

MAS SOBRE PUNTEROS-INCREMENTAR PUNTEROS

Tratemos de reforzar bien el concepto de punteros aunque seamos redundantes.

Es sencillo entender que si declaramos una variable int

```
int pepe =5
```

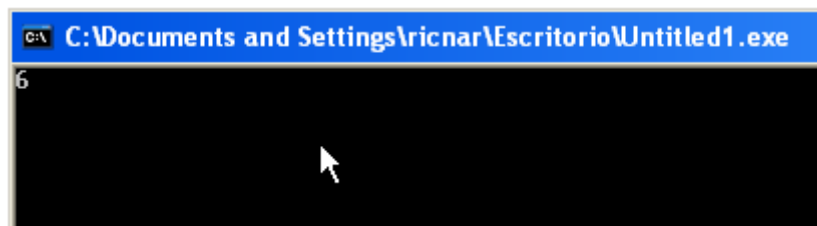
y hacemos

```
pepe ++
```

el valor de pepe valdrá 6, eso esta bien claro, el código completo

```
#include <stdio.h>  
#include <stdlib.h>  
  
main() {  
  
int pepe =5;  
  
    pepe ++;  
    printf ("%d",pepe);  
  
    getchar();  
}
```

Si lo corremos muestra que pepe vale 6



Ahora veamos el siguiente código:

```
#include <stdio.h>  
#include <stdlib.h>  
  
main() {  
  
int * punt;  
  
    punt = (int *) malloc (sizeof(int));  
    *punt = 3;
```

```
printf ("%x \n",punt);

punt ++;

printf ("%x\n",punt);

getchar();
}
```

declaramos una variable llamada **punt** que es un puntero

```
int * punt;
```

Lo inicializamos, por supuesto tendrá una dirección de memoria que devuelve el malloc

```
punt = (int *) malloc (sizeof(int));
```

guardamos un **3** en su contenido o sea en la dirección adonde apunta

```
*punt = 3;
```

Y luego incrementamos **punt**

```
punt ++;
```

Nosotros hemos incrementado el valor de “**punt**”. Como “**punt**” es un puntero, estamos modificando una dirección de memoria. Por ejemplo, si “**punt**” se refería a la posición de memoria número 10.000 de nuestro ordenador, ahora ya no es así, ahora es otra posición de memoria distinta.

Como ya sabemos, el espacio que ocupa una variable en C depende del sistema operativo. Así, en un sistema operativo de 32 bits, un “int” ocuparía 4 bytes, de modo que la operación

```
punt++;
```

haría que pasáramos de mirar la posición 10.000 a la 10.004.

Por si alguien tiene dudas ejecutemoslo:

```
#include <stdio.h>
#include <stdlib.h>

main() {

    int * punt;

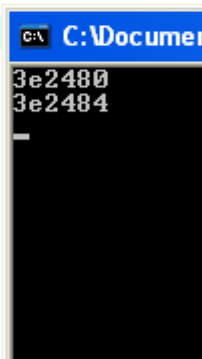
    punt = (int *) malloc (sizeof(int))
    *punt = 3;

    printf ("%x \n",punt);

    punt ++;

    printf ("%x\n",punt);

    getchar();
}
```



Como le puse **%x** me mostrara el valor hexadecimal de **punt**, vemos que en mi maquina (en otras variara), **punt** vale **0x3e2480** y luego de incrementarlo valdra **0x3e2484** ya que se incrementa de cuatro en cuatro para apuntar a un segundo int, que no hemos inicializado.

Si quisiéramos incrementar el 3 que guardamos en el contenido de **punt** deberíamos hacer.

```
(*punt) ++;
```

En este codigo

```
#include <stdio.h>
#include <stdlib.h>
```

```
main() {
```

```
int * punt;
```

```
    punt = (int *) malloc (sizeof(int));
    *punt = 3;
```

```
    printf ("%x \n",punt);
```

```
    printf ("%x\n",*punt);
```

```
    (*punt) ++;
```

```
    printf ("%x\n",*punt);
```

```
    getchar();  
}
```

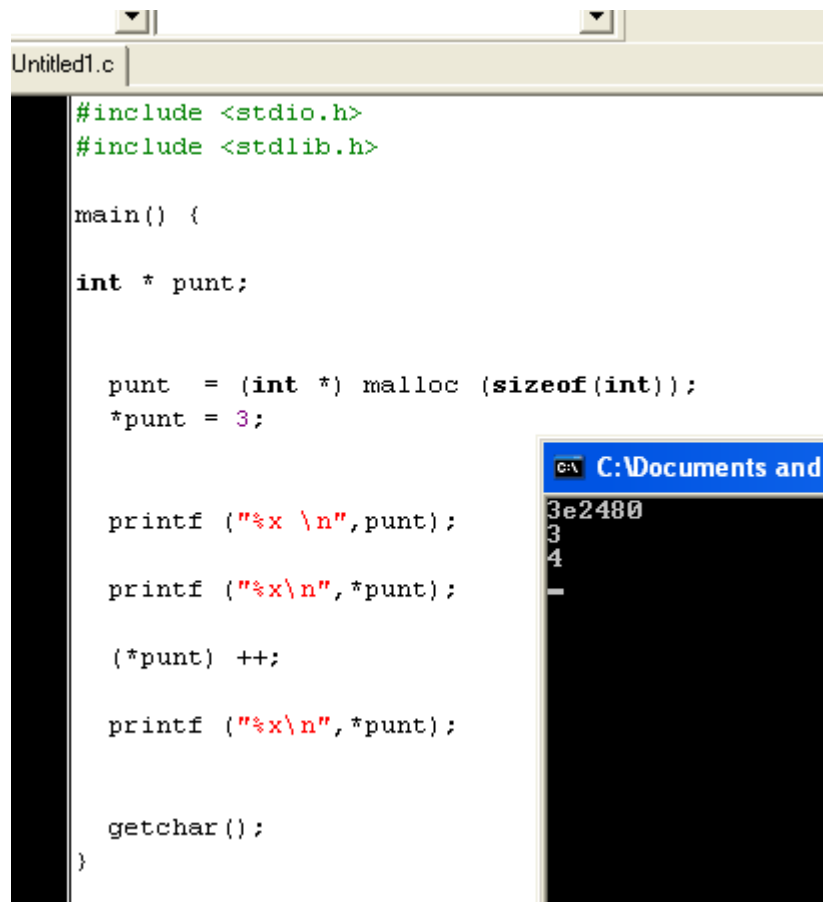
Vemos que primero imprimo el valor del puntero allí resaltado en verde, luego imprimo su contenido y luego lo incremento de 3 a 4 y luego lo imprimo nuevamente.

```
printf ("%x \n",punt);
```

```
printf ("%x\n",*punt);
```

```
(*punt) ++;
```

```
printf ("%x\n",*punt);
```



The screenshot shows a C program in a text editor window titled 'Untitled1.c'. The code defines a pointer 'punt' of type 'int', allocates memory for it using 'malloc', and assigns the value 3. It then prints the memory address of 'punt' (3e2480), the value at that address (3), increments the value to 4, and prints the new value (4). The output is shown in a separate window titled 'C:\Documents and Settings\...'.

```
#include <stdio.h>  
#include <stdlib.h>  
  
main() {  
  
    int * punt;  
  
    punt = (int *) malloc (sizeof(int));  
    *punt = 3;  
  
    printf ("%x \n",punt);  
  
    printf ("%x\n", *punt);  
  
    (*punt) ++;  
  
    printf ("%x\n", *punt);  
  
    getchar();  
}
```

C:\Documents and Settings\...
3e2480
3
4
-

Allí vemos que se incremento de 3 a 4.

La conclusión de todo esto es que para incrementar punteros usaremos

punt ++

y para incrementar su contenido

(* punt) ++

Lo mismo que pasa asignar valores a un puntero se hará directamente en **punt** y para asignar en su contenido se hará en ***punt**.

Asignando valores a **punt**

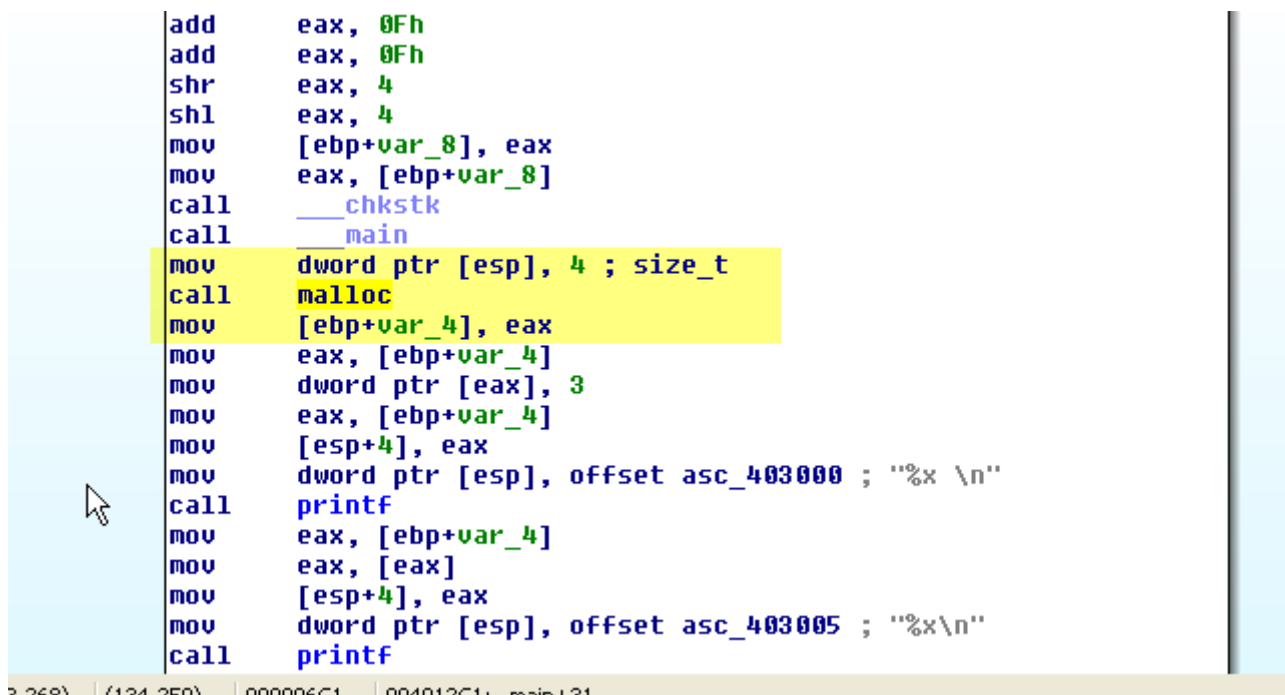
```
punt = (int *) malloc (sizeof(int));
```

Asignando valores a su contenido

```
*punt = 3;
```

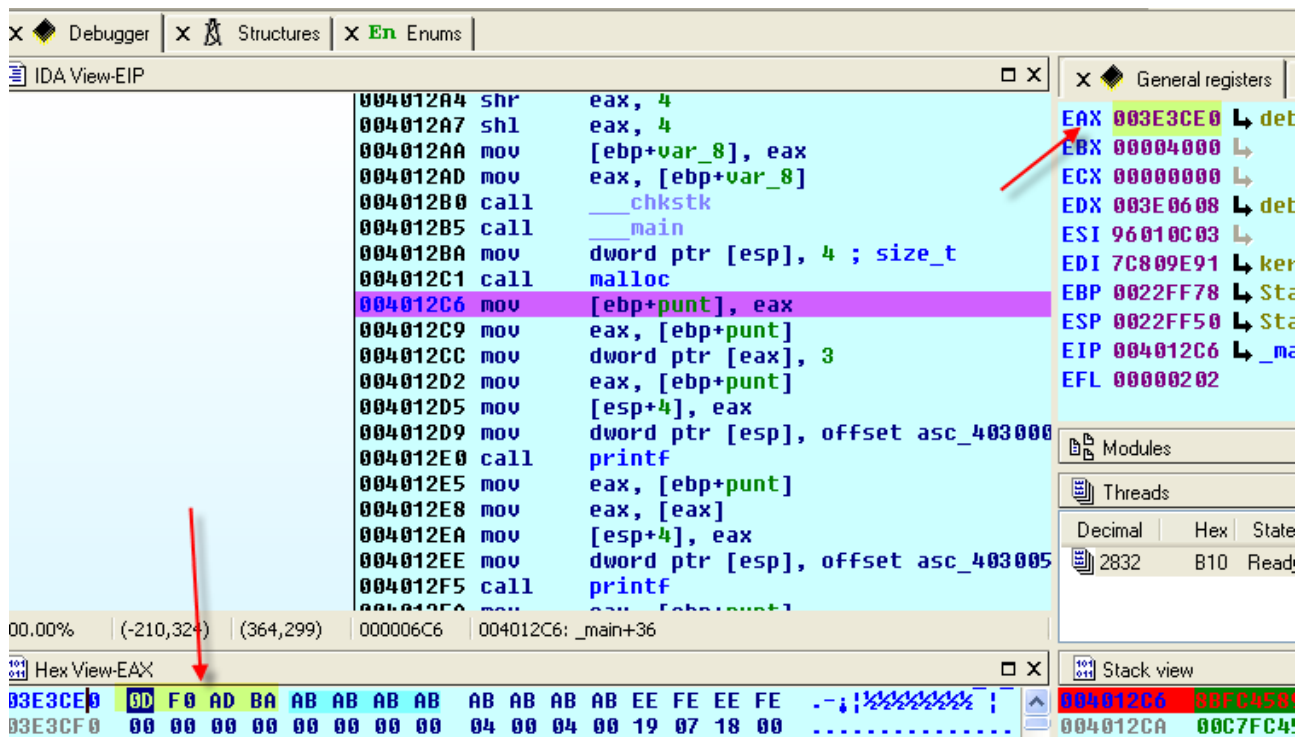
Es muy importante que esto quede claro, pues es la piedra fundamental de todo lo que sigue, por si alguno le quedo alguna duda veamoslo en IDA.

Allí vemos el código luego de lo agregado por el compilador



```
add    eax, 0Fh
add    eax, 0Fh
shr    eax, 4
shl    eax, 4
mov     [ebp+var_8], eax
mov     eax, [ebp+var_8]
call    ___chkstk
call    main
mov     dword ptr [esp], 4 ; size_t
call    malloc
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
mov     dword ptr [eax], 3
mov     eax, [ebp+var_4]
mov     [esp+4], eax
mov     dword ptr [esp], offset asc_403000 ; "%x \\n"
call    printf
mov     eax, [ebp+var_4]
mov     eax, [eax]
mov     [esp+4], eax
mov     dword ptr [esp], offset asc_403005 ; "%x\\n"
call    printf
```

Vemos que llama a **malloc** con el size 4 y que el resultado lo devuelve en **var_4** que sera nuestro **punt**, así que lo renombramos, sabemos que **punt** es un puntero o sea que sera una dirección de memoria, si ponemos un breakpoint allí.



Vemos que EAX tiene el valor de **punt** que en mi maquina es **0x3e3ce0** y que como lo arrancamos en un debugger y le pedimos 4 bytes en la memoria el contenido de **0x3e3ce0** tendrá BAAD FOOD jeje como dijimos en las partes anteriores y ABABABAB donde termina la zona reservada.



La cuestión es que luego de guardar EAX en **punt** lo que corresponde a nuestro código fuente

```
punt = (int *) malloc (sizeof(int));
```

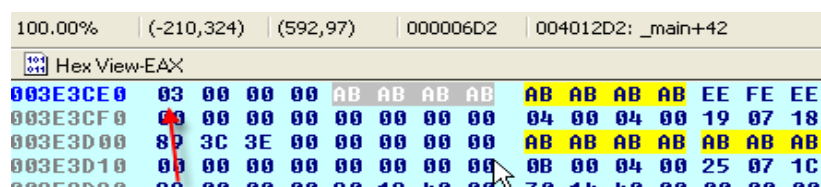
Lo vuelve a mover a EAX y guarda en su contenido, donde hace

```
004012CC mov    dword ptr [eax], 3
```

esta realizando lo que en nuestro código fuente era

```
*punt = 3;
```

Y modificando el contenido de **punt**, si pasamos con f8 dicha instrucción.



Vemos que guardo el 3.

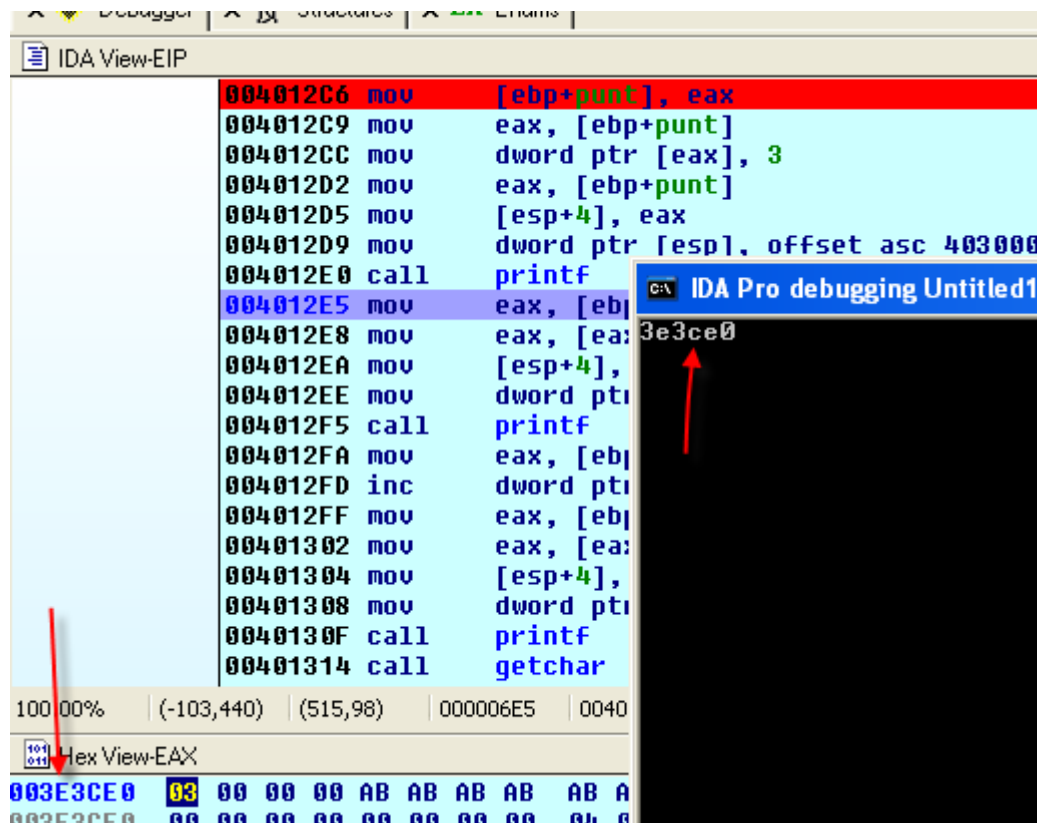
Luego vemos los tres **printf**

```
004012C6 mov     dword ptr [eax], 3
004012D2 mov     eax, [ebp+punt]
004012D5 mov     [esp+4], eax
004012D9 mov     dword ptr [esp], offset asc_403000 ; "%x \n"
004012E0 call    printf
004012E5 mov     eax, [ebp+punt]
004012E8 mov     eax, [eax]
004012EA mov     [esp+4], eax
004012EE mov     dword ptr [esp], offset asc_403005 ; "%x\n"
004012F5 call    printf
004012FA mov     eax, [ebp+punt]
004012FD inc     dword ptr [eax]
004012FF mov     eax, [ebp+punt]
00401302 mov     eax, [eax]
00401304 mov     [esp+4], eax
00401308 mov     dword ptr [esp], offset asc_403005 ; "%x\n"
0040130F call    printf
```

En el primero lee el valor de **punt** y lo imprime.

```
.text:004012D2 mov     eax, [ebp+punt]
.text:004012D5 mov     [esp+4], eax
.text:004012D9 mov     dword ptr [esp], offset asc_403000 ; "%x \n"
.text:004012E0 call    printf
```

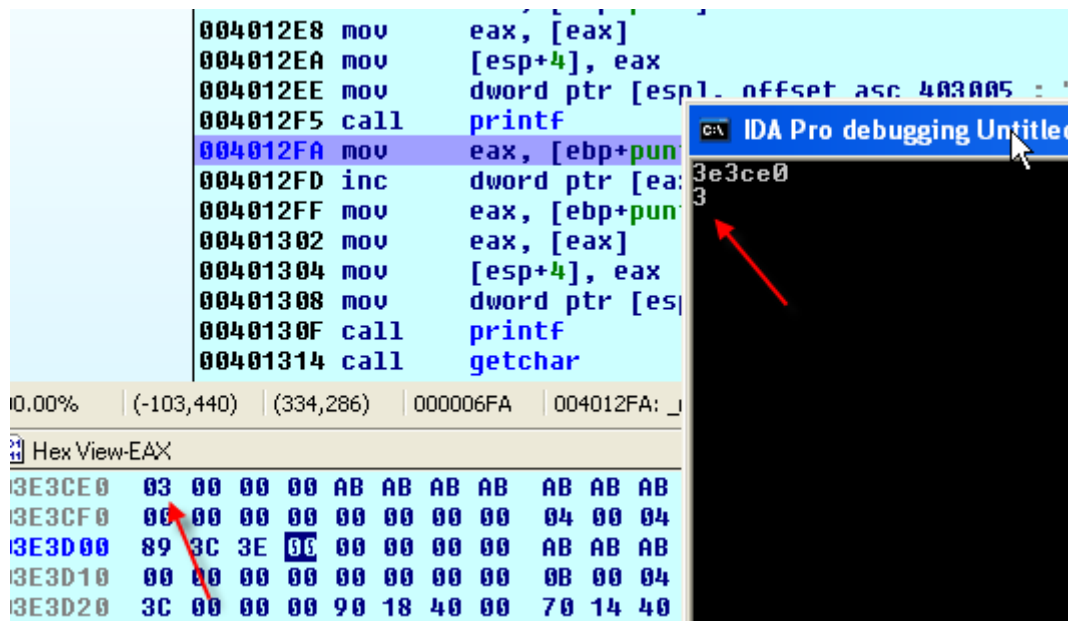
Si traceamos con f8 y lo pasamos al **printf** vemos en la consola el valor de **punt**.



El siguiente **printf** debería imprimir el contenido de **punt** o sea ***punt**.

```
.text:004012E5 mov    eax, [ebp+punt]
.text:004012E8 mov    eax, [eax]
.text:004012EA mov    [esp+4], eax
.text:004012EE mov    dword ptr [esp], offset asc_403005 ; "%x\n"
.text:004012F5 call   printf
```

Allí vemos la instrucción donde obtiene el contenido de **punt** o sea el valor 3 si pasamos el **printf** con f8, vemos en la consola el valor de ***punt**.



En el ultimo printf incrementábamos ***punt** de 3 a 4 y lo imprimíamos veamos.

```
.text:004012FA mov    eax, [ebp+punt]
.text:004012FD inc    dword ptr [eax]
```

En EAX estará **punt** y incrementara su contenido mediante la instrucción en verde, pasando el mismo a 4 si pasamos con f8 el **INC**.


```

004012F5 call    printf
004012FA mov     eax, [ebp+punt]
004012FD inc     dword ptr [eax]
004012FF mov     eax, [ebp+punt]
00401302 mov     eax, [eax]
00401304 mov     [esp+4], eax
00401308 mov     dword ptr [esp], offset asc_403005 ; "%x\n"
0040130F call    printf
00401314 call    getchar
00401319 leave
0040131A retn
0040131A _main endp
0040131A

```

100.00% | (-103,515) | (334,297) | 000006FF | 004012FF: _main+6F

Hex View-1

003E3CE0 04 00 00 00 AB AB AB AB AB AB AB AB EE FE EE FE ...
003E3CF0 00 00 00 00 00 00 00 00 04 00 04 00 19 07 18 00

Que fue el equivalente en nuestro código de

`(*punt) ++;`

Luego mueve el 4 a EAX y lo imprime.

```

.text:00401302 mov     eax, [eax]
.text:00401304 mov     [esp+4], eax
.text:00401308 mov     dword ptr [esp], offset asc_403005 ; "%x\n"
.text:0040130F call    printf

```

```

004012FD inc     dword ptr [eax]
004012FF mov     eax, [ebp+punt]
00401302 mov     eax, [eax]
00401304 mov     [esp+4], eax
00401308 mov     dword ptr [esp], offset asc_403005 ; "%x\n"
0040130F call    printf
00401314 call    getchar
00401319 leave
0040131A retn
0040131A _main endp
0040131A

```

100.00% | (-103,515) | (360,293) | 00000714 | 00401314: _r

Hex View-1

003E3CE0 04 00 00 00 AB AB AB AB AB AB AB AB
003E3CF0 00 00 00 00 00 00 00 00 04 00 04
003E3D00 89 3C 3E 00 00 00 00 00 AB AB AB

IDA Pro debugging Unti
3e3ce0
3
4

Al llegar a este punto muchos que trabajaron en bajo nivel siempre se preguntan, que diferencia hay entre & y *, ya que ambos son direcciones de memoria, y realmente la diferencia es mas de C que de bajo nivel, pero si entendimos todo hasta aquí vemos que & es usando para pasar direcciones de memoria de variables como argumento a otras funciones donde dicha variable no esta definida, o para incrementar el valor de dicha variable, es similar al LEA aunque en la practica nos da una dirección de memoria cuyo contenido es una variable, en cambio * es también una dirección de memoria pero en esta caso la variable puntero contendrá una dirección de memoria, en cuyo contenido podremos trabajar, la diferencia es muy fina y es posible que todavía haya dudas, veremos como usar ambas a la vez en el siguiente ejemplo.

```
#include <stdio.h>

void duplica(int *x) {
    *x = *x * 2;
}

main() {
    int n = 5;
    printf("n vale %d\n", n);
    duplica(&n);
    printf("Ahora n vale %d\n", n);

    getchar();
}
```

Vemos que la función **duplica** tiene definido que recibe un puntero (int *) y busca su contenido y lo multiplica por dos, modificando el contenido allí mismo sin devolver el valor.

```
void duplica(int *x) {
    *x = *x * 2;
}
```

En el main cuando se llama a la función **duplica** se pasa una dirección de memoria de la variable **n** que vale 5, por ejemplo le pasa **0x10000**, al pasarle la dirección de memoria a **duplica**, esta la toma, busca el contenido de **0x10000**, que es 5 y lo multiplica por dos y lo modifica allí mismo, guardando en **0x10000** el 10 decimal.

```
main() {
    int n = 5;
    printf("n vale %d\n", n);
    duplica(&n);
    printf("Ahora n vale %d\n", n);
}
```

Por eso en el **printf** aparece **n** con un nuevo valor que no se cambio en la función main, este es un ejemplo de lo que decíamos en las primeras partes del curso, que para modificar el valor de una variable local, ingresando en otra función, donde dicha variable no esta definida, la única forma es pasarle con **&** o **LEA** su dirección y que luego dentro de la misma se trabaje en el contenido de dicha variable, lo cual se realiza por ejemplo si la dirección esta en EAX usando [EAX] para modificar su contenido y eso es lo que hace duplica usando * para modificar el contenido de una dirección.

O sea este es el ejemplo clásico de como modificar el valor de una variable local usando & o LEA, dentro de otra función y por eso cuando estamos reverseando una función, y vemos que en IDA al apretar X en una variable local, si además de la asignaciones directas, hay un LEA, enseguida nos fijamos en esa instrucción, pues allí puede cambiar de valor de dicha variable al entrar en otra subfunción, o al modificar el contenido de la memoria.

```

#include <stdio.h>

void duplica(int *x) {
    *x = *x * 2;
}

main() {
    int n = 5;
    printf("n vale %d\n", n);
    duplica(&n);
    printf("Ahora n vale %d\n", n);

    getchar();
}

```

Si lo vemos en IDA.

```

call    __main
mov     [ebp+n], 5
mov     eax, [ebp+n]
mov     [esp+4], eax
mov     dword ptr [esp], offset aNvaleD ; "n vale %d\n"
call    printf
lea     eax, [ebp+n]
mov     [esp], eax
call    sub_401290
mov     eax, [ebp+n]
mov     [esp+4], eax
mov     dword ptr [esp], offset aAhoraNvaleD ; "Ahora n vale %d\n"
call    printf
call    getchar
leave
ret

```

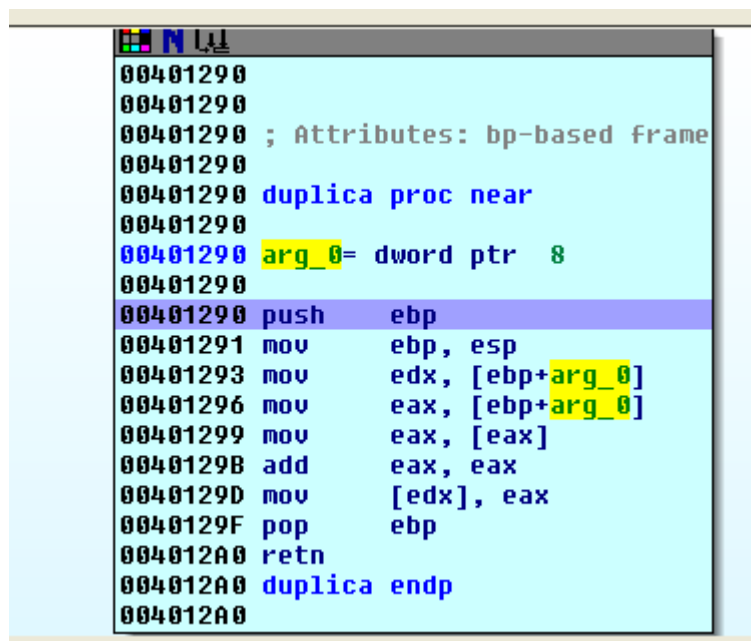
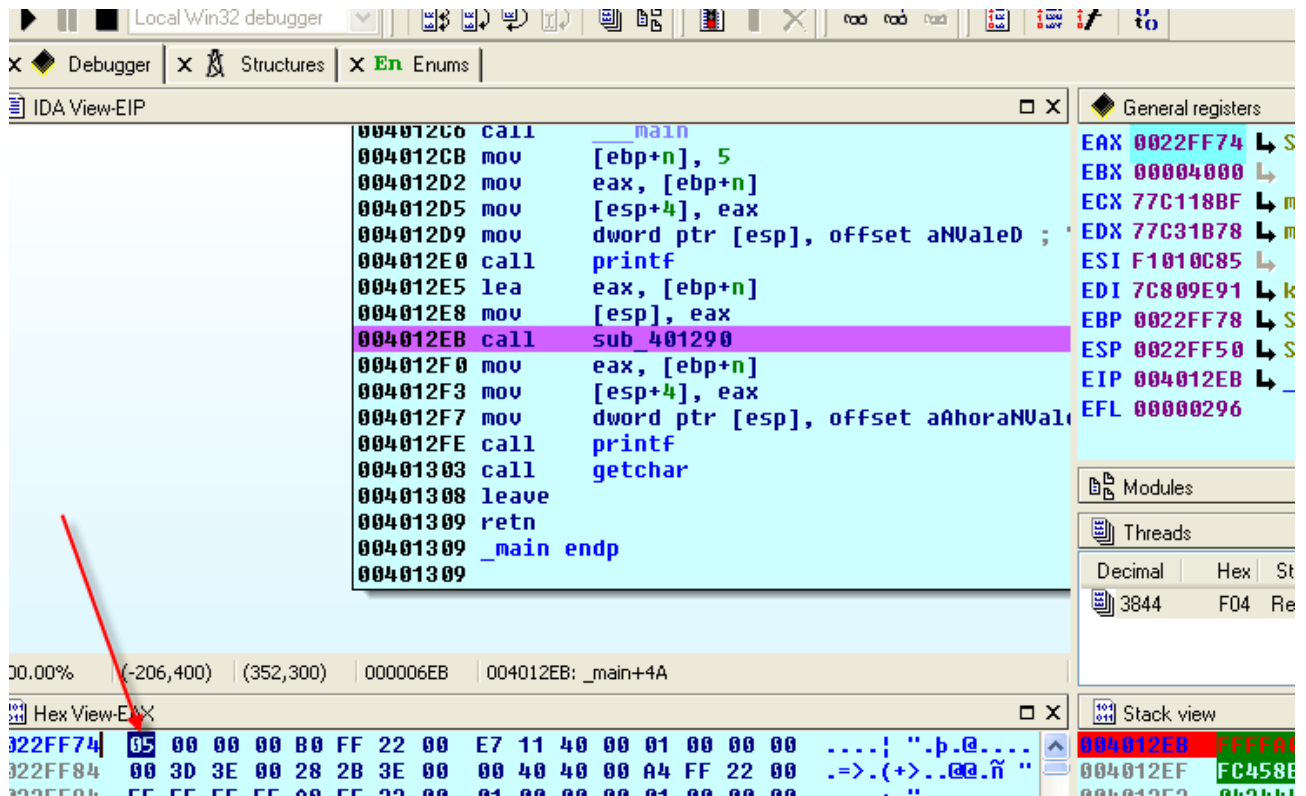
Vemos que **n** vale 5 y usa printf para imprimir su valor.

```

lea     eax, [ebp+n]

```

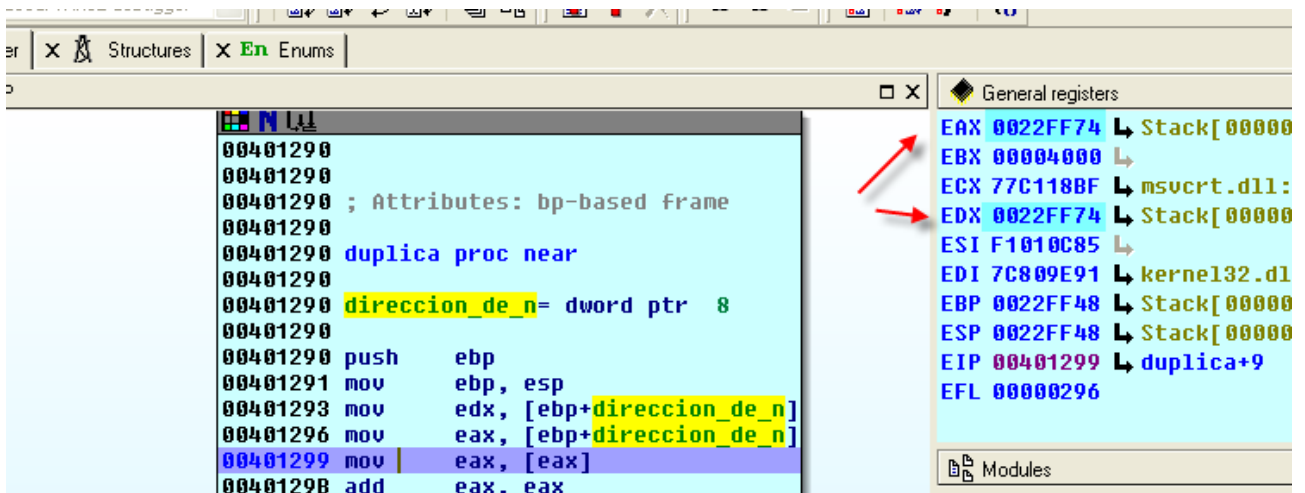
Luego le pasa la dirección de memoria de la variable **n**, si ponemos un breakpoint allí, vemos que en EAX estará la misma, la cual se pasa a la función **duplica**, entramos en ella.



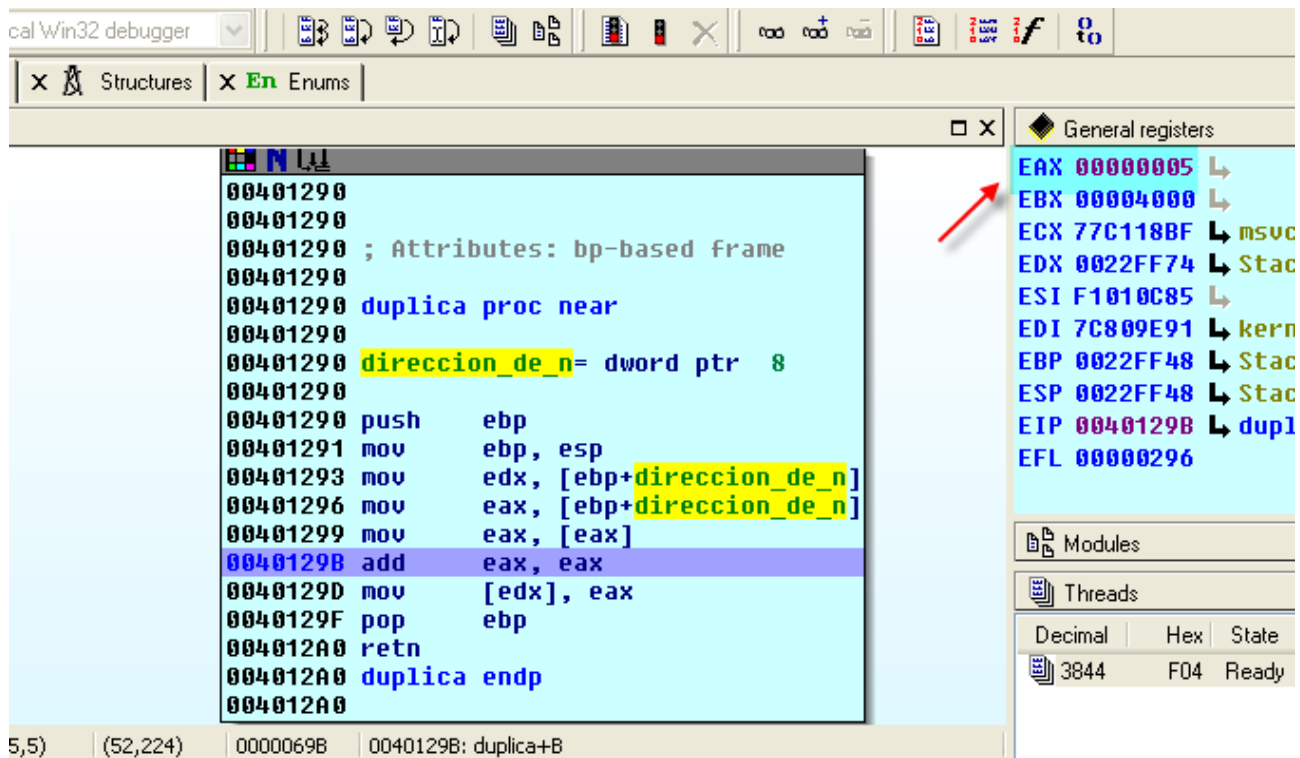
Si traceamos vemos que lógicamente `arg_0` sera la dirección de memoria de `n`, aquí `n` no tiene significado pues es una variable local, pero si podemos trabajar con su dirección de memoria y cambiar su contenido.

```
00401290 ; Attributes: bp-based frame
00401290
00401290 duplica proc near
00401290 direccion_de_n= dword ptr 8
00401290
00401290 push    ebp
00401291 mov     ebp, esp
00401293 mov     edx, [ebp+direccion_de_n]
00401296 mov     eax, [ebp+direccion_de_n]
00401299 mov     eax, [eax]
0040129B add     eax, eax
0040129D mov     [edx], eax
0040129F pop     ebp
004012A0 retn
004012A0 duplica endp
004012A0
```

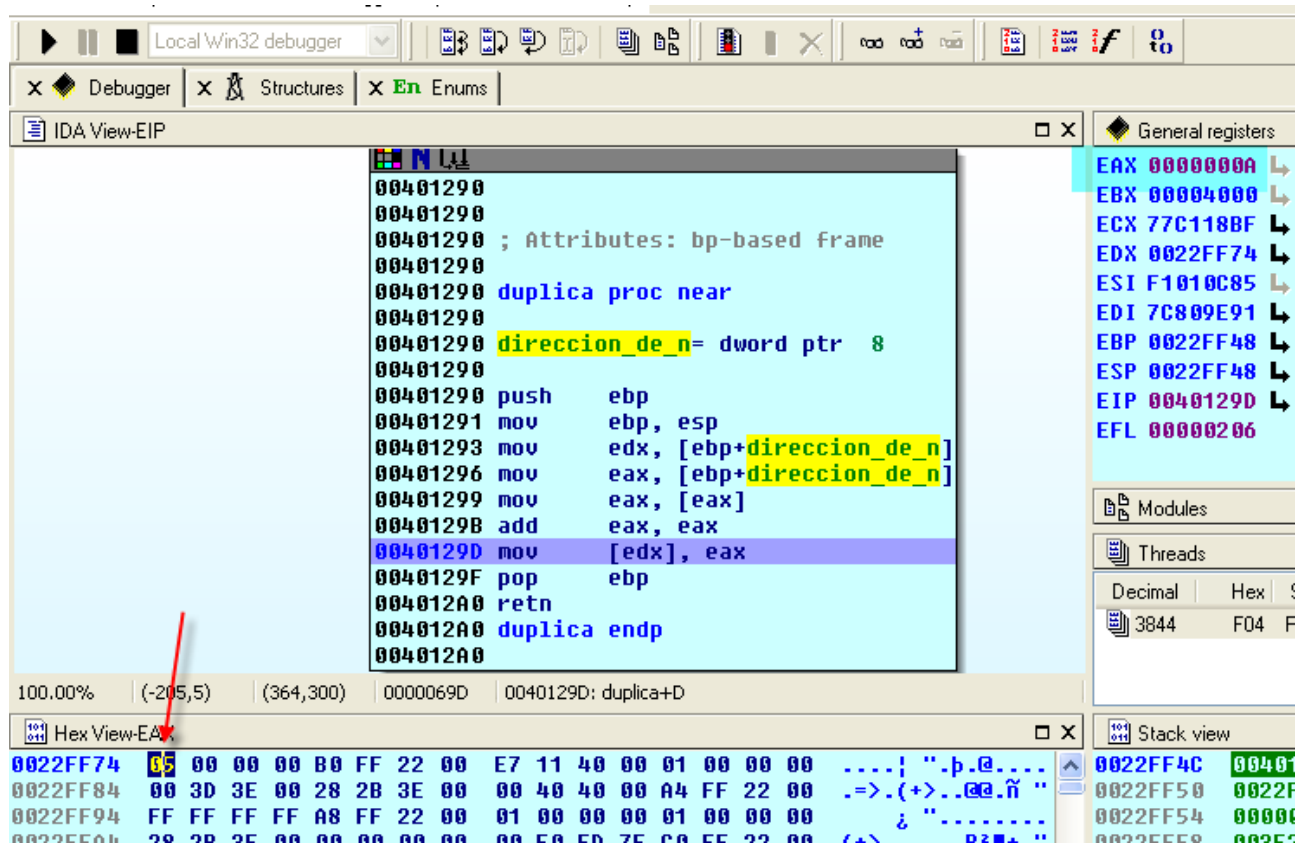
Aquí el argumento **direccion_de_n** es un puntero, vemos que lo mueve a EDX y a EAX.



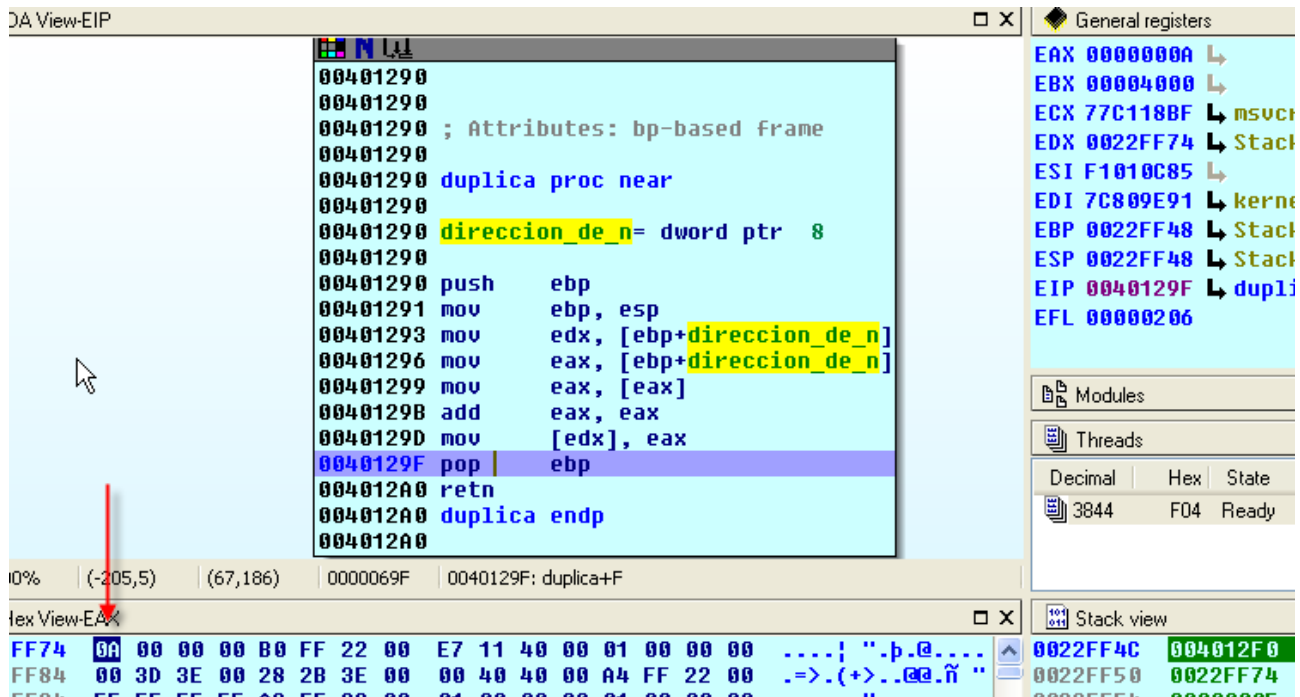
Luego mueve a EAX el contenido de dicha dirección de memoria que sera el valor de **n** que estará en el contenido de EAX.



Luego con **add EAX,EAX** duplica el valor. y lo vuelve a guardar en el contenido de EDX o sea reemplaza el 5 por el 10.



Al ejecutar el **mov [EDX],EAX** cambiara a 10 decimal o sea 0a hexa.



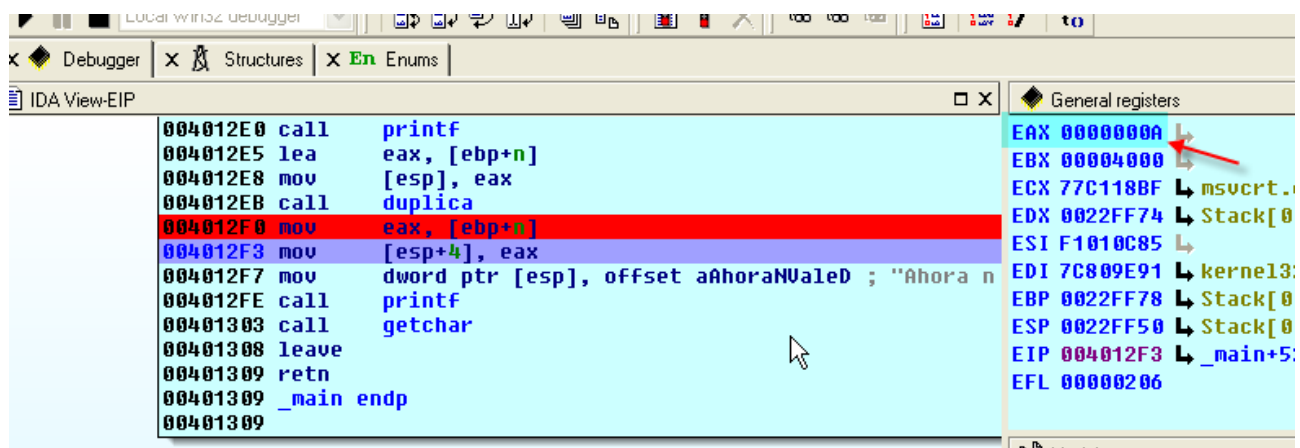
Y allí volvemos a main.

```

004012E5 lea     eax, [ebp+n]
004012E8 mov     [esp], eax
004012EB call    duplica
004012F0 mov     eax, [ebp+n]
004012F3 mov     [esp+4], eax
004012F7 mov     dword ptr [esp], offset a
004012FE call    printf
00401303 call    getchar
00401308 leave
00401309 retn
00401309 _main endp
00401309

```

Ahora aquí movemos el valor de **n** a EAX pero este ya no vale 5 sino vale **0a** hexa jeje, dentro de duplica se cambio su valor.



Y si por eso cuando se pasa como argumento de una función una dirección de memoria de una variable local, que se halla con LEA, podemos esperar que dentro de dicha función se cambie el valor de la variable local.

Bueno no quiero seguir más hasta que no digieran bien esto les recomiendo que repasen bien, los ejemplos de punteros pues se va a venir más difícil si no entendieron bien esto.

Hasta la parte siguiente, adjunto un ejemplo para reversear para que practiquen
Ricardo Narvaja