[18]
.data:01383000 char * string   db 'Donde se ubicara?',0
.data:01383000                                         ; DATA XREF: ejemplos_ubicacion(void)+F9↑o
.data:01383012                 align 4
```

We see that there is a reference there.

```
01381113                call    _printf
01381118                add     esp, 8
0138111B                push    offset char * string2
01381120                push    offset aDireccionStr_0 ; "direccion string2 = 0x%x\n"
01381125                call    _printf
0138112A                add     esp, 8
0138112D                call    ds:__imp__getchar
```

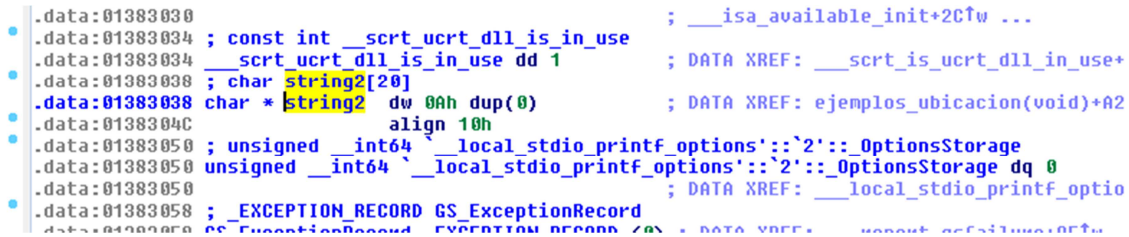
It is an arg to printf whose second arg is an address. If we right click, we'll see that it is the address the string is.

```
013810F5                mov     ecx, [ebp+var_88]
013810FB                push    ecx
013810FC                push    offset aDireccionMen_2 ; "direccion mensaje_en_heap = 0x%
01381101                call    _printf
01381106                add     esp, 8
01381109                push    1383000h
0138110E                push    offset aDireccionStrin ; "direccion string = 0x%x\n"
01381113                call    _printf
01381118                add     esp, 8
0138111B                push    1383038h
01381120                push    offset aDireccionStr_0 ; "direccion string2 = 0x%x\n"
01381125                call    _printf
0138112A                add     esp, 8
0138112D                call    ds:__imp__getchar
01381133                pop     edi
01381134                pop     esi
```

There, when right clicking, I directly replaced the offsets to the strings by the address it will print. In both, they are in the data section.

```
data:01383038 ; char string2[20]
data:01383038 char * string2  dw 0                    ; DATA XREF: ejemplos_ubicacion(void)+A2↑o
data:0138303A                 db    0
data:0138303B                 db    0
data:0138303C                 db    0
data:0138303D                 db    0
data:0138303E                 db    0
data:0138303F                 db    0
data:01383040                 db    0
data:01383041                 db    0
data:01383042                 db    0
data:01383043                 db    0
data:01383044                 db    0
data:01383045                 db    0
data:01383046                 db    0
data:01383047                 db    0
data:01383048                 db    0
data:01383049                 db    0
data:0138304A                 db    0
data:0138304B                 db    0
data:0138304C                 align 10h
data:01383050 ; unsigned __int64 `__local_stdio_printf_options'::`2'::_OptionsStorage
data:01383050 unsigned __int64 `__local_stdio_printf_options'::`2'::_OptionsStorage dq 0
```

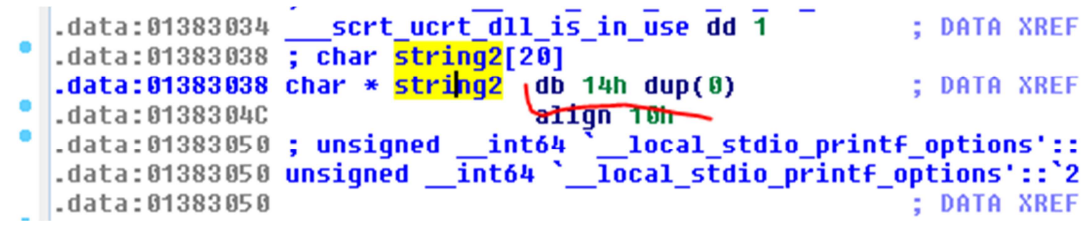
The only difference is that String2 is not initialized so with the A of ASCII, it won't reverted to the original state because it is full of 0s, and it is empty. So, right click- ARRAY will work.



char * string2 dw 0Ah dup(0)

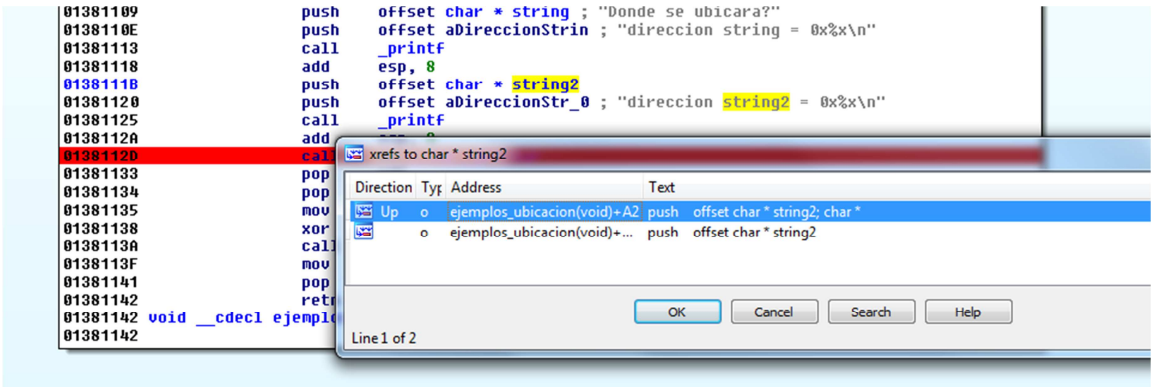
dup(0)

It means that the 0 is repeated dw(2 bytes) 0xA times or 20 decimal.



If I change it to bytes, it is more direct. It will be 0x14 or 20 byte full of 0s.

Let's see the differences where it is filled.



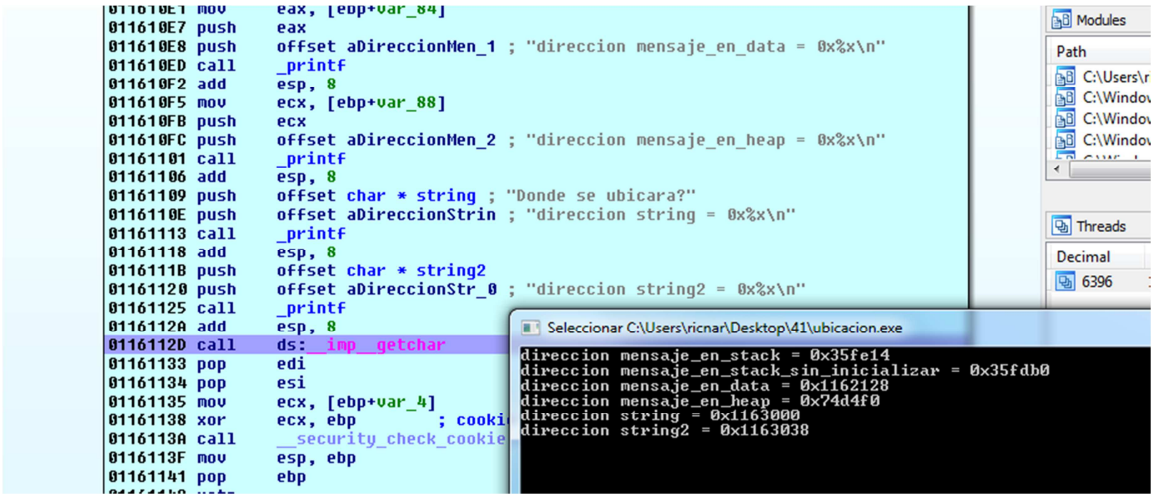
I converted the OFFSET that is passed as an arg to the original representation, with right click choosing **offset * char** and then with X, I see the references. The first one is where the buffer will be filled.

```
013810A4                push    eax ; void *
013810A5                call    _memcpy
013810AA                add     esp, 0Ch
013810AD                push    offset aDondeSeUbicara ; "Donde se ubicara?"
013810B2                push    offset char * string2 ; char *
013810B7                call    _strcpy
```

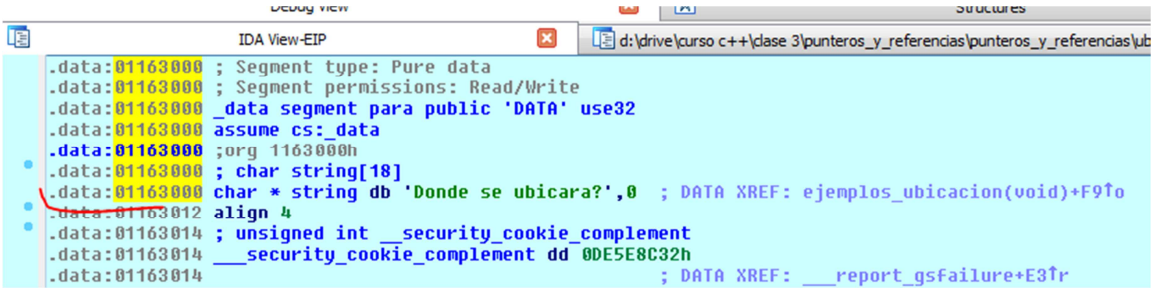
The “Donde se ubicara?” string that comes from **rdata** is copied there to the buffer in **data** that is writable and can be modified.

Obviously, in the data section, as it is a writable section that can have buffers, there can be overflows too, if the string length is not well calculated, stepping the global variables there affecting the program.

If we try it, we’ll see when debugging…



The string and string2 addresses that correspond to the **data** section.



```
.data:01163030 ___isa_enabled dd 7                     ; DATA XREF: ___isa_available_init+11↑w
.data:01163030                                         ; ___isa_available_init+2C↑w ...
.data:01163034 ; const int __scrt_ucrt_dll_is_in_use
.data:01163034 ___scrt_ucrt_dll_is_in_use dd 1         ; DATA XREF: ___scrt_is_ucrt_dll_in_use+2↑r
.data:01163038 ; char string2[20]
.data:01163038 char * string2 db 44h, 6Fh, 6Eh, 64h, 65h, 20h, 73h, 65h, 20h, 75h, 62h, 69h, 63h, 61h
.data:01163038                                         ; DATA XREF: ejemplos_ubicacion(void)+A2↑o
.data:01163038                                         ; ejemplos_ubicacion(void)+10B↑o
.data:01163038 db 72h, 61h, 3Fh, 0, 0, 0
```

We need to see the one in the heap. It was a pointer in the stack that saved the string address located in the heap.

```
00401036  movsb
00401037  mov     [ebp+mensaje_en_data], offset aHolaReverser_0 ; "hola reverser en data"
0040103E  mov     eax, [ebp+mensaje_en_data]
00401041  push    eax ; char *
00401047  call    _strlen
```

There, it saves the OFFSET or the “hola reverser en data” string address and then, it passes the address to strlen to get its length.

It add 1 to the result as in the source code.

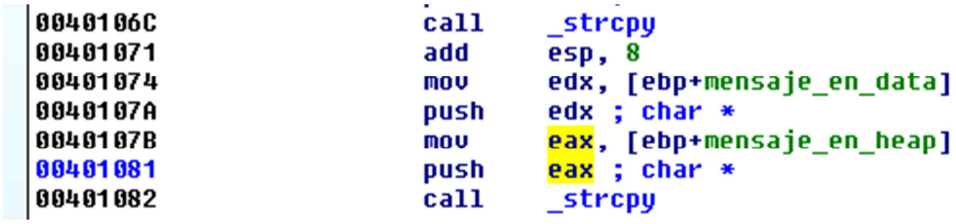
```
00401047  push    eax ; char *
00401048  call    _strlen
0040104D  add     esp, 4
00401050  add     eax, 1
00401053  push    eax ; Size
00401054  call    ds:__imp__malloc
```

```
mensaje_en_heap = (char *)malloc(strlen(mensaje_en_data)+1);
```

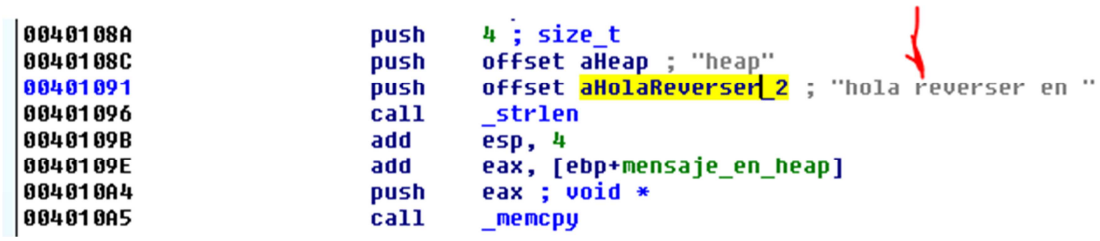
And it passes that size to malloc to reserve, in the heap, a dynamic buffer of the string length + 1.

The returned address will vary, but it will be a reserved zone with read and write permission where it will copy later.

There, it saves that heap address that points to that buffer, in the stack pointer variable called **mensaje_en_heap**.



Then, it copies the string pointed by **mensaje_en_data** to the buffer created in the heap.



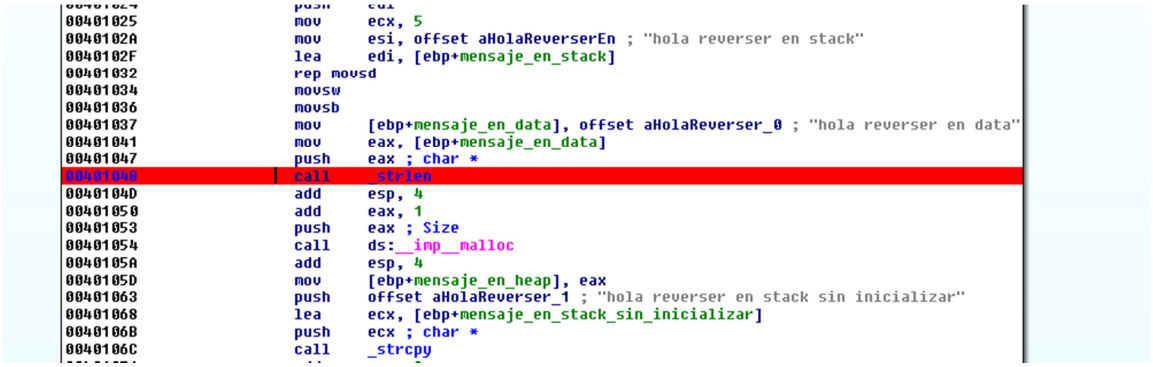
Then, it gets the length of the “hola reverser en “ string and adds it to the buffer start address in the heap. It points to replace the word **data** for **heap** because it does a 4-byte memcpy passing “heap” as **source** and the pointer to the word data as **destination** that is in the heap buffer.

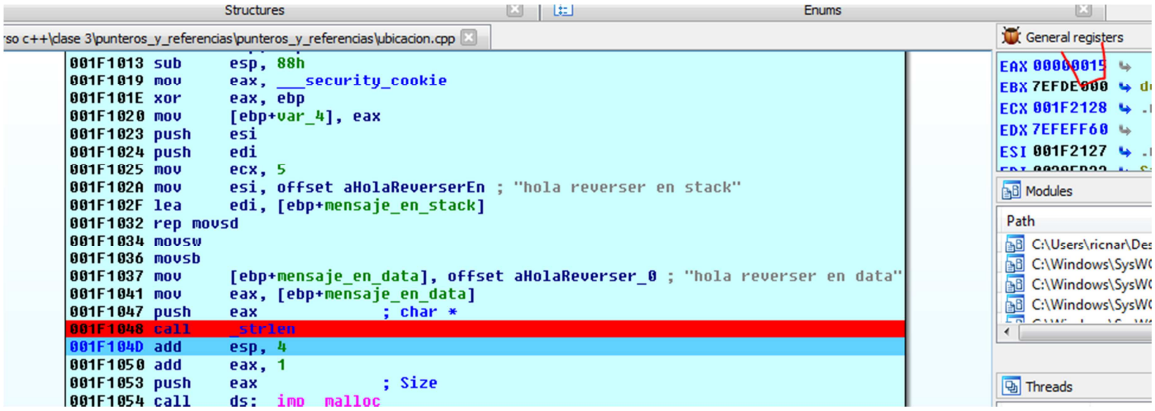
In the source code we see this:

memcpy(mensaje_en_heap+strlen("hola reverser en "),"heap",4);

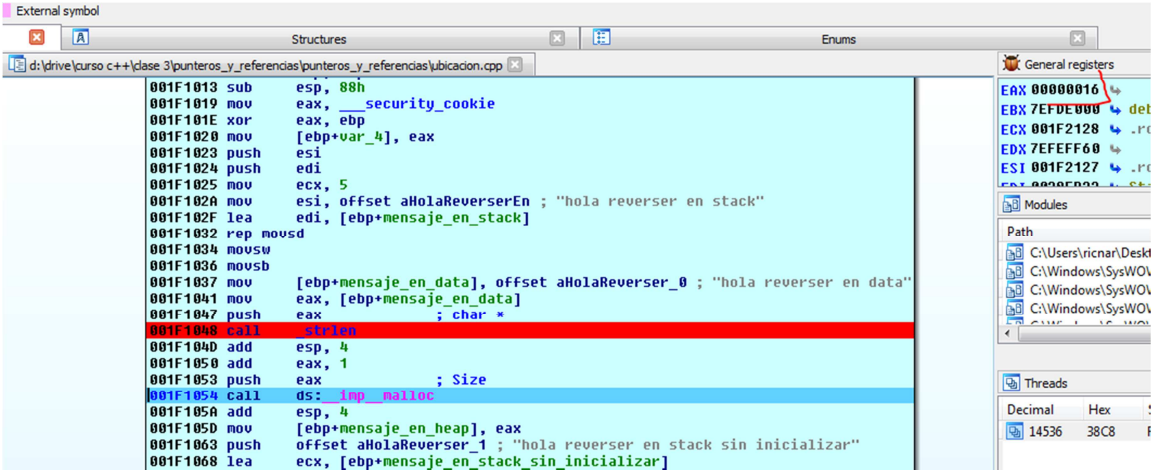
The **destination** is the **mensaje_en_heap** address + “hola reverser en “ string length. That will point to the word data which will be stepped with 4 bytes being the source the word “heap”.

Let’s debug to see if that is true.

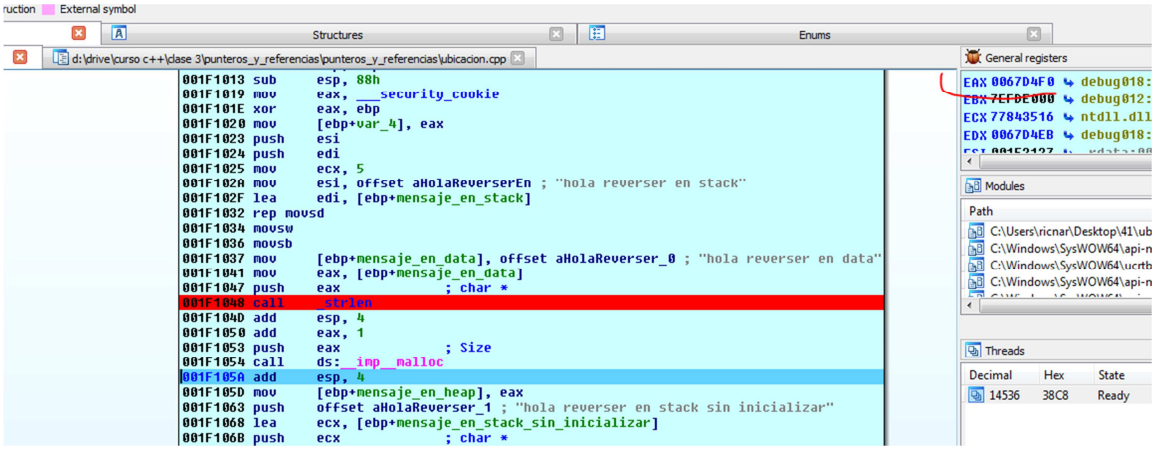




There, it got the “hola reverser en data” string length. It is 15 with the terminated null.



There, it adds 1 and it passed it to malloc to reserve 0x16 bytes.



There, we have the buffer start in the heap. In my case, it is 0x67d4f0.

```
debug018:0067D4ED db  3Ah ; :
debug018:0067D4EE db    0
debug018:0067D4EF db  1Ah
debug018:0067D4F0 db  0Dh
debug018:0067D4F1 db 0F0h ;
debug018:0067D4F2 db 0ADh ; ¡
debug018:0067D4F3 db 0BAh ; ¦
debug018:0067D4F4 db  0Dh
debug018:0067D4F5 db 0F0h ;
debug018:0067D4F6 db 0ADh ; ¡
debug018:0067D4F7 db 0BAh ; ¦
debug018:0067D4F8 db  0Dh
debug018:0067D4F9 db 0F0h ;
debug018:0067D4FA db 0ADh ; ¡
debug018:0067D4FB db 0BAh ; ¦
debug018:0067D4FC db  0Dh
debug018:0067D4FD db 0F0h ;
debug018:0067D4FE db 0ADh ; ¡
debug018:0067D4FF db 0BAh ; ¦
debug018:0067D500 db  0Dh
debug018:0067D501 db 0F0h ;
debug018:0067D502 db 0ADh ; ¡
debug018:0067D503 db 0BAh ; ¦
debug018:0067D504 db 0EEh ;
debug018:0067D505 db 0FEh ; ■
debug018:0067D506 db 0ABh ; ½
debug018:0067D507 db 0ABh ; ½
debug018:0067D508 db 0ABh ; ½
debug018:0067D509 db 0ABh ; ½
debug018:0067D50A db 0ABh ; ½
debug018:0067D50B db 0ABh ; ½
debug018:0067D50C db 0ABh ; ½
debug018:0067D50D db 0ABh ; ½
debug018:0067D50E db 0EEh ;
debug018:0067D50F db 0FEh ; ■
```

As the heap is in debug mode, when running from the debugger the 0x16 looks clear. It is 22 decimal if we convert it in dwords from the start.

```
debug018:0067D4F0 dd 0BAADF00Dh
debug018:0067D4F4 dd 0BAADF00Dh
debug018:0067D4F8 dd 0BAADF00Dh
debug018:0067D4FC dd 0BAADF00Dh
debug018:0067D500 dd 0BAADF00Dh
debug018:0067D504 dw 0FEEEh
debug018:0067D506 db 0ABh ; ½
```

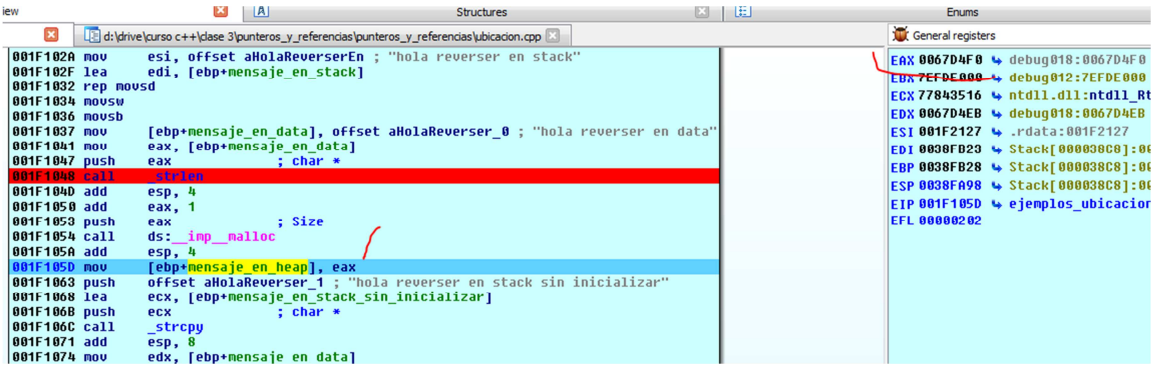
As it is in debug mode (if it isn't like this, it won't work) and there is nothing written yet, we see BAAD FOOD. ☺

They are 5 dwords or a 20-byte buffer + the 2 final bytes: 0xFEEE. I asked it 0x16 or 22 decimal.

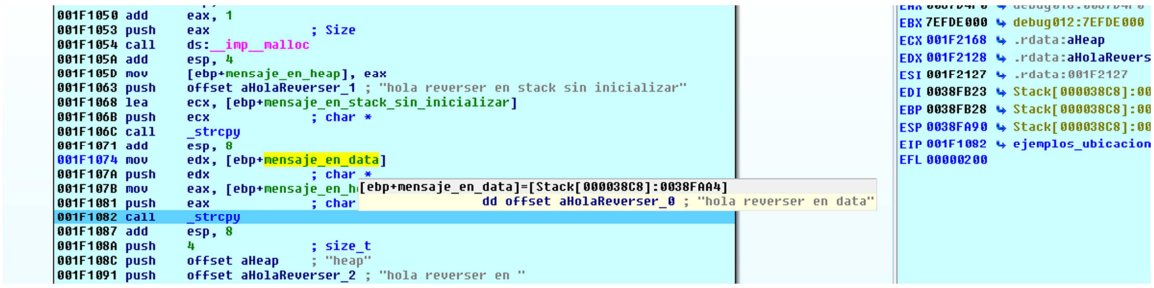
```
Python>0x16
22
```

We'll see more stuff about the heap. By now, that's the buffer full of **BAD FOOD**. ☺

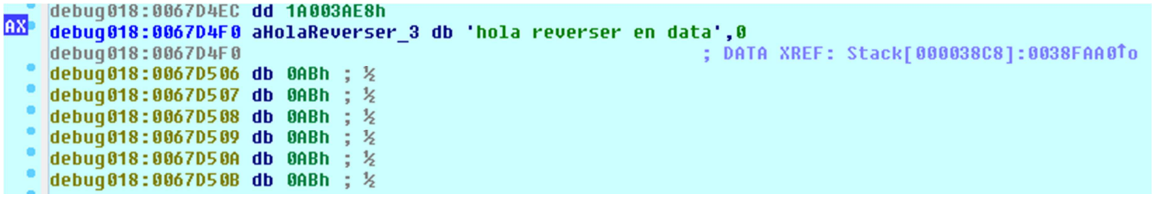
Then, it saves the address in the stack pointer variable.



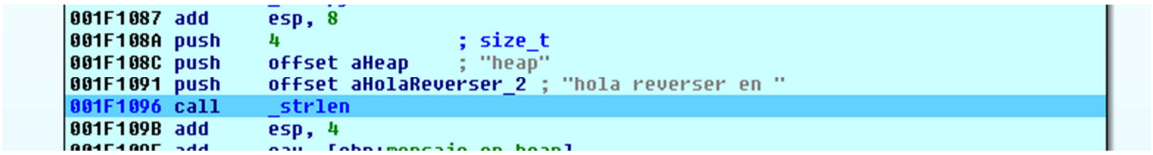
Then, in that strcpy, it will copy the hola reverser en data string in **data** in my baad food. ☺



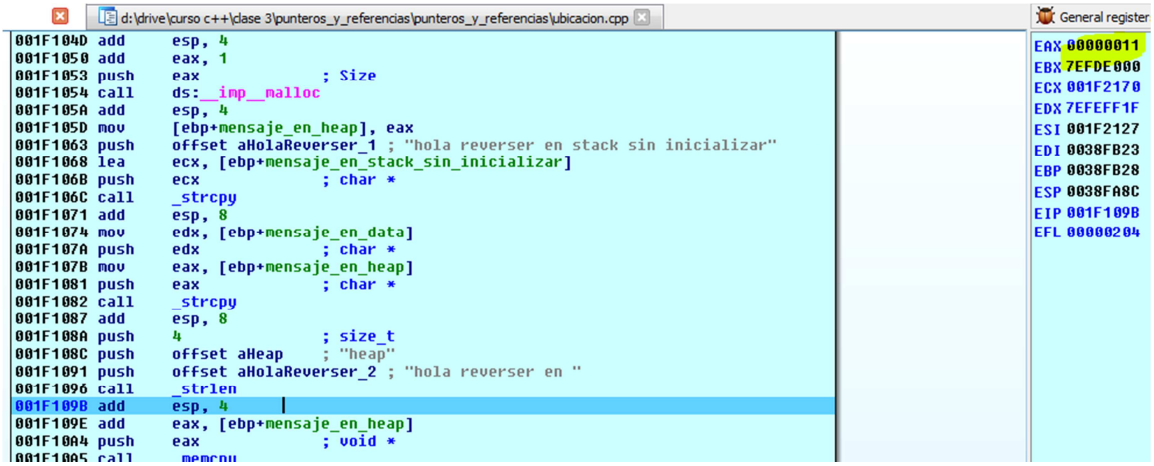
If I trace it with F8…



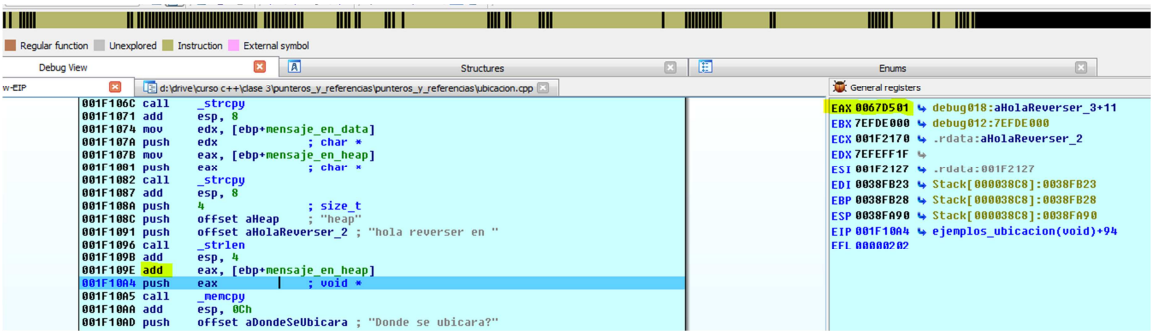
I see the copied bytes in the heap buffer. By pressing the letter A, I convert them into ASCII.



There, it will get the “hola reverser en ” length.

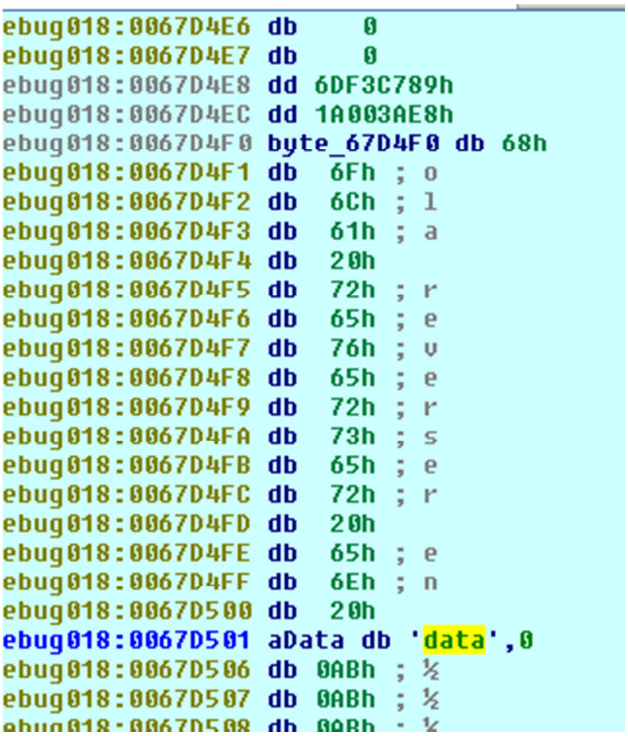


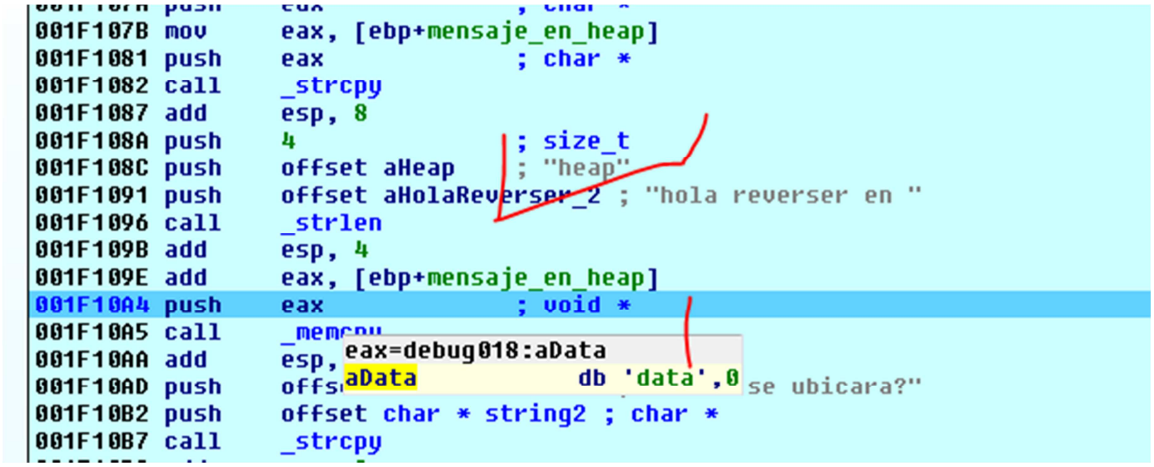
It gives 11, then it adds the address to the buffer start in the heap.



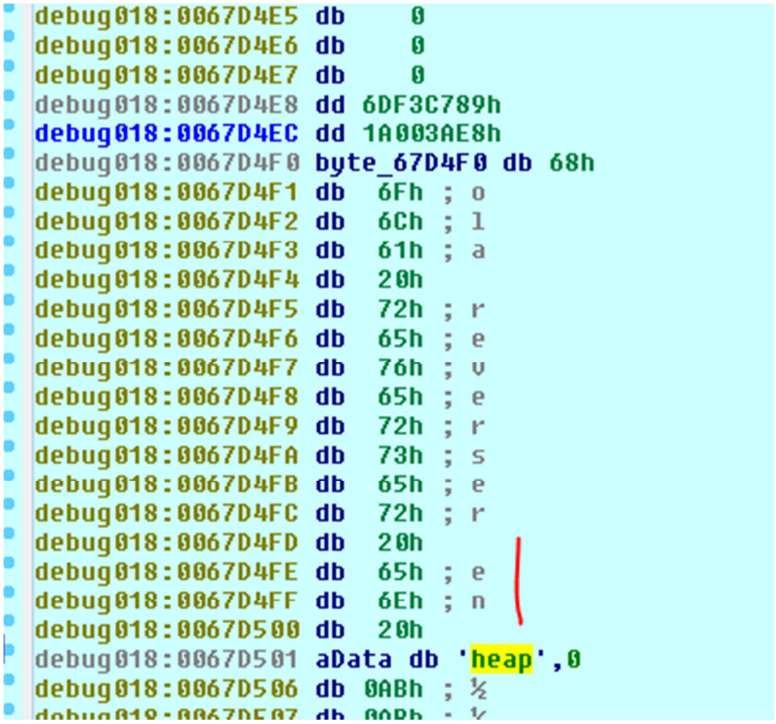
If I decompose the original string in bytes…

Then, I press A there.

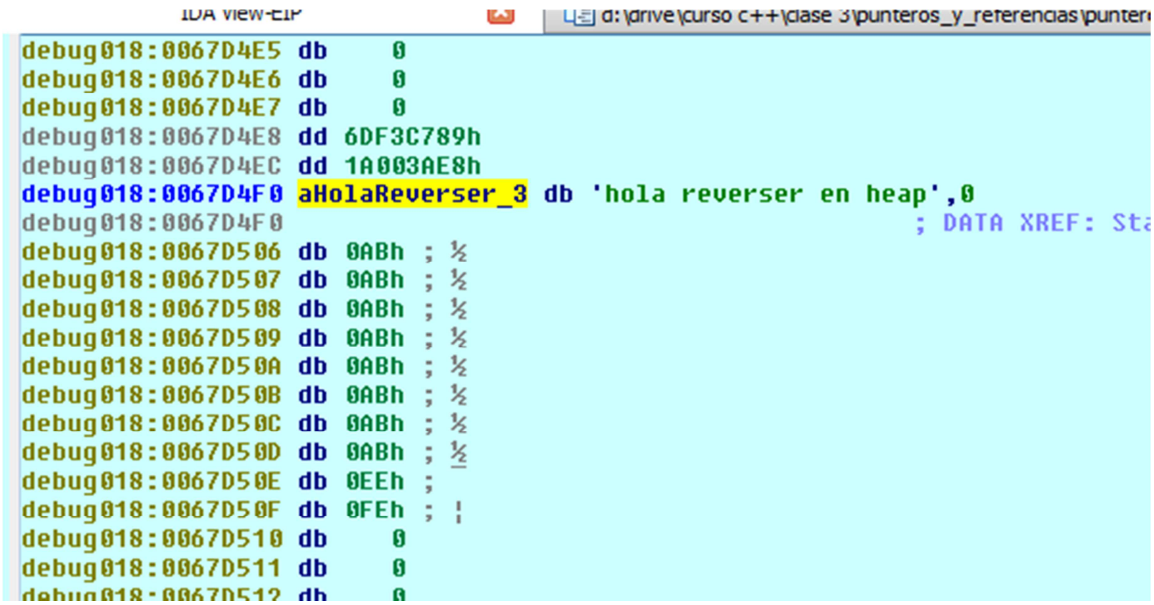




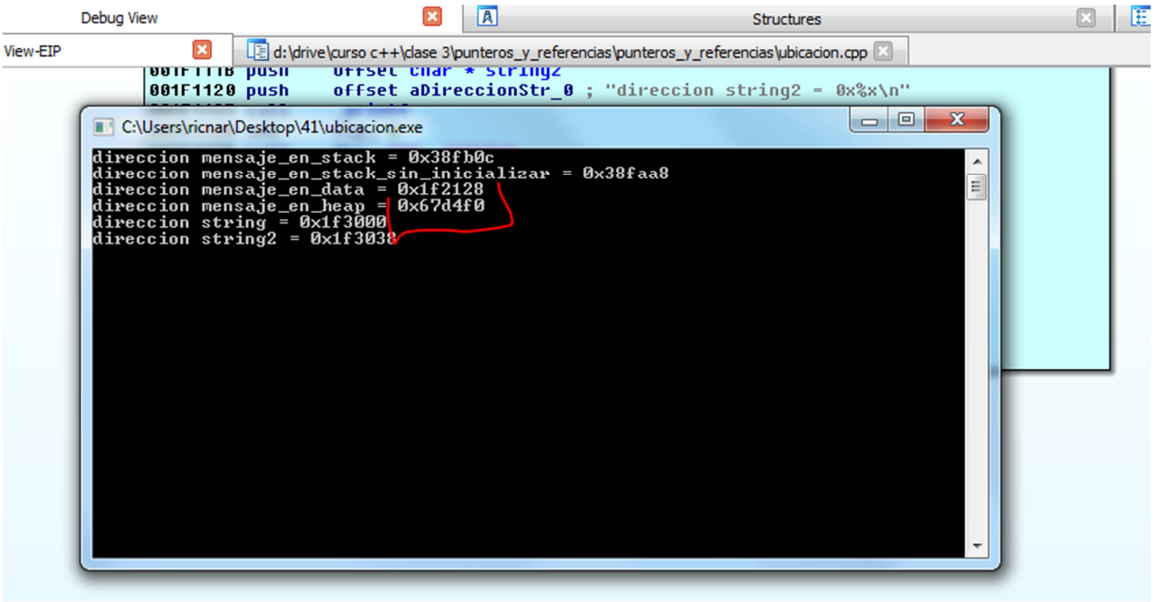
It will write 4 bytes stepping the word “data” with “heap”.

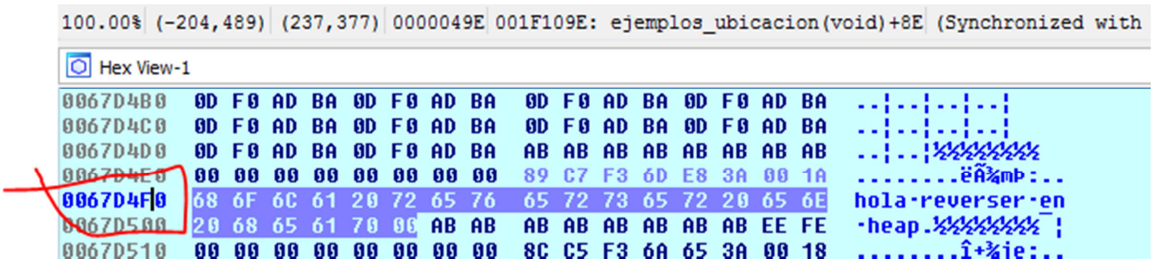


So, I can decompose the word heap and create the whole string.



The string is already created in the heap. It only remains to print the address.





That´s all folks. Try to practice, see the strings well and handle them comfortably to get used to it slowly.

Ricardo Narvaja

Translated by: @IvinsonCLS