

Punteros y gestión dinámica de memoria

¿Por qué usar estructuras dinámicas?

(Del curso de C de Cabanes)

Hasta ahora teníamos una serie de variables que declaramos al principio del programa o de cada función. Estas variables, que reciben el nombre de ESTÁTICAS, tienen un tamaño asignado desde el momento en que se crea el programa.

Este tipo de variables son sencillas de usar y rápidas... si sólo vamos a manejar estructuras de datos que no cambien, pero resultan poco eficientes si tenemos estructuras cuyo tamaño no sea siempre el mismo.

Es el caso de una agenda: tenemos una serie de fichas, e iremos añadiendo más. Si reservamos espacio para 10, no podremos llegar a añadir la número 11, estamos limitando el máximo. Una solución sería la de trabajar siempre en el disco: no tenemos límite en cuanto a número de fichas, pero es muchísimo más lento.

Lo ideal sería aprovechar mejor la memoria que tenemos en el ordenador, para guardar en ella todas las fichas o al menos todas aquellas que quepan en memoria.

Una solución “típica” (pero mala) es sobredimensionar: preparar una agenda contando con 1000 fichas, aunque supongamos que no vamos a pasar de 200. Esto tiene varios inconvenientes: se desperdicia memoria, obliga a conocer bien los datos con los que vamos a trabajar, sigue pudiendo verse sobrepasado, etc.

La solución suele ser crear estructuras DINÁMICAS, que puedan ir creciendo o disminuyendo según nos interesen.

Todas estas estructuras tienen en común que, si se programan bien, pueden ir creciendo o decreciendo según haga falta, al contrario que un array, que tiene su tamaño prefijado.

En todas ellas, lo que vamos haciendo es reservar un poco de memoria para cada nuevo elemento que nos haga falta, y enlazarlo a los que ya teníamos. Cuando queramos borrar un elemento, enlazamos el anterior a él con el posterior a él (para que no “se rompa” nuestra estructura) y liberamos la memoria que estaba ocupando.

¿Qué son los punteros?

Un puntero no es más que una dirección de memoria. Lo que tiene de especial es que normalmente un puntero tendrá un tipo de datos asociado: por ejemplo, un “puntero a entero” será una dirección de memoria en la que habrá almacenado (o podremos almacenar) un número entero.

Vamos a ver qué símbolo usamos en C para designar los punteros:

int num; //donde "num" es un número entero

int *pos; //donde "pos" es un "puntero a entero" (dirección de memoria en la que podremos guardar un entero)

Esta nomenclatura ya la habíamos utilizado aun sin saber que era eso de los punteros. Por ejemplo, cuando queremos acceder a un fichero, hacemos

FILE* fichero;

Antes de entrar en más detalles, y para ver la diferencia entre trabajar con “arrays” o con punteros, vamos a hacer dos programas que pidan varios números enteros al usuario y muestren su suma. El primero empleará un “array” (una tabla, de tamaño predefinido) y el segundo empleará memoria que reservaremos durante el funcionamiento del programa.

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

```
main() {
    int datos[100]; /* Preparamos espacio para 100 numeros */
    int cuantos;    /* Preguntaremos cuantos desea introducir */
    int i;          /* Para bucles */
    long suma=0;    /* La suma, claro */

    do {
        printf("Cuantos numeros desea sumar? ");
        scanf("%d", &cuantos);
        if (cuantos>100) /* Solo puede ser 100 o menos */
            printf("Demasiados. Solo se puede hasta 100.");
    } while (cuantos>100); /* Si pide demasiado, no le dejamos */

    /* Pedimos y almacenamos los datos */
    for (i=0; i<cuantos; i++) {
        printf("Introduzca el dato número %d: ", i+1);
        scanf("%d", &datos[i]);
    }

    /* Calculamos la suma */
    for (i=0; i<cuantos; i++)
        suma += datos[i];

    printf("Su suma es: %ld\n", suma);
}
```

Bueno el programa es similar a los que vimos anteriormente tiene un array de tamaño 100, preparado para ingresar y almacenar datos, y dentro de un **do-while** que es igual a un **while** pero con la condición al final.

El punto en que comienza a repetirse se indica con la orden “**do**”, así:

```
do
sentencia;
while (condición);
```

O sea que aquí se repetiría **sentencia** mientras la condición sea verdadera, volvamos a nuestro ejemplo.

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

```

main() {
    int datos[100]; /* Preparamos espacio para 100 numeros */
    int cuantos;    /* Preguntaremos cuantos desea introducir */
    int i;          /* Para bucles */
    long suma=0;    /* La suma, claro */

```

Aquí esta donde declaramos las variables, el array de enteros de tamaño 100 se llama **datos**, una variable **int** llamada **cuantos** para guardar cuantos datos vamos a ingresar, un int llamado **i** como contador para los ciclos, y un **long** llamado **suma**, que lo inicializamos a cero, para guardar el resultado.

```

do {
    printf("Cuantos numeros desea sumar? ");
    scanf("%d", &cuantos);
    if (cuantos>100) /* Solo puede ser 100 o menos */
        printf("Demasiados. Solo se puede hasta 100.");
    } while (cuantos>100); /* Si pide demasiado, no le dejamos */

```

Luego en este while se repetirá lo que esta resaltado mientras **cuantos** sea mayor que 100, o sea te volverá a preguntar la cantidad eternamente, mientras pongas una cantidad mayor que 100, cuando pongas una cantidad menor, te dejara seguir y salir del while.

```

for (i=0; i<cuantos; i++) {
    printf("Introduzca el dato número %d: ", i+1);
    scanf("%d", &datos[i]);
}

```

Luego tenemos un **for** donde el contador es **i**, el cual se inicializa en cero y se incrementa de uno en uno hasta llegar al máximo que es **cuantos**, dentro de este for, se van introduciendo los campos del array usando **scanf**, y guardándolo en los campos del array **datos[i]**.

```

for (i=0; i<cuantos; i++)
    suma += datos[i];

printf("Su suma es: %ld\n", suma);

```

Luego otro **for** ira sumando todos los campos y guardando en **suma** y al final lo imprimirá.

Si compilamos y lo vemos en IDA, vemos allí donde marca la flecha roja donde comienza realmente nuestro programa ya que lo anterior es agregado del compilador.

```

; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

var_1AC= dword ptr -1ACh
var_1A4= dword ptr -1A4h
var_1A0= dword ptr -1A0h
var_19C= dword ptr -19Ch
var_198= dword ptr -198h
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 1C8h
and     esp, 0FFFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_1AC], eax
mov     eax, [ebp+var_1AC]
call    __chkstk
call    __main
mov     [ebp+var_1A4], 0

```

Dentro del ciclo

```

loc_4012CD:                ; "Cuantos numeros desea sumar? "
mov     dword ptr [esp], offset aCuantosNumeros
call    printf
lea     eax, [ebp+var_19C]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf
cmp     [ebp+var_19C], 64h
jle     short loc_401304

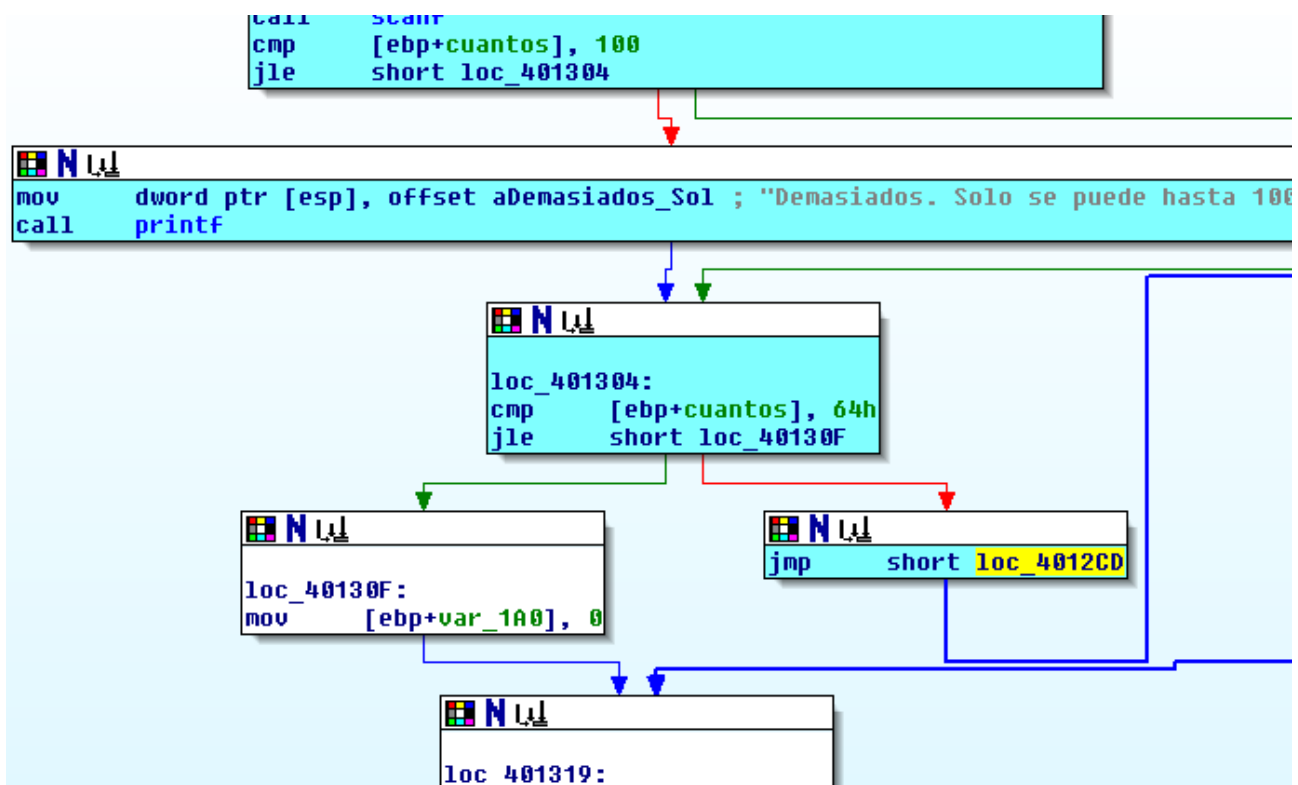
```

Vemos allí resaltada la variable **cuantos** que es donde mediante **scanf** se guarda la cantidad de datos que vamos a ingresar, la renombramos y vemos que la compara si es menor o igual que **64h** o sea 100 decimal, así que cambiamos el 64h por 100 decimal para que sea mas visible.

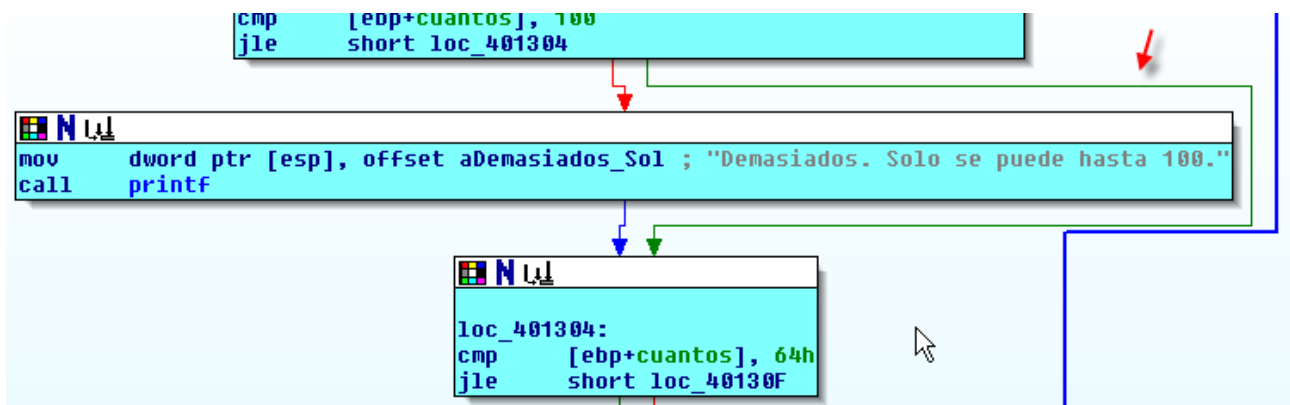
```
loc_4012CD:                                ; "Cuantos numeros desea sumar?"
mov     dword ptr [esp], offset aCuantosNumeros
call    printf
lea     eax, [ebp+cuantos]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf
cmp     [ebp+cuantos], 64h
jle     short loc_401304
```

Group nodes
Use standard symbolic constant
100 H
144n

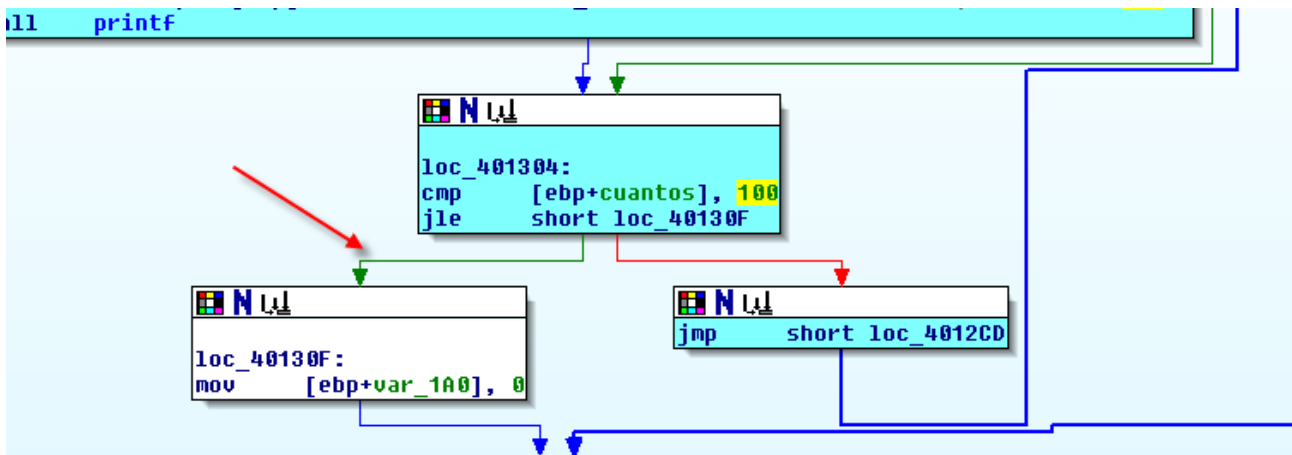
Voy coloreando los bloques que pertenecen al loop



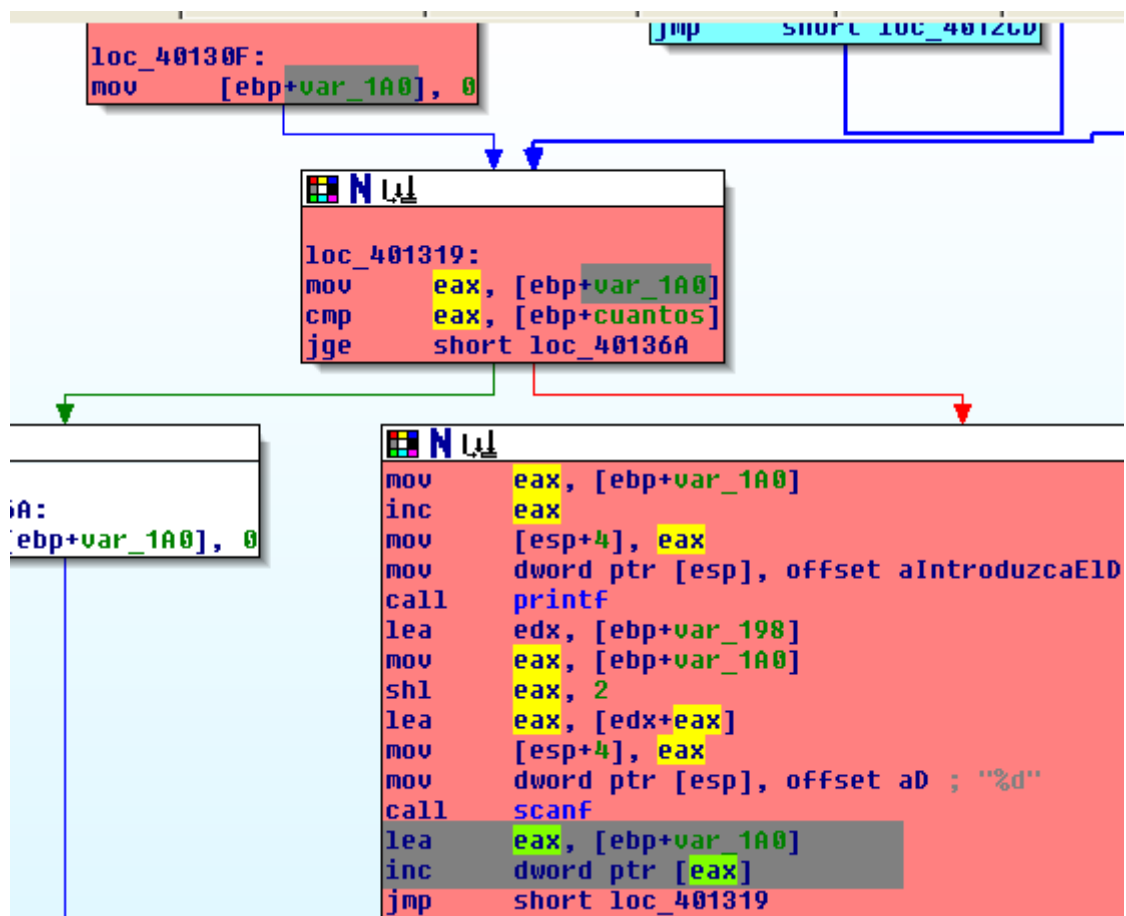
Veo que dentro del mismo si es menor o igual que 100, saltea el **printf** que dice que son “Demasiados...” siguiendo el camino de la flecha verde.



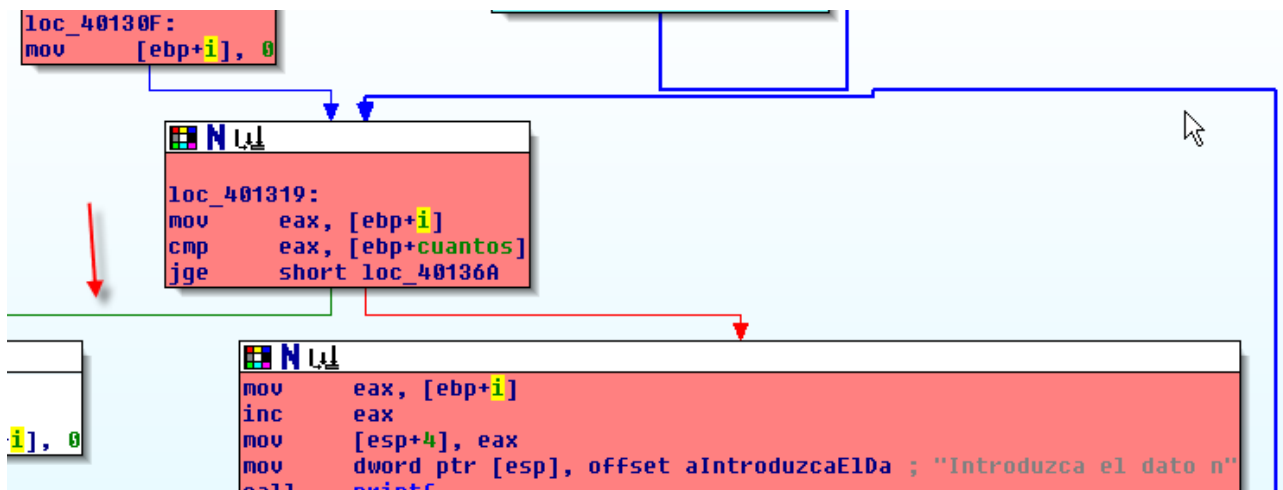
Luego ya comprueba la condición de salida volviendo a comprobar si **cuantos** es menor que 64h el cual cambiamos a 100 decimal, esto nos hace ver que es un **while** pues no hay contador y saldrá dependiendo de lo que tipee el usuario, por el camino de la flecha verde, sino va a un **jmp** que vuelve al inicio del **while**.



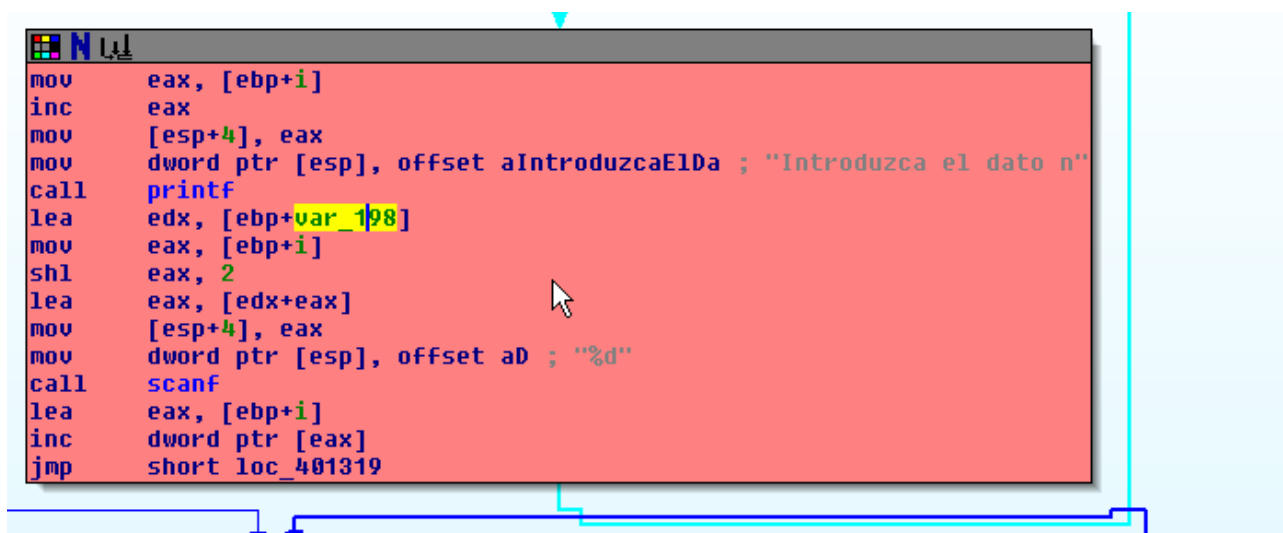
A continuación vemos un típico **for** con su contador que lo llamamos **i** en el código fuente, vemos como lo inicia al principio, luego compara la condición de salida y al final lo incrementa.



Lo renombramos a **i**.



Vemos la condición de salida marcada con la flecha, cuando **i** sea mayor o igual que **cuantos**, saldrá por allí, sino continuara por el siguiente bloque rosado.



La variable **var_198** es el inicio de nuestro array debemos definirlo en las variables, aquí vemos como pasa a EDX la dirección inicial y le va sumando **i*4** (obtenido con **shl eax,2**), para ir saltando de cuatro en cuatro los campos del array y guardar consecutivamente en cada uno, también deducimos que allí debe haber un array ya que al mirar la zona de las variables, vemos mucho espacio vacío entre **var_198** y lo siguiente definido.

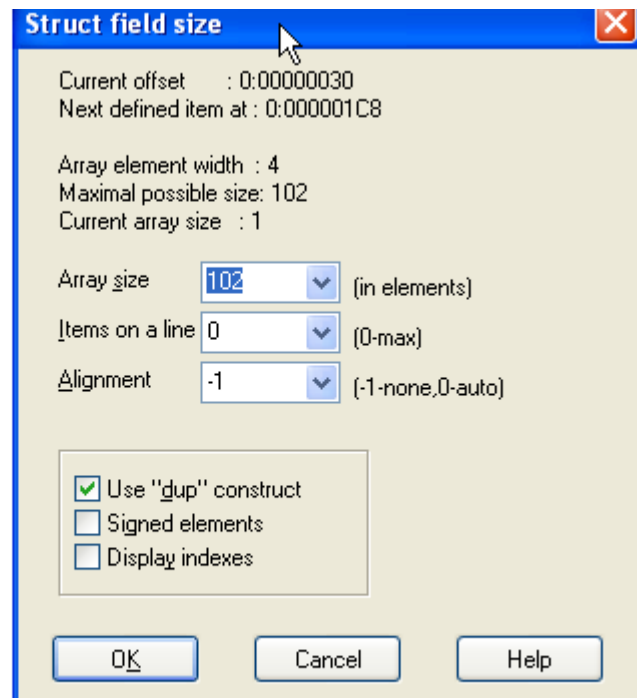
Address	Disassembly	Comment
-000001A5	db ?	; undefined
-000001A4	dd ?	
-000001A0	dd ?	
-0000019C	dd ?	
-00000198	dd ?	
-00000194	db ?	; undefined
-00000193	db ?	; undefined
-00000192	db ?	; undefined
-00000191	db ?	; undefined
-00000190	db ?	; undefined
-0000018F	db ?	; undefined
-0000018E	db ?	; undefined
-0000018D	db ?	; undefined
-0000018C	db ?	; undefined
-0000018B	db ?	; undefined
-0000018A	db ?	; undefined
-00000189	db ?	; undefined
-00000188	db ?	; undefined
-00000187	db ?	; undefined
-00000186	db ?	; undefined
-00000185	db ?	; undefined
-00000184	db ?	; undefined
-00000183	db ?	; undefined

Llega hasta el **stored ebp y return address**.

Address	Disassembly	Comment
-00000009	db ?	; undefined
-00000008	db ?	; undefined
-00000007	db ?	; undefined
-00000006	db ?	; undefined
-00000005	db ?	; undefined
-00000004	db ?	; undefined
-00000003	db ?	; undefined
-00000002	db ?	; undefined
-00000001	db ?	; undefined
+00000000	db 4 dup(?)	
+00000004	db 4 dup(?)	
+00000008	dd ?	
+0000000C	dd ?	; offset
+00000010	dd ?	; offset
+00000014		
+00000014	; end of stack variables	

Así que no hay nada que se pueda pisar hacia abajo, y hacia arriba esta todo ya definido así que apretamos asterisco, vemos que hay espacio para 102 enteros, como ya sabemos que hay un máximo de 100 porque el mismo programa nos muestra en los mensajes que el máximo es 100, le ponemos 100.

```
·000001A0 var_1A0
·000001A8
·000001A7
·000001A6
·000001A5
·000001A4 var_1A4
·000001A0 i
·0000019C cuantos
·00000198 var_198
·00000194
·00000193
·00000192
·00000191
·00000190
·0000018F
·0000018E
·0000018D
·0000018C
·0000018B
·0000018A
·00000189
·00000188
·00000187
·00000186
·00000185
·00000184
```



Quedara así.

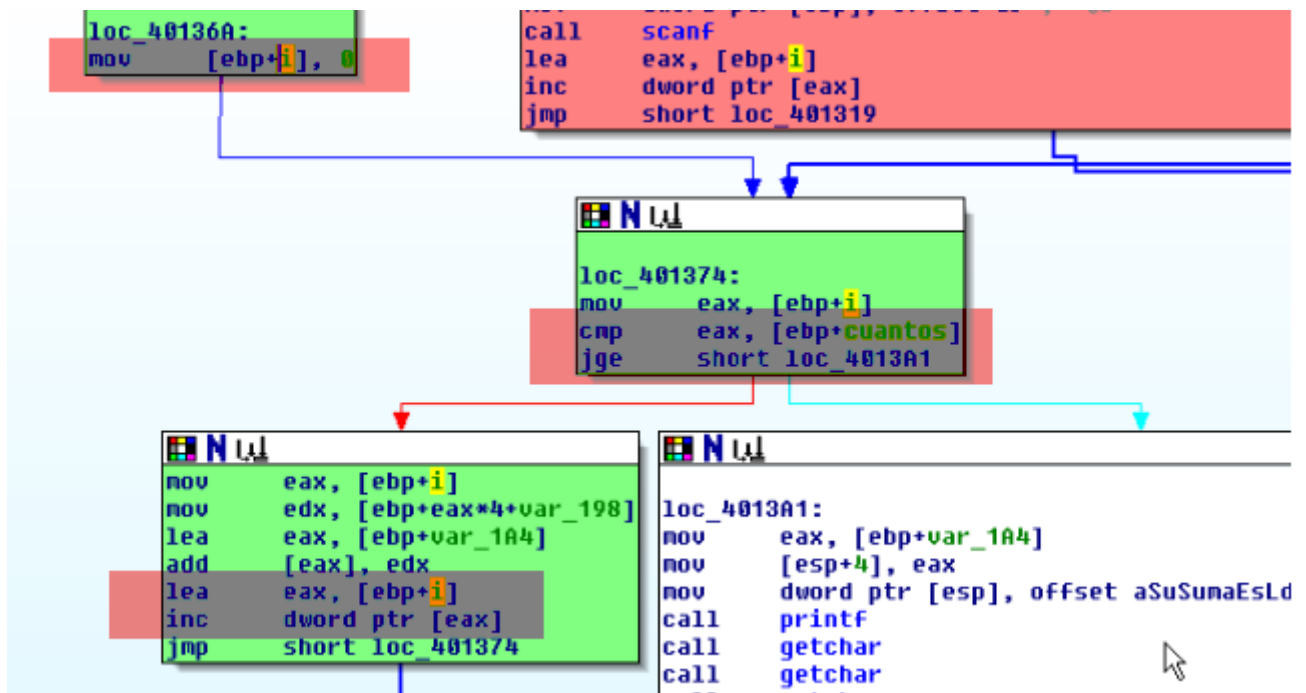
```

-0000001AF      db ? ; undefined
-0000001AE      db ? ; undefined
-0000001AD      db ? ; undefined
-0000001AC      dd ? ; undefined
-0000001A8      db ? ; undefined
-0000001A7      db ? ; undefined
-0000001A6      db ? ; undefined
-0000001A5      db ? ; undefined
-0000001A4      dd ? ; undefined
-0000001A0      i
-00000019C      cuantos
-000000198      var_198      dd 100 dup(?)
-000000008      db ? ; undefined
-000000007      db ? ; undefined
-000000006      db ? ; undefined
-000000005      db ? ; undefined
-000000004      db ? ; undefined
-000000003      db ? ; undefined
-000000002      db ? ; undefined
-000000001      db ? ; undefined
+000000000      s
+000000004      r
+000000008      argc
+00000000C      argv
+000000010      envp

```

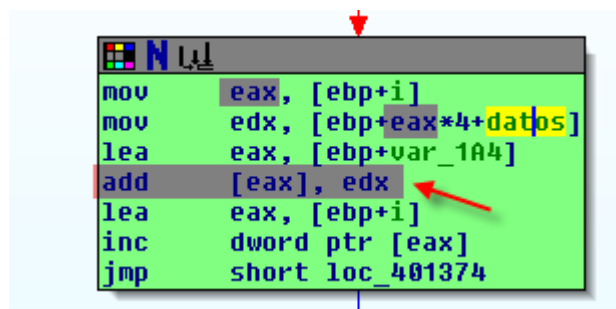
Luego hay otro for que usa el mismo contador **i** para lo cual lo vuelve a inicializar a cero.

Vemos el for pintado de verde, al inicio la inicializacion del contador **i**, luego la comparación de la condición de salida si **i** es mayor o igual que **cuantos**, y al final el incremento del contador y vuelta al inicio.



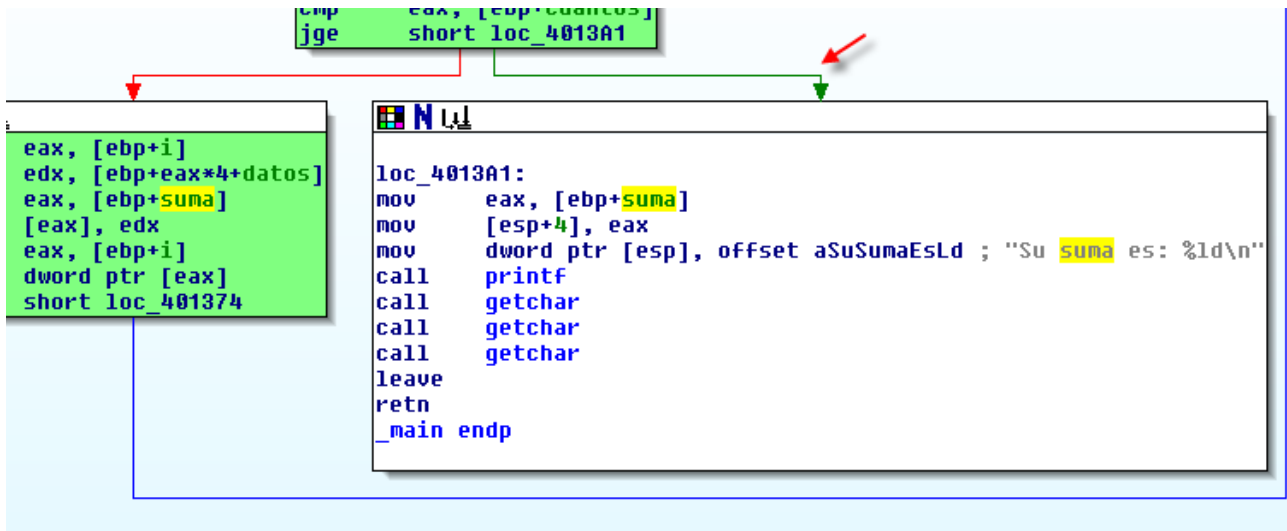
Vemos que dentro del for va tomando la **var_198** que es nuestro array que debemos renombrar como **datos** y va leyendo los valores de cada campo del mismo saltando de 4 en 4 usando **i** como contador y multiplicándolo por 4 aquí.

```
mov     edx, [ebp+eax*4+datos]
```



Así que en EDX tendremos el valor de cada campo al loopear y eso lo guarda en la variable **var_1a4** que sera la variable **suma** ya que con **lea** obtiene la dirección y le suma siempre EDX que es valor del campo actual, a la salida tendremos allí la suma de todos valores de los campos del array

Así que renombramos dicha variable **var_1a4** como **suma**.



Al salir del for imprime el valor de **suma**.

Ahora veremos un ejemplo de un programa que es similar, vemos que el ejemplo anterior no es eficiente, ya que se puede sumar a medida que se lee y además pongamosle que tengamos que almacenar para realizar cálculos más complejos, en realidad reservamos 100 dwords de memoria para sumar por ahí dos o tres números lo cual es un desperdicio, veamos como se hace reservando la memoria justa usando los nuevos conceptos.

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

```
main() {
    int* datos;    /* Necesitaremos espacio para varios numeros */
    int cuantos;   /* Preguntaremos cuantos desea introducir */
    int i;         /* Para bucles */
    long suma=0;   /* La suma, claro */

    do {
        printf("Cuantos numeros desea sumar? ");
        scanf("%d", &cuantos);
        datos = (int *) malloc (cuantos * sizeof(int));
        if (datos == NULL) /* Si no hay espacio, avisamos */
            printf("No caben tantos datos en memoria.");
    } while (datos == NULL); /* Si pide demasiado, no le dejamos */

    /* Pedimos y almacenamos los datos */
    for (i=0; i<cuantos; i++) {
        printf("Introduzca el dato número %d: ", i+1);
        scanf("%d", &datos[i]);
    }

    /* Calculamos la suma */
    for (i=0; i<cuantos; i++)
        suma += *(datos+i);

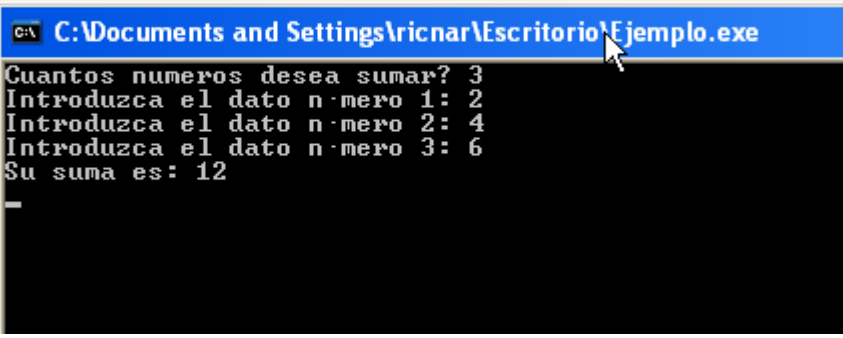
    printf("Su suma es: %ld\\n", suma);
    free(datos);
}
```

```
}
```

Si lo ejecutamos vemos que funciona en forma similar al anterior.

```
int i;           /* Para bucles */
long suma=0;     /* La suma, claro */
do {
    printf("Cu
    scanf("%d"
    datos = {i
    if (datos
        printf("
    } while (dat

/* Pedimos y
for (i=0; i<
```



Ahora veamos el cambio en el código fuente.

```
main() {
    int* datos; /* Necesitaremos espacio para varios numeros */
    int cuantos; /* Preguntaremos cuantos desea introducir */
    int i; /* Para bucles */
    long suma=0; /* La suma, claro */
```

Cuando declaramos **int* datos;** en vez de **int datos[100];** La diferencia es que ahora no reservamos 100 dwords fijos en la memoria sino que tenemos una variable que es un puntero a una zona que habrá **ints** pero no sabemos cuantos, los mismos podrán ser tantos como la memoria de la maquina nos lo permita.

Así que **datos** ahora es una variable puntero a **int**, y por ahora no esta inicializada cuando lo hagamos guardara una dirección o sea un puntero al int o a los ints., se debe ver bien esta diferencia no es lo mismo que una variable guarde datos que guarde un puntero adonde estarán los datos.

```
do {
    printf("Cuantos numeros desea sumar? ");
    scanf("%d", &cuantos);
    datos = (int *) malloc (cuantos * sizeof(int));
    if (datos == NULL) /* Si no hay espacio, avisamos */
        printf("No caben tantos datos en memoria.");
    } while (datos == NULL); /* Si pide demasiado, no le dejamos */
```

Aquí esta la clave del asunto, al igual que antes nos pregunta cuantos números vamos a sumar y lo guarda en la variable **cuantos**.

Ahora realizamos la cuenta de cuanto espacio necesitaremos en la memoria según la cantidad de enteros a sumar, la misma sera la cantidad de enteros que esta en **cuantos** por el tamaño que ocupa en bytes cada int, lo que se halla haciendo **sizeof(int)** lo que nos devuelve 4.

```
cuantos * sizeof(int)
```

Así que esto nos dará el tamaño en bytes que necesitamos en memoria, el cual se pasa a la función malloc que reserva dicho tamaño y nos devuelve un puntero al mismo.

```
datos = (int *) malloc (cuantos * sizeof(int));
```

Como datos es un int * conviene castear el resultado de malloc ya que realmente devuelve un puntero a la memoria que acaba de reservar, pero es un número, así que como vimos en la parte de casting forzamos a que el programa lo interprete como int * de forma de que no haya problemas de asignación en datos y nos de error al compilar.

CamisasBotasZapatillasBoisaLentesCarteras

malloc

funcion

<stdlib.h>

```
void * malloc ( size_t size );
```

Allocate memory block

Allocates a block of *size* bytes of memory, returning a pointer to the beginning of the block.

The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.

Parameters

size

Size of the memory block, in bytes.

Return Value

On success, a pointer to the memory block allocated by the function.

The type of this pointer is always void*, which can be cast to the desired type of data pointer in order to be dereferenceable. If the function failed to allocate the requested block of memory, a null pointer is returned.

El resto del programa es similar:

```
/* Pedimos y almacenamos los datos */
for (i=0; i<cuantos; i++) {
    printf("Introduzca el dato número %d: ", i+1);
    scanf("%d", &datos[i]);
}
```

```
/* Calculamos la suma */
for (i=0; i<cuantos; i++)
    suma += *(datos+i);
```

Vemos primero que nada, que en este caso la función **scanf** no necesita **&** delante de donde guardara los datos, sabemos que ello hacia que se halle la dirección a la variable donde se guardaran los datos, en este caso la misma ya es un puntero así que no necesita **&**.

Lo siguiente diferencia es que antes en un array recorríamos los campos del mismo como **datos[0]**, **datos[1]** y eso iba saltando entre los mismos, aquí sera ***datos**, el segundo ***(datos+1)**, el tercero será ***(datos+2)** y así en adelante. Por eso, donde antes hacíamos **suma += datos[i]**; ahora usamos **suma += *(datos+i)**; el significado es que siempre sera un puntero e ira recorriendo la memoria apuntando al dato actual, sumándole valores constantes al puntero inicial, con lo cual se obtiene un nuevo puntero.

```
/* Calculamos la suma */
for (i=0; i<cuantos; i++)
    suma += *(datos+i);

printf("Su suma es: %ld\n", suma);
free(datos);
getchar();
getchar();
getchar();
}
```

T

También aparece una llamada a **free** que es lo contrario de **malloc**, liberara la zona reservada de memoria para que se pueda disponer de ella, ya que no la usamos mas.

free

function

void free (void * ptr);

<stdlib.h>

Deallocate space in memory

A block of memory previously allocated using a call to **malloc**, **calloc** or **realloc** is deallocated, making it available again for further allocations.

Notice that this function leaves the value of *ptr* unchanged, hence it still points to the same (now invalid) location, and not to the null pointer.

Parameters

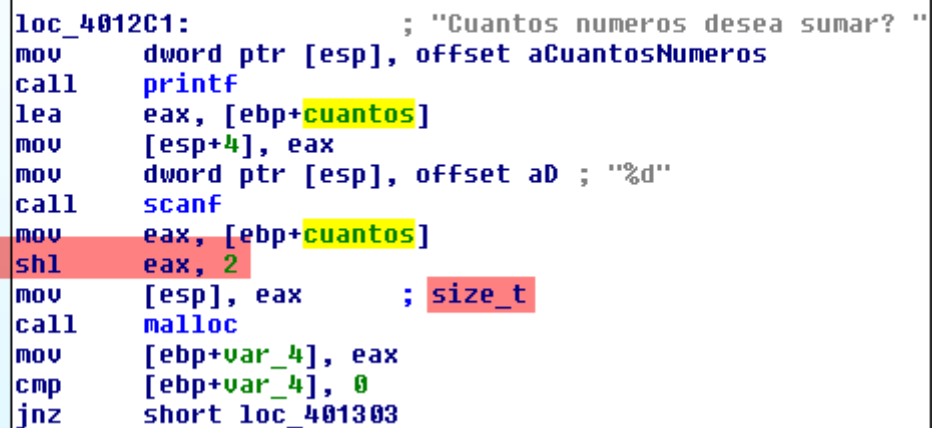
ptr

Pointer to a memory block previously allocated with **malloc**, **calloc** or **realloc** to be deallocated. If a null pointer is passed as argument, no action occurs.

Return Value

(none)

Vemos el programa en el IDA a ver las diferencias.

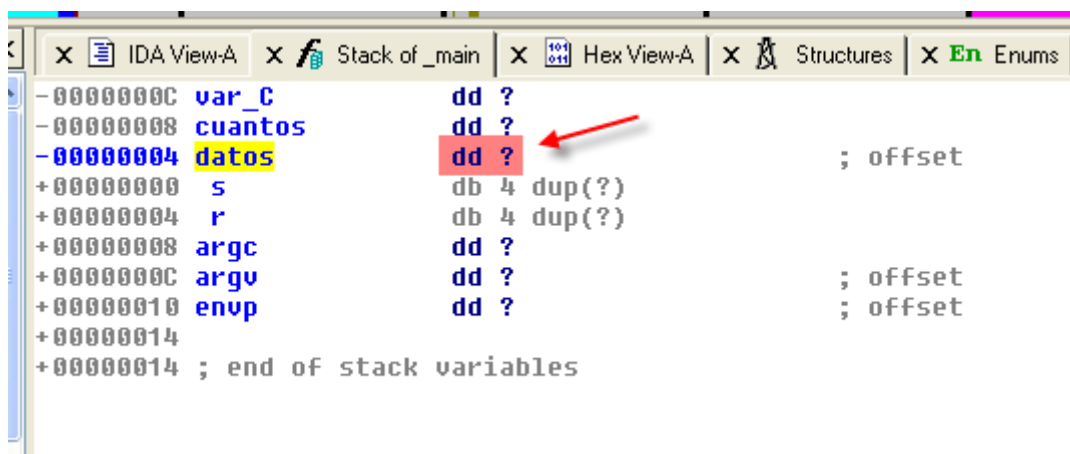


```

loc_4012C1:                                ; "Cuantos numeros desea sumar? "
mov     dword ptr [esp], offset aCuantosNumeros
call    printf
lea     eax, [ebp+cuantos]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf
mov     eax, [ebp+cuantos]
shl     eax, 2
mov     [esp], eax ; size_t
call    malloc
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jnz     short loc_401303

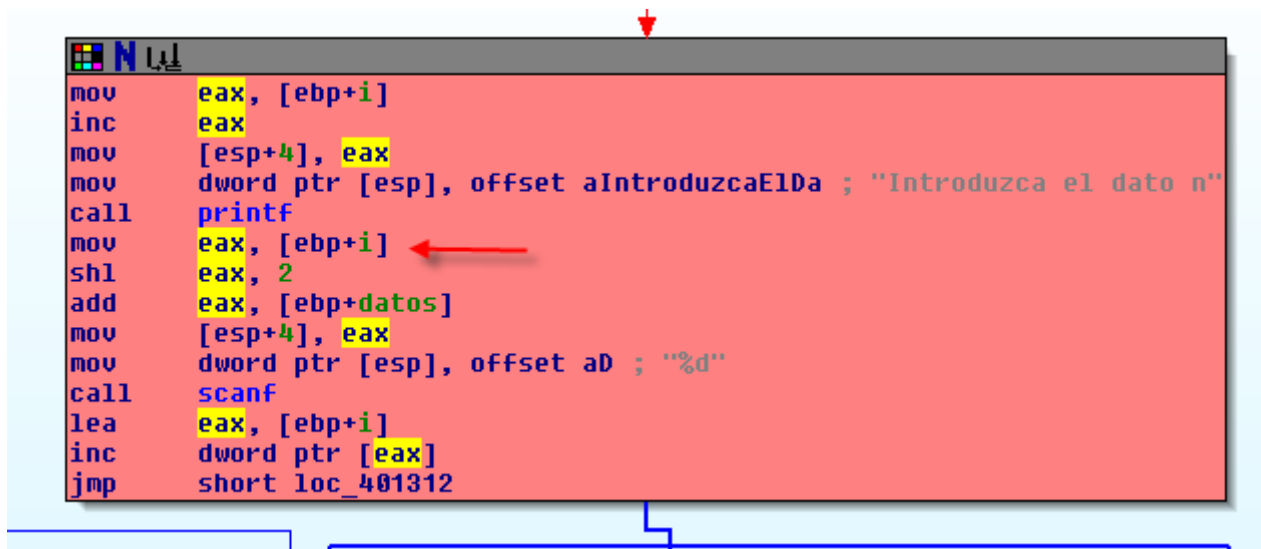
```

Vemos que una vez que guarda en **cuantos** la cantidad de números a sumar, lo multiplica por 4 para hallar el tamaño de bytes a reservar en la memoria, y eso lo pasa como argumento a **malloc** que nos devolverá el puntero a la zona reservada de dicho tamaño, el puntero lo guarda en **var_4** que es nuestro puntero **int *** llamado **datos** así que lo renombramos, aquí no hay desperdicio de memoria pues el tamaño de un puntero es un **dword** que es lo que vemos que reservo para guardar ese valor.



Address	Variable	Type	Comment
-00000000	var_C	dd ?	
-00000008	cuantos	dd ?	
-00000004	datos	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		

Aquí vemos como maneja el puntero cuando guarda los datos dentro del for



```
mov    eax, [ebp+i]
inc    eax
mov    [esp+4], eax
mov    dword ptr [esp], offset aIntroduzcaElDa ; "Introduzca el dato n"
call   printf
mov    eax, [ebp+i]
shl    eax, 2
add    eax, [ebp+datos]
mov    [esp+4], eax
mov    dword ptr [esp], offset aD ; "%d"
call   scanf
lea    eax, [ebp+i]
inc    dword ptr [eax]
jmp     short loc_401312
```

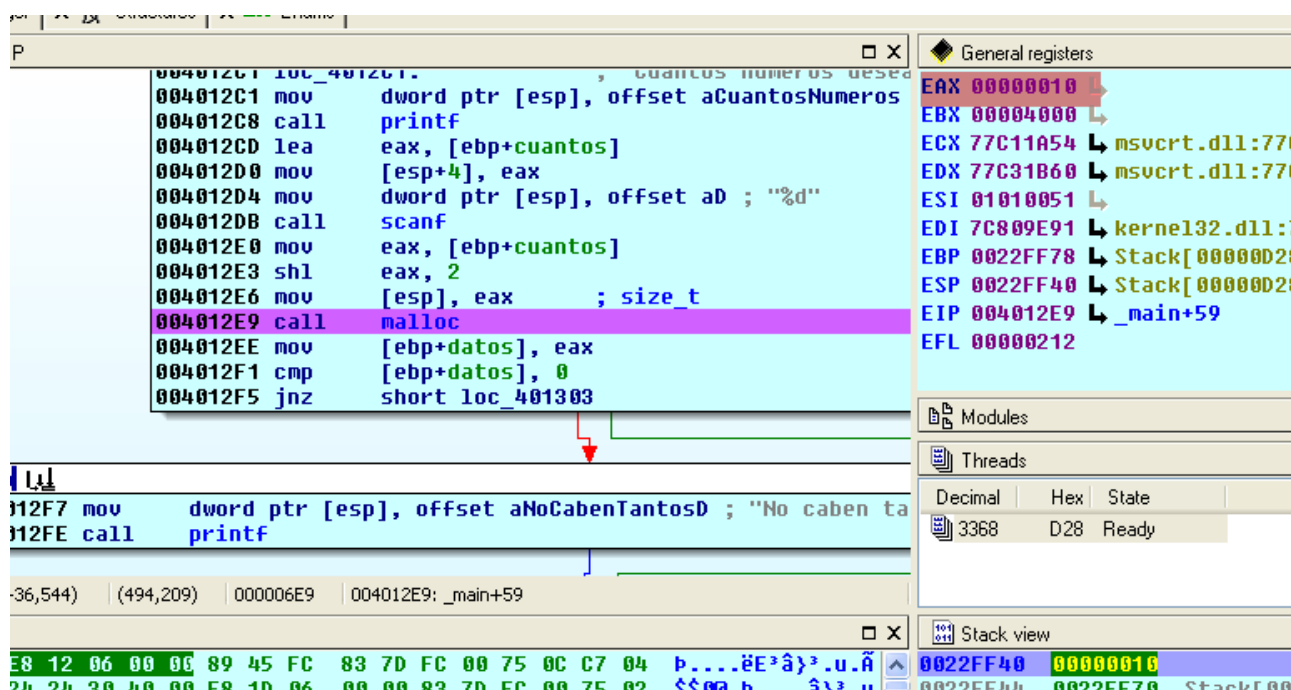
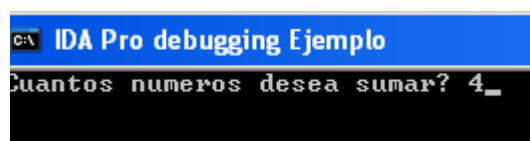
La variable **i** sera igual que antes, el contador, lo multiplica por 4 y lo suma al puntero **datos**, de esa forma cuando i valga 0, el puntero **datos** apuntara al inicio de la zona reservada, luego cuando i valga 1, sera por lo tanto $i*4$, y eso se le sumara al puntero al inicio con lo cual obtendremos un nuevo puntero al segundo int ya que vamos barriendo de 4 en cuatro, sumándole al puntero original, para que quede mas claro lo debuggaremos.

```

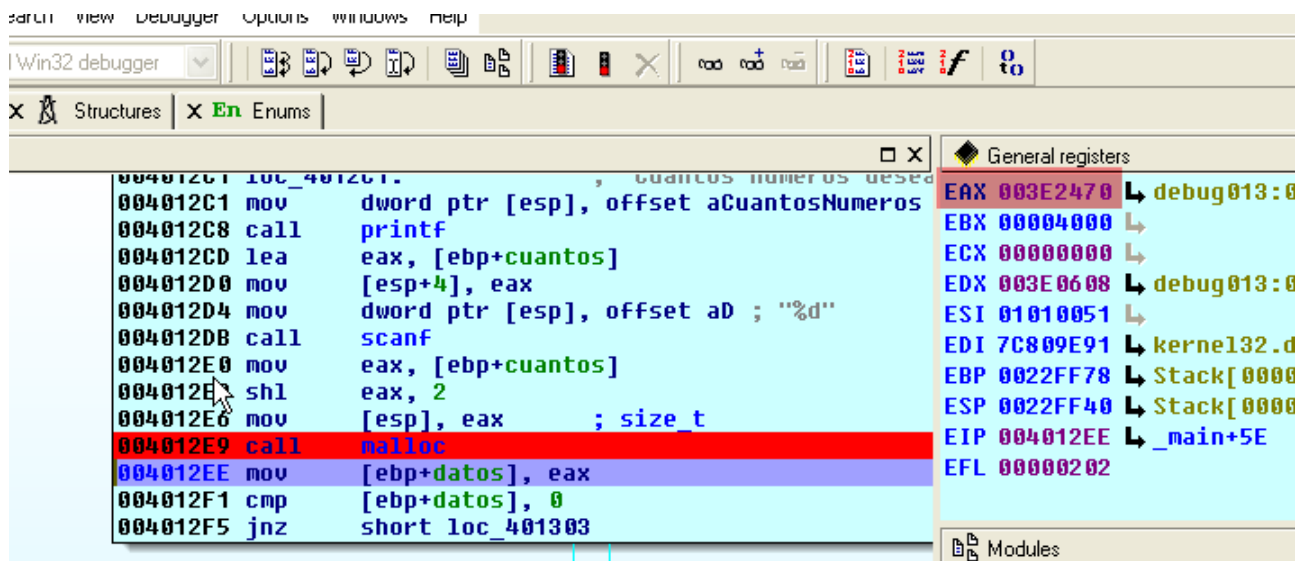
loc_4012C1:                ; "Cuantos numeros desea sumar? "
mov     dword ptr [esp], offset aCuantosNumeros
call    printf
lea     eax, [ebp+cuantos]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    scanf
mov     eax, [ebp+cuantos]
shl     eax, 2
mov     [esp], eax          ; size_t
call    malloc
mov     [ebp+datos], eax

```

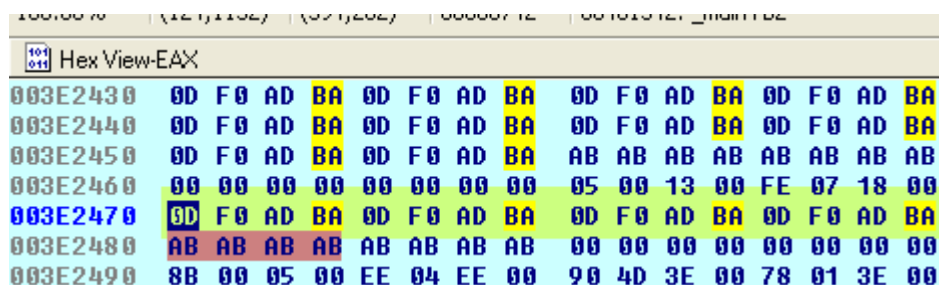
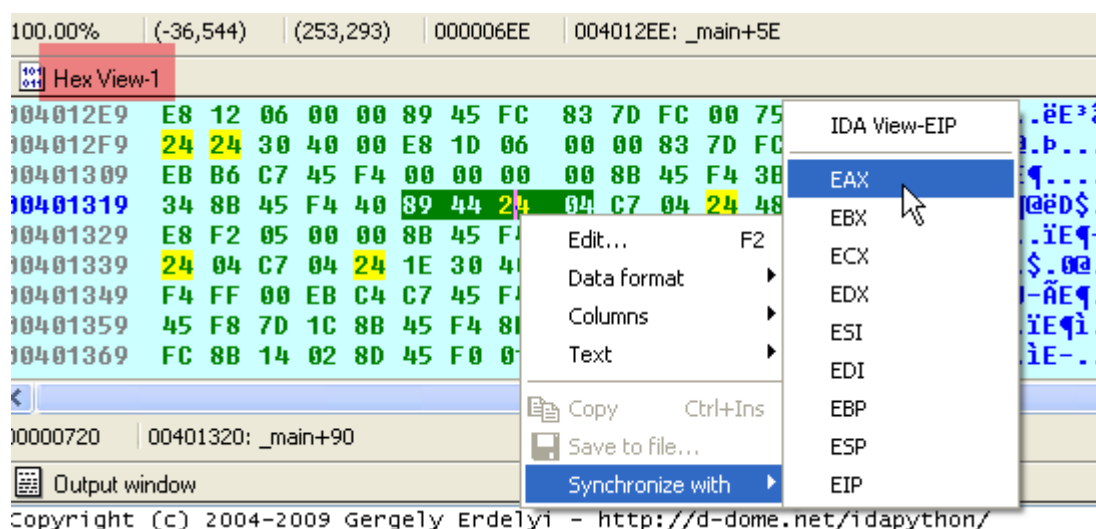
Arranco el programa y me pide cuanto números sumare le pongo como ejemplo 4, luego parara en el llamado a **malloc**.



El 4 que ingrese lo guarda en **cuantos** y lo multiplica por 4 para hallar el tamaño a reservar que sera 16, o sea 10 hexa como vemos en EAX y en el stack si hacemos JMP TO ESP, ese sera el tamaño de la memoria a reservar, pasemos el call malloc con f8.



Allí vemos que en EAX nos devuelve un puntero adonde reservo la memoria, la dirección puede cambiar de maquina en maquina pero si miro dicha zona.

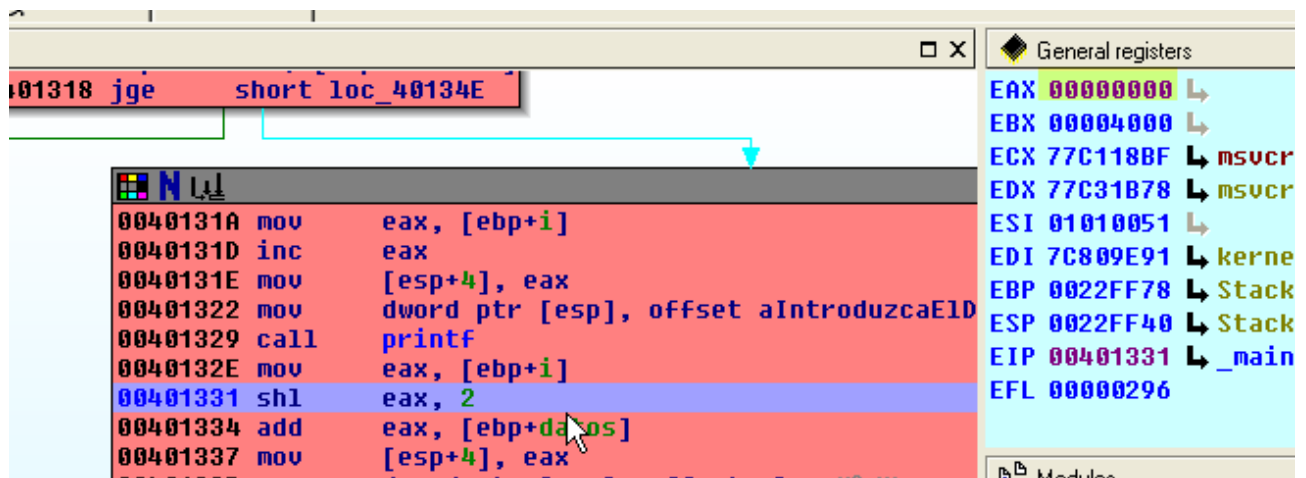


Allí hay lugar para 4 dwords y esta relleno con **0D F0 AD BA** (bad food) porque el programa esta siendo debuggeado para que veamos bien la zona reservada, sino serian ceros, jeje y donde termina la zona reservada vemos **ABABABAB** (también solo en modo debug).

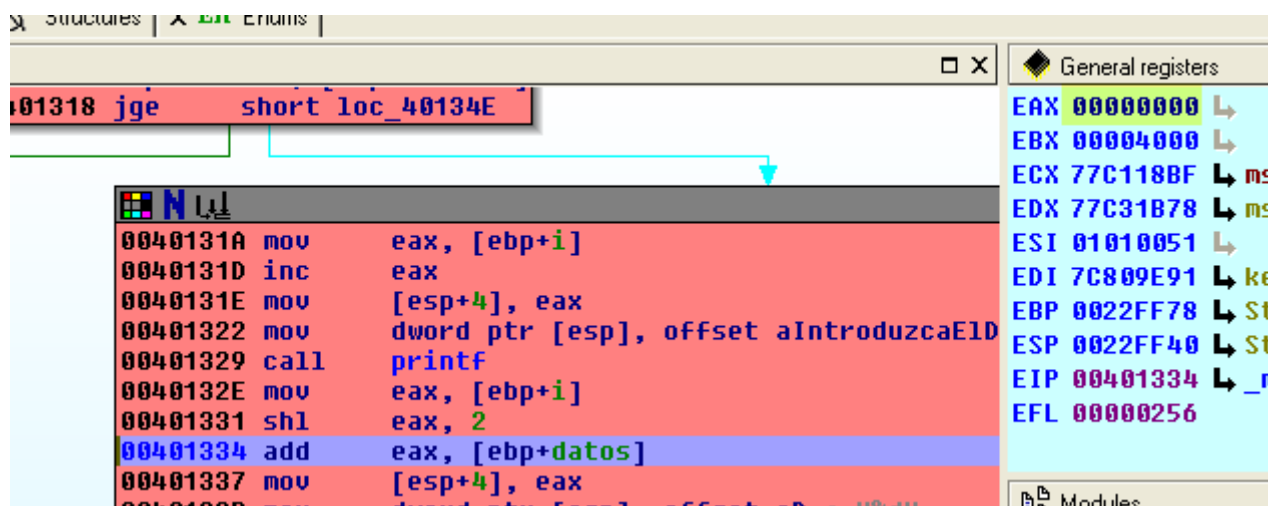
Si traceamos y llegamos hasta dentro del **for** aquí

```
0040131E mov     [esp+4], eax
00401322 mov     dword ptr [esp], offset aIntroduzcaEl
00401329 call    printf
0040132E mov     eax, [ebp+i]
00401331 shl     eax, 2
00401334 add     eax, [ebp+datos]
00401337 mov     [esp+4], eax
0040133B mov     dword ptr [esp], offset aD ; "%d"
00401342 call    scanf
00401347 jmp     [ebp+i]
```

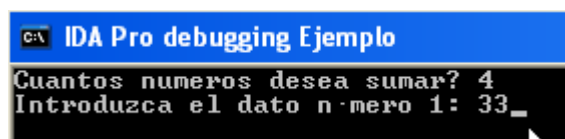
Vemos que **i** vale 0 y se mueve a **EAX**.



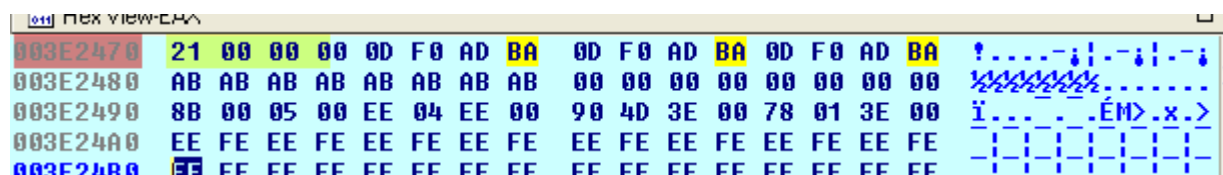
Luego con el **SHL EAX, 2** lo multiplica por 4, el resultado sera **cero**.



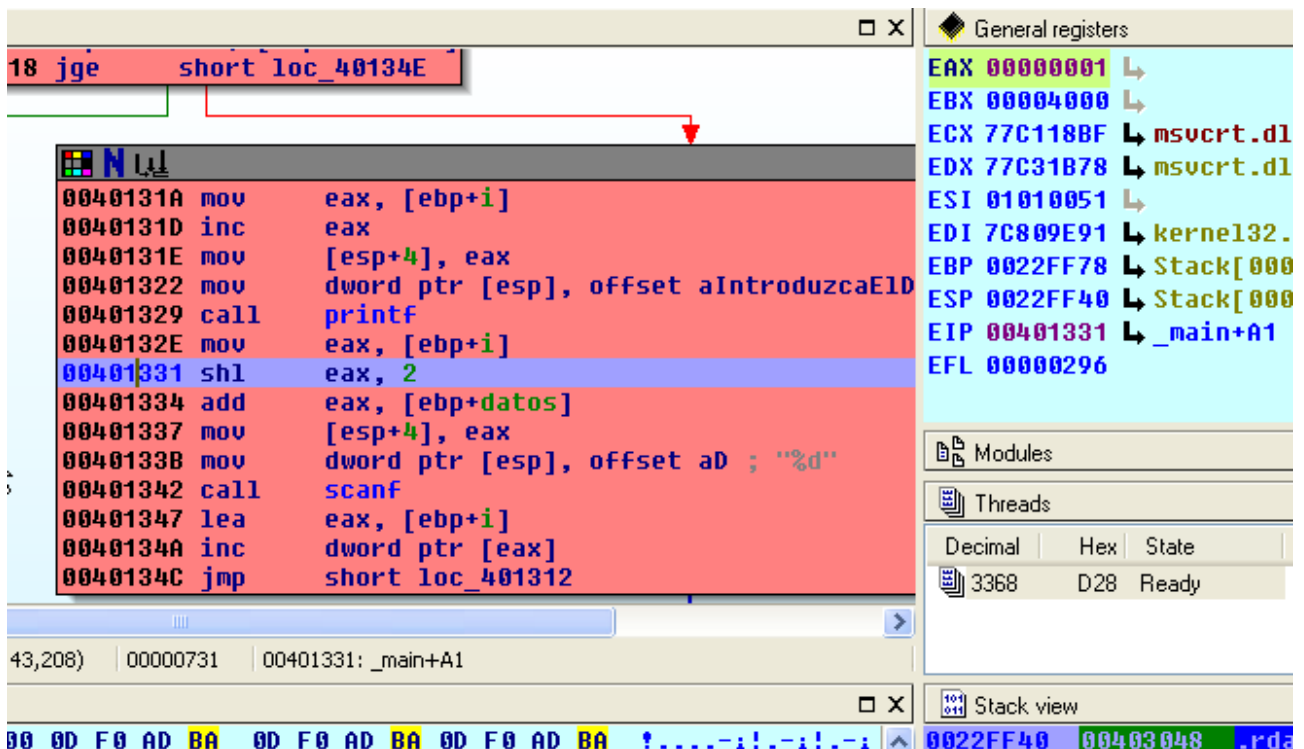
Luego toma nuestro puntero **datos** y le suma el resultado obtenido o sea cero, recordemos que es un **int** * así que al sumarle una constante dará un puntero en este caso al sumarle cero dará el mismo valor que apunta al primer **int** de la zona de memoria reservada donde guardara lo que tipeamos, apretamos f8 hasta que pasamos el **scanf**.



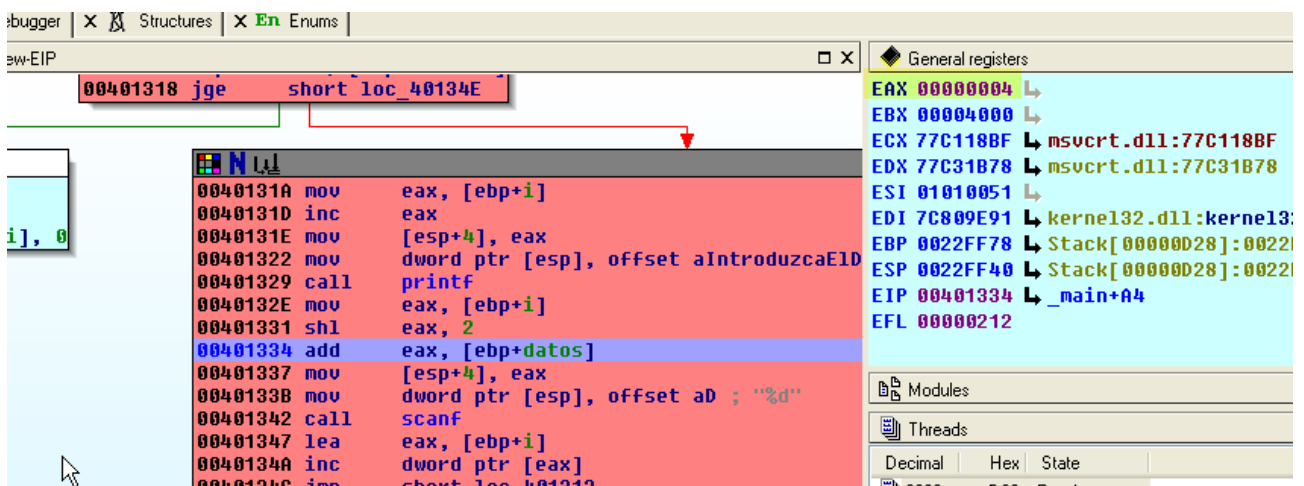
Allí guardara el primer valor 21 hexa que es 33 decimal.



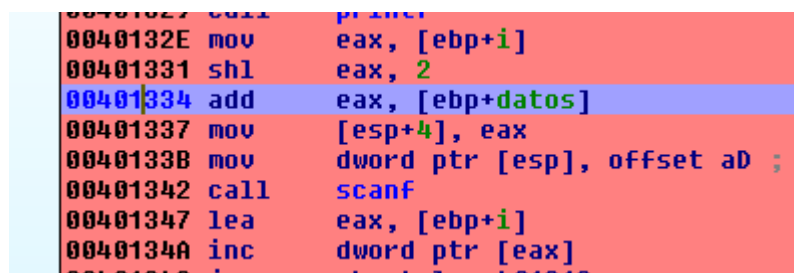
Si repetimos traceando con f8 y llegamos al mismo bloque nuevamente.



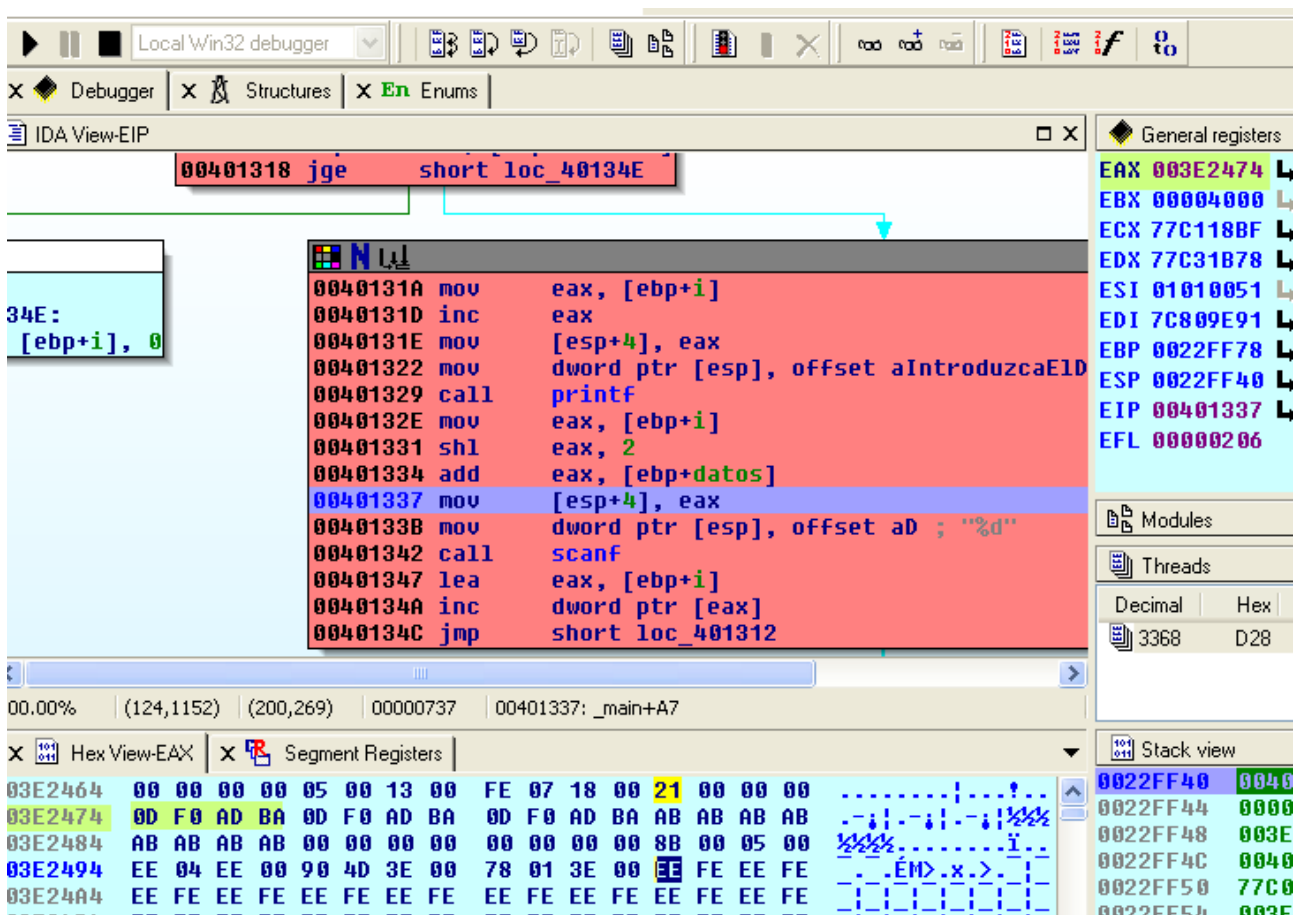
Ahora el contador `i` vale 1, lo multiplica por 4 con el `SHL` lo que dara 4.



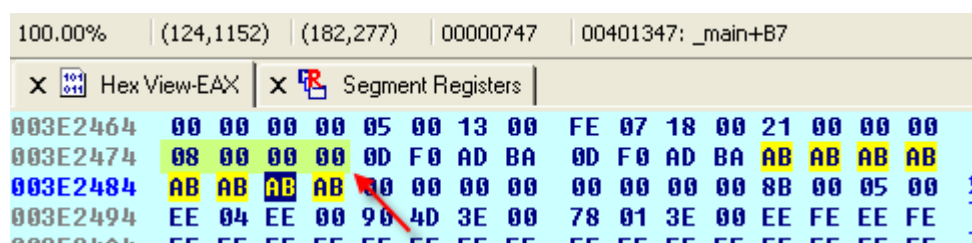
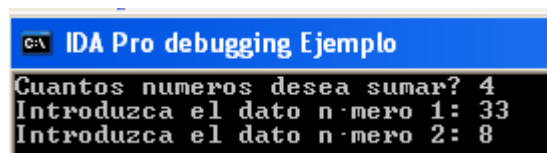
Lo suma al puntero que tenemos en **datos**.



Quedando en `EAX` un nuevo puntero al segundo `int` a guardar.



Si pasamos el **scanf** con f8 veremos como guarda allí el segundo valor.



Así que lo que se indexa aquí es el puntero, y se le van sumando en este caso de 4 en 4 para ir guardando en la zona reservada en forma correcta, lo mismo cuando lo lea, tomara el puntero a dicha zona y le ira sumando cuatro para leer los valores guardados, si ponemos un BP en el call a free al final luego de guardar los cuatro valores y realizar la suma.

```

0040138C mov     eax, [ebp+datos]
0040138F mov     [esp], eax      ; void *
00401392 call    free
00401397 call    getchar
0040139C call    getchar
004013A1 call    getchar
004013A6 leave
004013A7 retn
004013A7 _main endp

```

```

C:\> IDA Pro debugging Ejemplo
Cuantos numeros desea sumar? 4
Introduzca el dato n-mero 1: 33
Introduzca el dato n-mero 2: 8
Introduzca el dato n-mero 3: 2
Introduzca el dato n-mero 4: 5

```

Al llegar al free se le pasa como argumento el puntero a la zona reservada

General registers	
EAX	003E2470 → deb
EBX	00004000 →
ECX	77C118BF → msu
EDX	77C31B78 → msu
ESI	01010051 →
EDI	7C809E91 → ker
EBP	0022FF78 → Sta
ESP	0022FF40 → Sta
EIP	00401392 → _ma
EFL	00000296


```

00401379
00401379 loc_401379:
00401379 mov     eax, [ebp+var_10]
0040137C mov     [esp+4], eax
00401380 mov     dword ptr [esp], offset aSuSumaEsLd ; "
00401387 call    printf
0040138C mov     eax, [ebp+datos]
0040138F mov     [esp], eax      ; void *
00401392 call    free
00401397 call    getchar
0040139C call    getchar

```

Si lo pasamos con f8 vemos que dicha zona cambio y ya no tiene nuestros datos sino punteros, eso se estudiara mas adelante el mecanismo que tiene el sistema para saber como una zona esta libre para reservar o no.

[eax], edx	00401387 call print
eax, [ebp+i]	0040138C mov eax, [ebp+datos]
dword ptr [eax]	0040138F mov [esp], eax ; void
short loc_401355	00401392 call free
	00401397 call getchar
	0040139C call getchar
	004013A1 call getchar
	004013A6 leave
	004013A7 retn
	004013A7 _main endp
	004013A7

100.00% (206,1581) (326,267) 000007A6 004013A6: _main+116

Hex View-EAX

Segment Registers

003E2430	0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA
003E2440	0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA
003E2450	0D F0 AD BA 0D F0 AD BA AB AB AB AB AB AB AB AB
003E2460	00 00 00 00 00 00 00 00 90 00 13 00 FE 04 18 00
003E2470	90 4D 3E 00 78 01 3E 00 EE FE EE FE EE FE EE FE
003E2480	FF FF FF FF FF FF FF FF FF FF FF FF FF FF

Allí hay un ejemplo de punteros a resolver lo único molesto es que usa floats y eso a veces es un poco molesto de ver en IDA pero bueno es así la vida jeje.

Hasta la próxima parte
 Ricnar