

SEGUIMOS CON MAS SOBRE PUNTEROS

Cuando declaramos un dato como array, existe una similitud con un puntero, lo veremos en el ejemplo a continuacion:

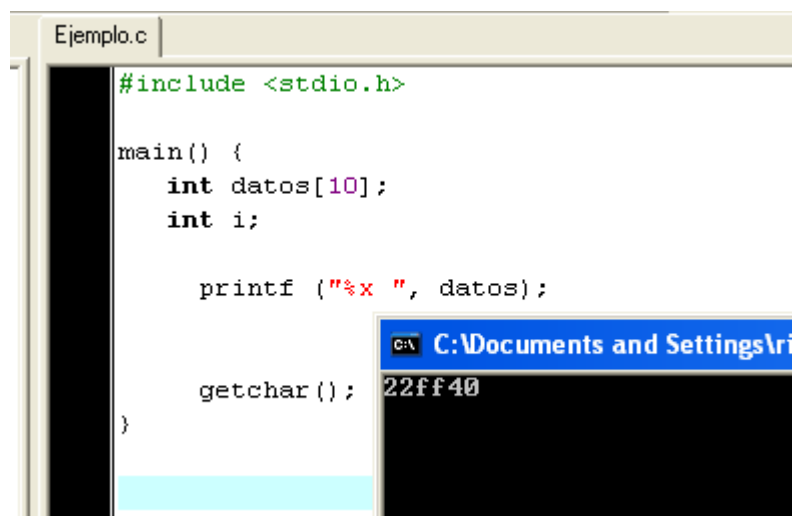
```
#include <stdio.h>
```

```
main() {  
    int datos[10];  
  
    datos[0]=5 ;  
    printf ("%d ", datos[0]);  
  
    getchar();  
}
```

Sabemos que para asignar valores a los campos del array normalmente usamos **datos[x]** , como en el ejemplo **datos[0]**, pero en si que pasa si imprimimos el valor de la variable **datos** solo sin subindice, realmente **datos** tiene algún valor?

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

```
main() {  
    int datos[10];  
    int i;  
  
    printf ("%x ", datos);  
  
    getchar();  
}
```



The screenshot shows a code editor window titled 'Ejemplo.c' containing the following C code:

```
#include <stdio.h>  
  
main() {  
    int datos[10];  
    int i;  
  
    printf ("%x ", datos);  
  
    getchar();  
}
```

Below the code editor, a terminal window displays the output of the program. The first line of output is the memory address '22ff40', which is the address of the first element of the array 'datos'.

Vemos que cuando declaramos un array, al utilizar el nombre solo sin subindices contiene la dirección de memoria donde se inicia el array, al cual se le pueden asignar y leer valores como si

fuera una variable puntero.

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

```
main() {  
    int datos[10];  
    int i;  
  
    printf ("%x\n", datos);  
  
    for (i=0; i<10; i++)  
        *(datos+i) = i*2;  
  
    for (i=0; i<10; i++)  
        printf ("%d ", *(datos+i));  
  
    getchar();  
}
```

Vemos que le asignamos valores a sus campos, tal cual **datos** fuera puntero, y los lee de la misma forma, la asignación directa también puede realizarse como puntero.

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

```
main() {  
    int datos[10];  
    int i;  
  
    printf ("%x\n", datos);  
  
    *(datos)= 20;  
  
    printf ("%d ", *(datos));  
  
    getchar();  
}
```

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

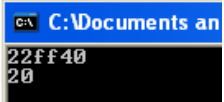
main() {
    int datos[10];
    int i;

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

    *(datos)= 20;

    printf ("%d ", *(datos));

    getchar();
}
```



Aquí lo asignamos como puntero y lo leemos en la forma tradicional

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

```
main() {
    int datos[10];
    int i;

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

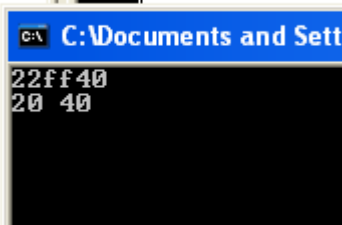
    *(datos)= 20;

    printf ("%d ", *(datos));

    *(datos+1)= 40;

    printf ("%d ", datos[1]);
    getchar();
}
```

```
#include <st
```



Punteros y estructuras (choreado de Cabanes)

Igual que creamos punteros a cualquier tipo de datos básico, le reservamos memoria con “malloc” cuando necesitamos usarlo y lo liberamos con “free” cuando terminamos de utilizarlo, lo mismo podemos hacer si se trata de un tipo de datos no tan sencillo, como un “struct”.

Eso sí, la forma de acceder a los campos del **struct** cambiará ligeramente. Para un dato que sea un número entero, ya sabemos que lo declararíamos con **int *n** y cambiaríamos su valor haciendo algo

como ***n=2**, de modo que para un **struct** podríamos esperar que se hiciera algo como ***persona.edad = 20**. Pero esa no es la sintaxis correcta: deberemos utilizar el nombre de la variable y el del campo, con una flecha (->) entremedio, así: **persona->edad = 20**. Vamos a verlo con un par de ejemplos para que se aclare bien:

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

```
main() {
```

```
    /* Primero definimos nuestro tipo de datos */
```

```
    struct datosPersona {
```

```
        char nombre[30];
```

```
        char email[25];
```

```
        int edad;
```

```
    };
```

```
    /* La primera persona será estática */
```

```
    struct datosPersona persona1;
```

```
    /* Damos valores a la persona estática */
```

```
    strcpy(persona1.nombre, "Juan");
```

```
    strcpy(persona1.email, "j@j.j");
```

```
    persona1.edad = 20;
```

```
    printf("Primera persona: %s, %s, con edad %d\n",  
        persona1.nombre, persona1.email, persona1.edad);
```

```
    getchar();
```

```
}
```

Este es el caso clásico que ya habíamos visto en partes anteriores, la persona, esta instanciada como variable estática en el stack, veámoslo en IDA así vemos la diferencia con el caso dinámico que veremos luego.

```

var_4C= dword ptr -4Ch
var_48= byte ptr -48h
var_10= dword ptr -10h
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 68h
and     esp, 0FFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_4C], eax
mov     eax, [ebp+var_4C]
call    ___chkstk
call    main
mov     dword ptr [esp+4], offset aJuan ; "Juan"
lea     eax, [ebp+var_48]
mov     [esp], eax ; char *
call    strcpy
mov     dword ptr [esp+4], offset aJ@j_j ; "j@j.j"
lea     eax, [ebp+var_48]

```

Vemos la línea roja que separa lo agregado por el compilador, asimismo la variable **var_4c** es creada por el mismo, la primera variable nuestra es la **var_48**

```

mov     eax, [ebp+var_4C]
call    ___chkstk
call    main
mov     dword ptr [esp+4], offset aJuan ; "Juan"
lea     eax, [ebp+var_48]
mov     [esp], eax ; char *
call    strcpy
mov     dword ptr [esp+4], offset aJ@j_j ; "j@j.j"
lea     eax, [ebp+var_48]
add     eax, 1Eh
mov     [esp], eax ; char *
call    strcpy
mov     [ebp+var_10], 14h
mov     eax, [ebp+var_10]
mov     [esp+0Ch], eax

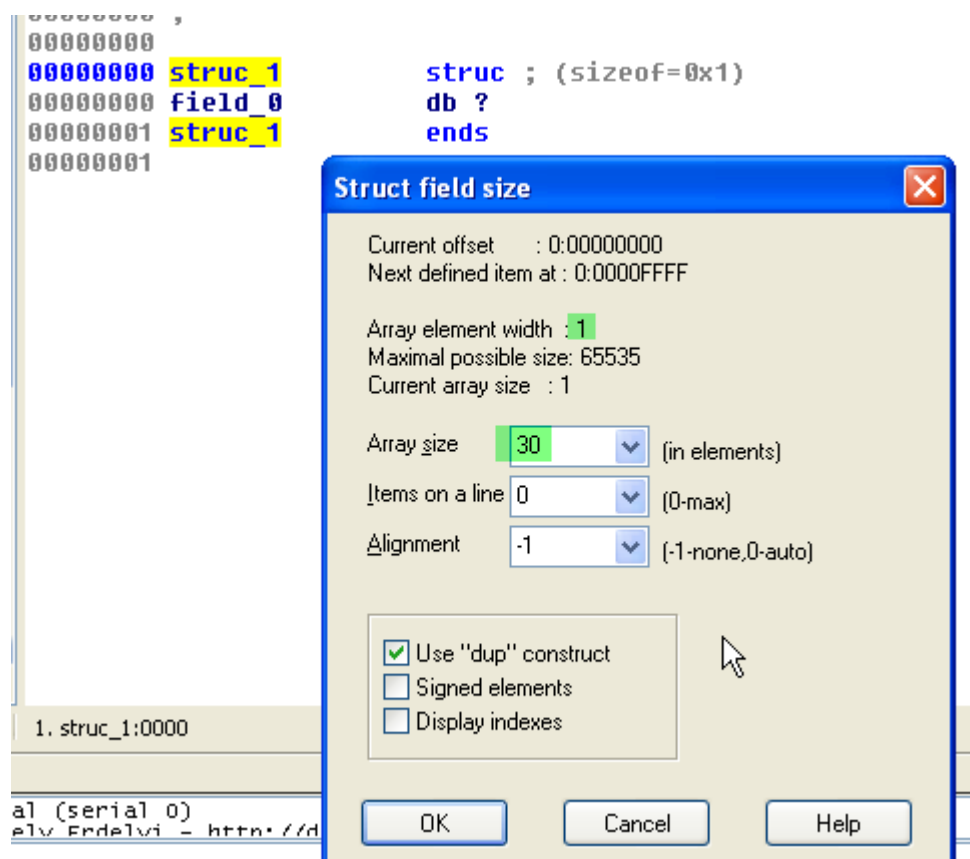
```

Allí vemos cuando inicializa la variable **var_48**, en celeste vemos que copia la string **“Juan”**, luego vemos en amarillo que se suma **1eh** desde **var_48** y copia otra string con el mail, esto nos podría hacer pensar que es un array de strings, pero luego vemos la **var_10** a la cual se le asigna el valor 14, y pensamos que todo esto es una estructura ya que si vemos las variables.

-00000040		db ; , undefined
-0000004C	var_4C	dd ?
-00000048	var_48	db ?
-00000047		db ? ; undefined
-00000046		db ? ; undefined
-00000045		db ? ; undefined
-00000044		db ? ; undefined
-00000043		db ? ; undefined
-00000042		db ? ; undefined
-00000041		db ? ; undefined
-00000040		db ? ; undefined
-0000003F		db ? ; undefined
-0000003E		db ? ; undefined
-0000003D		db ? ; undefined
-0000003C		db ? ; undefined
-0000003B		db ? ; undefined
-0000003A		db ? ; undefined
-00000039		db ? ; undefined
-00000038		db ? ; undefined
-00000037		db ? ; undefined
-00000036		db ? ; undefined
-00000035		db ? ; undefined
-00000034		db ? ; undefined
-00000033		db ? ; undefined
-00000032		db ? ; undefined
-00000031		db ? ; undefined
-00000030		db ? ; undefined

Ya ver espacio vacío no definido entre variables, es sospechoso de estructura o array estatico, una asignación de un valor en un espacio intermedio no definido, entre las variables como aquí en **var_48 + 1e**, es también sospechoso de array o estructura, pero al seguir hacia abajo y ver que donde asigna el valor a la **var_10** esta consecutiva al espacio no definido y es de un tipo diferente podemos concluir que no es un array pues tiene campos de diferentes tipos, así que estamos en presencia de una estructura.

Así que vayamos a la pestaña de creación de estructuras, sabemos que para escribir la segunda string le suma **1eh** así que el largo de la primera sera 30, creamos la estructura y agregamos con asterisco un array.



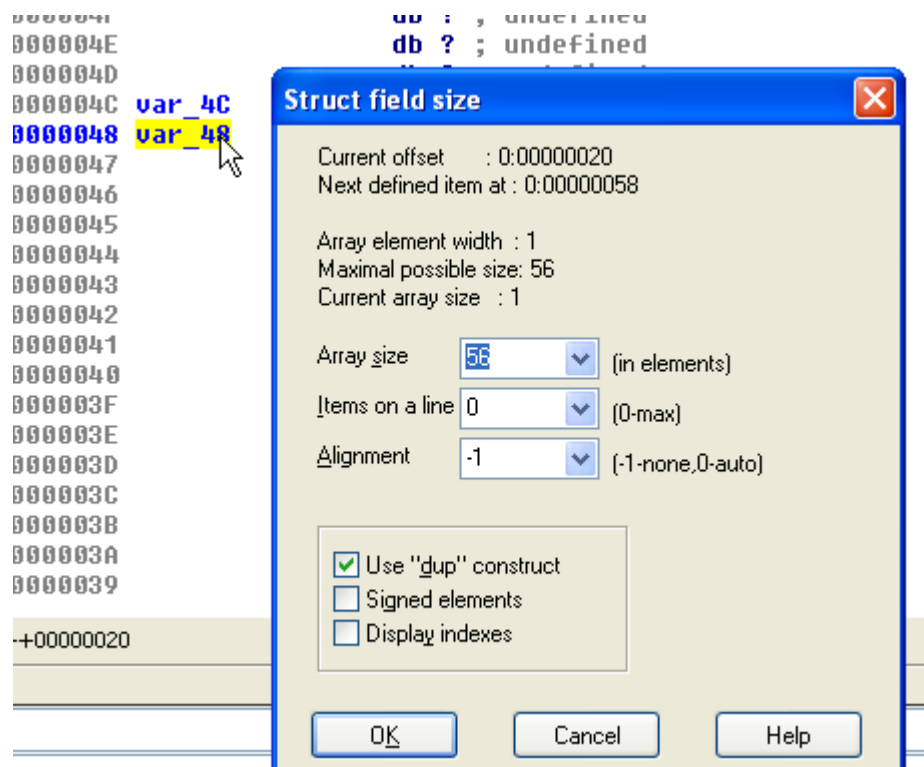
Como tomo como char o sea un byte el inicio, luego al colocar 30 bytes de largo, pues creara el array de caracteres correspondiente, le pongo el nombre correcto.

```

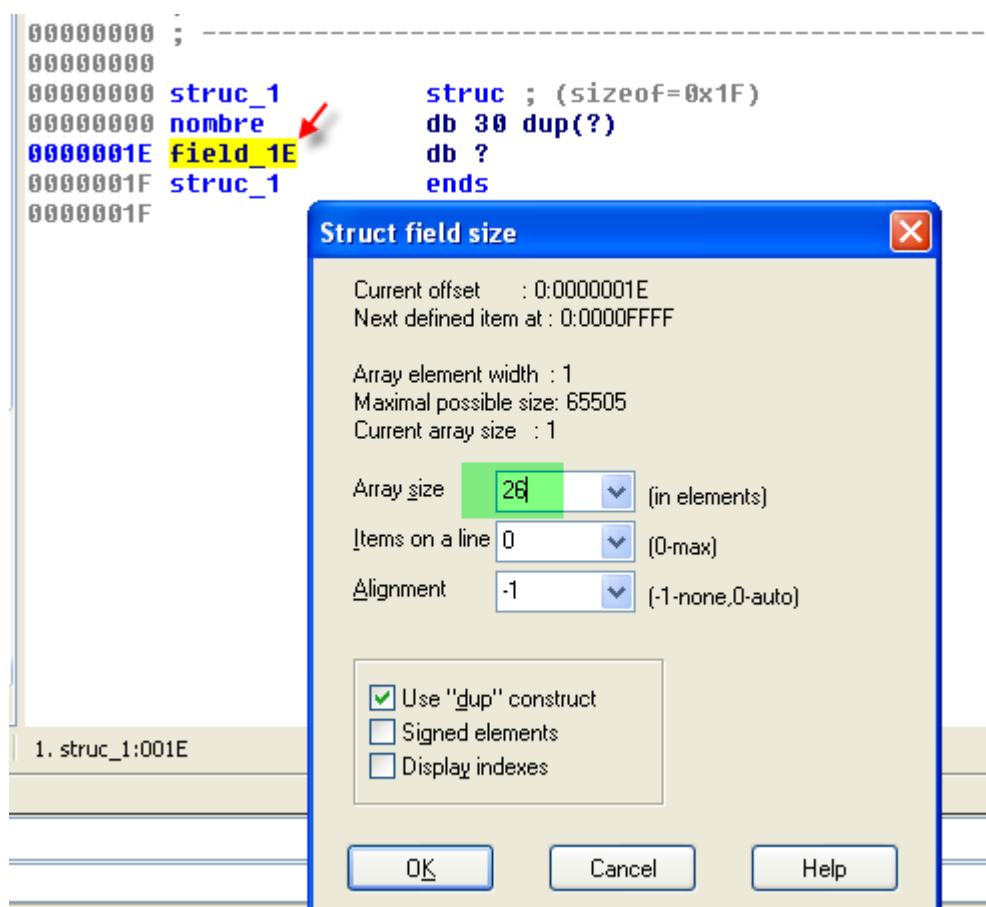
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; -----
00000000
00000000 struc 1      struc ; (sizeof=0x1E)
00000000 nombre      db 30 dup(?)
0000001E struc_1
0000001E ends

```

Vuelvo a las variables y aprieto asterisco en la primera solo para que me diga el espacio en bytes que hay hasta la **var_10**, no creo nada.



Veo que hay 56 bytes, así que si le resto los 30 del primer campo, quedarán 26 bytes, así que cancelo la ventana y vuelvo adonde están las estructuras y creo un segundo campo, que sea un array de 26 hexa de largo.



Vemos que en el mismo nombre nos indica que esta en la posición **1eh** desde el inicio, así que lo creamos parece que vamos bien.

```

00000000 ; D/A/*      : create structure member (data/.
00000000 ; N          : rename structure or structure |
00000000 ; U          : delete structure member
00000000 ; -----
00000000
00000000 struc_1      struc ; (sizeof=0x38)
00000000 nombre       db 30 dup(?)
0000001E email        db 26 dup(?)
00000038 struc_1      ends
00000038

```

Ahora agregamos el tercer campo que es un int que debería coincidir con la **var_10**.

```

00000000
00000000 struc_1      struc ; (sizeof=0x3C)
00000000 nombre       db 30 dup(?)
0000001E email        db 26 dup(?)
00000038 edad         dd ?
0000003C struc_1      ends
0000003C

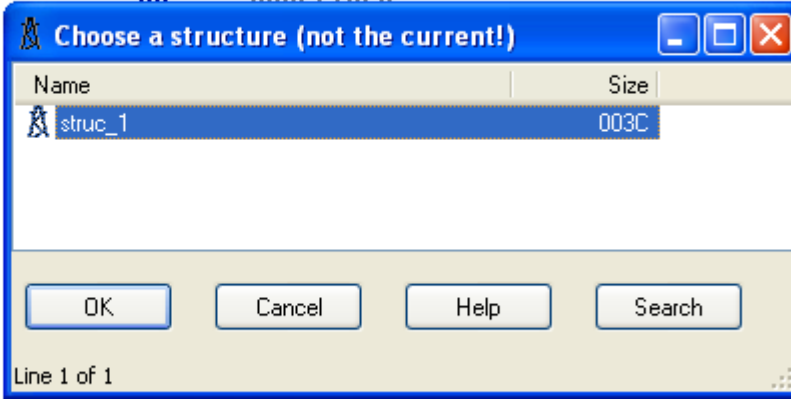
```

Allí tenemos armada la estructura así que ahora asignemos a la variable **var_48**, apretando ALT mas Q sobre la misma.

```

-0000004F          db ? ; undefined
-0000004E          db ? ; undefined
-0000004D          db ? ; undefined
-0000004C var_4C
-00000048 var_48
-00000047
-00000046
-00000045
-00000044
-00000043
-00000042
-00000041
-00000040
-0000003F
-0000003E
-0000003D
-0000003C
-0000003B

```



Nos queda renombrar la estructura pongamosle el nombre original ya que lo sabemos.

```

00000000 ; N      : rename structure or struct
00000000 ; U      : delete structure member
00000000 ; -----
00000000
00000000 datosPersona      struc ; (sizeof=0x3C)
00000000 nombre          db 30 dup(?)
0000001E email           db 26 dup(?)
00000038 edad           dd ?
0000003C datosPersona      ends
0000003C

```

Así que ahora **var_48** es del tipo **datosPersona**, podemos renombrar la variable **var_48** ya que es una instancia de **datosPersona** y en nuestro código la llamábamos **persona1**, así que lo hacemos.

```

; int __cdecl main(int argc, const cha
_main proc near
var_4C= dword ptr -4Ch
var_48= datosPersona ptr -48h
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 68h
and     esp, 0FFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh

```

Renombrada

```

var_4C= dword ptr -4Ch
persona1= datosPersona ptr -48h
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

```

Por supuesto **persona1** esta creada en el stack en forma estática usando el lugar que predeterminamos en el mismo para cada campo.

```

004012A7 shl     eax, 4
004012AA mov     [ebp+var_4C], eax
004012AD mov     eax, [ebp+var_4C]
004012B0 call    ___chkstk
004012B5 call    main
004012BA mov     dword ptr [esp+4], offset aJuan ; "Juan"
004012C2 lea     eax, [ebp+persona1]
004012C5 mov     [esp], eax ; char *
004012C8 call    strcpy
004012CD mov     dword ptr [esp+4], offset aJ@j_j ; "j@j.j"
004012D5 lea     eax, [ebp+persona1]
004012D8 add     eax, 1Eh
004012DB mov     [esp], eax ; char *
004012DE call    strcpy
004012E3 mov     [ebp+persona1.edad], 14h
004012EA mov     eax, [ebp+persona1.edad]
004012ED mov     [esp+0Ch], eax
004012F1 lea     eax, [ebp+persona1]
004012F4 add     eax, 1Eh
004012F7 mov     [esp+8], eax
004012FB lea     eax, [ebp+persona1]
004012FE mov     [esp+4], eax
00401302 mov     dword ptr [esp], offset aPrimeraPersona ; "Pr
00401309 call    printf
0040130E call    getchar
00401313 leave

```

Pongamos un breakpoint allí para ver si coincide al ejecutar el programa.

004012AD	mov	eax, [ebp+var_4C]	EBX 00004000	↳
004012B0	call	___chkstk	ECX 00401360	↳ ___do_globa
004012B5	call	main	EDX 77C31AE8	↳ msvcrt.dll:
004012BA	mov	dword ptr [esp+4], offset aJuan ; "Juan"	ESI 33010740	↳
004012C2	lea	eax, [ebp+persona1]		
004012C5	mov	[esp], eax ; char *	; Flags 40000040: Data Readable	
004012C8	call	strcpy	; Alignment : default	
004012CD	mov	dword ptr [esp+4], offset a		
004012D5	lea	eax, [ebp+persona1]	; Segment type: Pure data	
004012D8	add	eax, 1Eh	; Segment permissions: Read	
004012DB	mov	[esp], eax ; char *	_rdata segment para public 'DATA' use32	
004012DE	call	strcpy	assume cs: rdata	
004012E3	mov	[ebp+persona1.edad], 14h	;org 403000h	
004012EA	mov	eax, [ebp+persona1.edad]	; char aJuan[]	
004012ED	mov	[esp+0Ch], eax	aJuan db 'Juan',0	
004012F1	lea	eax, [ebp+persona1]		

Bueno vemos que va a pasar como argumento de **strcpy** el puntero a donde se encuentra la string Juan, en este caso 403000.

Luego lee la dirección de **persona1** con **lea** para pasársela como argumento a **strcpy** la cual copiará la misma allí, pasemos el **strcpy** con f8.

Si hago click en persona1

```

004012B0 call    ___chkstk
004012B5 call    main
004012BA mov     dword ptr [esp+4], offset aJuan ;
004012C2 lea     eax, [ebp+persona1]
004012C5 mov     [esp], eax      char *
004012C8 call    strcpy
004012CD mov     dword ptr [esp+4], offset aJ@j_j ;
004012D5 lea     eax, [ebp+persona1]

```

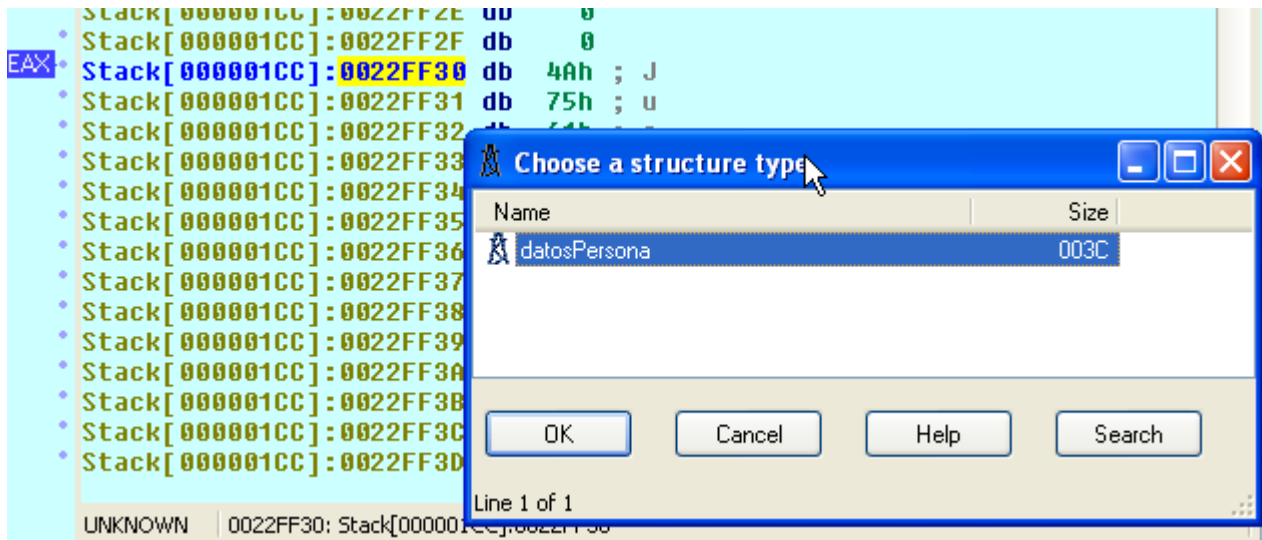
Veo el stack, la palabra Juan y el cero final de la string

```

Stack[000001CC]:0022FF2E db 0
Stack[000001CC]:0022FF2F db 0
EAX Stack[000001CC]:0022FF30 db 4Ah ; J
Stack[000001CC]:0022FF31 db 75h ; u
Stack[000001CC]:0022FF32 db 61h ; a
Stack[000001CC]:0022FF33 db 6Eh ; n
Stack[000001CC]:0022FF34 db 0
Stack[000001CC]:0022FF35 db 0FFh
Stack[000001CC]:0022FF36 db 22h ; "
Stack[000001CC]:0022FF37 db 0
Stack[000001CC]:0022FF38 db 8
Stack[000001CC]:0022FF39 db 0
Stack[000001CC]:0022FF3A db 0
Stack[000001CC]:0022FF3B db 0
Stack[000001CC]:0022FF3C db 0E0h ; 0
Stack[000001CC]:0022FF3D db 0FFh

```

Para aclarar asigno allí a 22ff30 la estructura que cree con ALT mas Q.

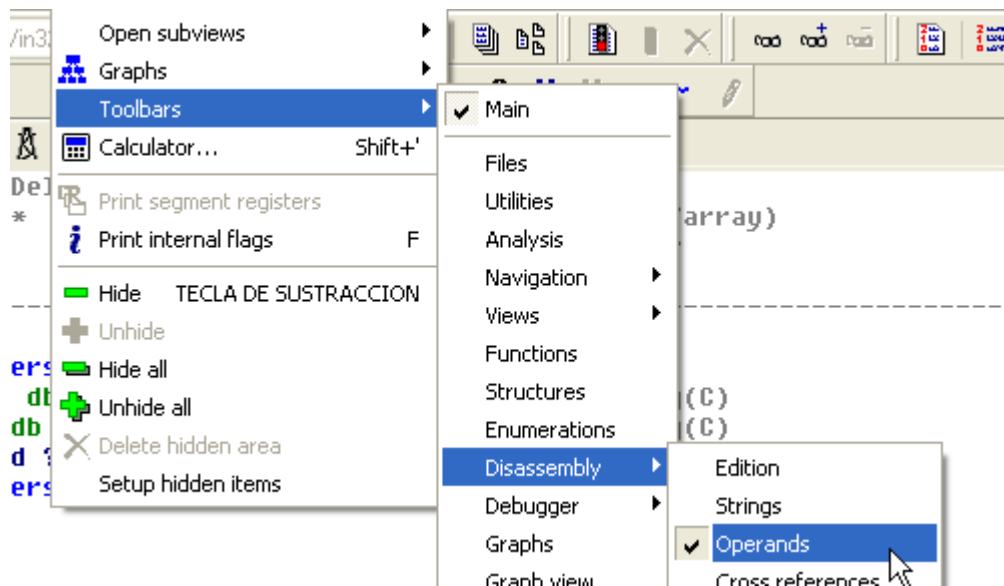


```

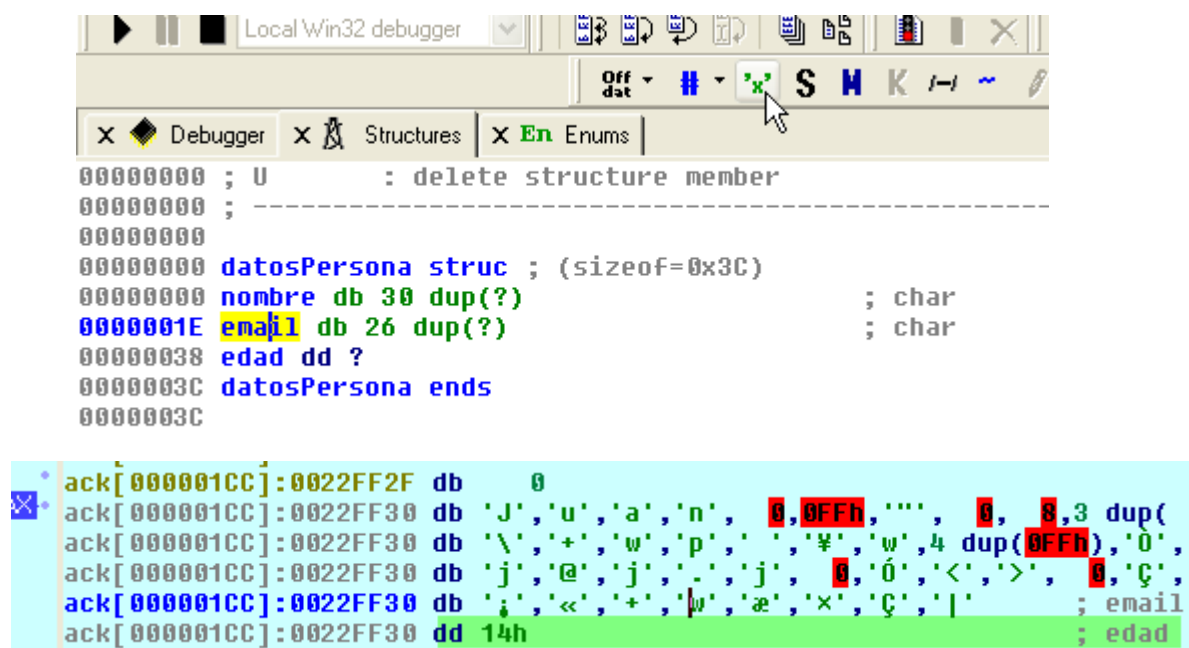
:0022FF2F db 0
:0022FF30 db 4Ah, 75h, 61h, 6Eh, 0, 0FFh, 22h, 0, 8, 3 dup(0), 0E0h, 0
:0022FF30 db 94h, 5Ch, 0C0h, 77h, 70h, 20h, 0BEh, 77h, 4 dup(0FFh), 0E
:0022FF30 db 0BFh, 77h, 14h, 0B8h, 0C0h, 77h, 0E0h, 3Ch, 3Eh, 0, 80h,
:0022FF30 db 8, 3 dup(0), 0ADh, 0AEh, 0C0h, 77h, 91h, 9Eh, 80h, 7Ch; e
:0022FF30 dd 33010740h ; edad
:0022FF6C db 008h - ;

```

Ugh quedo feo veamos como arreglarlo aunque el hecho de que tenga basura entre medio de las strings complica a que el IDA lo reconozca bien, pero bueno yendo a la definición de la estructura y poniendo visible el menú.

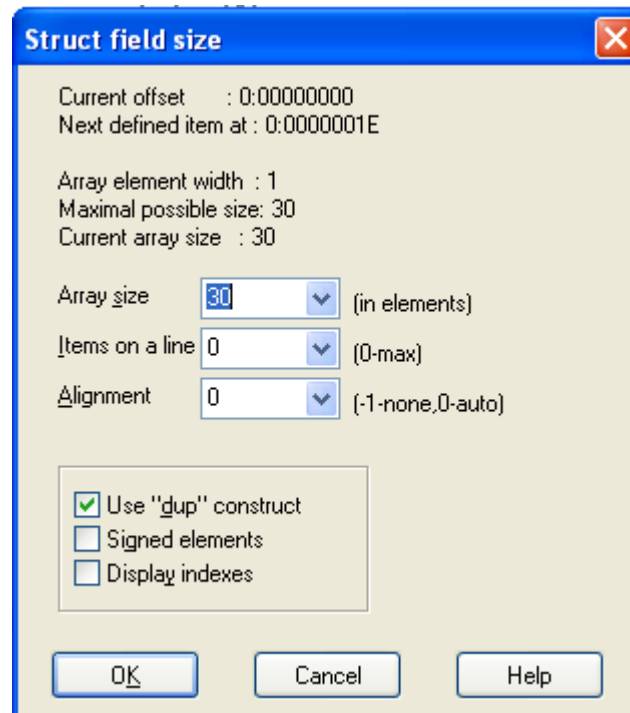


Al marcar los campos de strings y apretar allí se pone verde y muestra char, aunque todavía no queda bien.

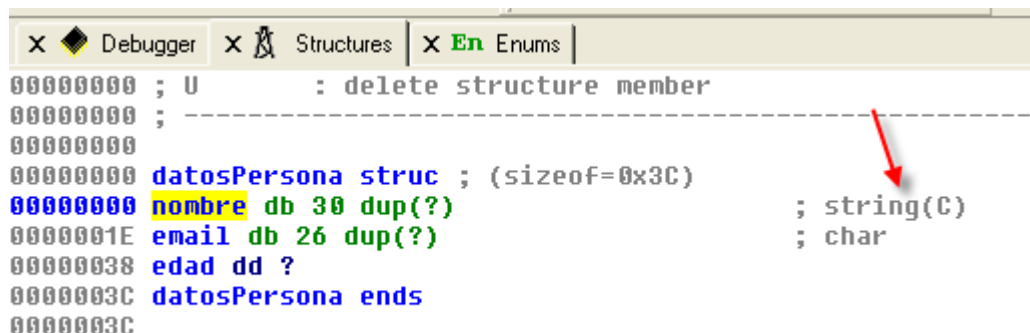


Queda un poco mejor pero no como string, ahora un truco medio de la galera si originalmente en

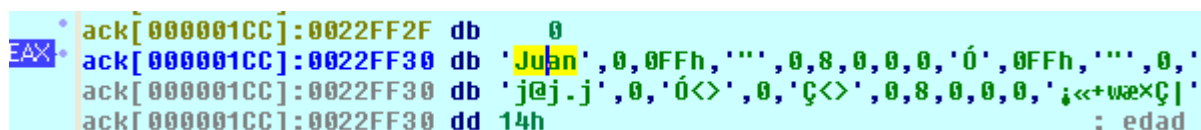
vez de asterisco encima de nombre apretamos la A.



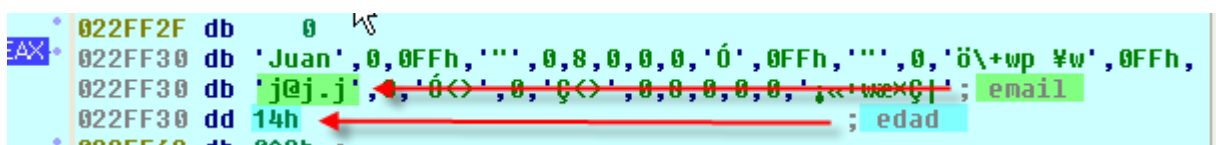
y aceptamos vemos que cambio el tipo a string C



Hago lo mismo con la otra.



Bueno al menos vemos las strings, el tema de que no las agrupa debe ser por la basura intermedia, pero al menos vemos que coinciden, donde empieza email esta la string del mismo y donde empieza edad esta el 14h.



```

.text:004012E0 mov     [esp+00h], eax
.text:004012F1 lea     eax, [ebp+persona1]
.text:004012F4 add     eax, 1Eh
.text:004012F7 mov     [esp+8], eax
.text:004012FB lea     eax, [ebp+persona1]
.text:004012FE mov     [esp+4], eax
.text:00401302 mov     dword ptr [esp], offset aPrimeraPersona ; "Prim
.text:00401309 call    printf

```

Vemos que luego saca las direcciones del primer campo nombre y le suma **1eh** para obtener la dirección del segundo campo **email** y los imprime, ahora veremos el mismo caso pero dinámico.

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

```
main() {
```

```
/* Primero definimos nuestro tipo de datos */
```

```
struct datosPersona {
```

```
    char nombre[30];
```

```
    char email[25];
```

```
    int edad;
```

```
};
```

```
/* La segunda persona será dinamica */
```

```
struct datosPersona *persona2;
```

```
/* Ahora a la dinámica */
```

```
persona2 = (struct datosPersona*)
```

```
    malloc (sizeof(struct datosPersona));
```

```
strcpy(persona2->nombre, "Pedro");
```

```
strcpy(persona2->email, "p@p.p");
```

```
persona2->edad = 21;
```

```
/* Mostramos los datos y liberamos la memoria */
```

```
printf("Segunda persona: %s, %s, con edad %d\n",
```

```
    persona2->nombre, persona2->email, persona2->edad);  
free(persona2);
```

```
    getchar();  
}
```

Vemos que la estructura es la misma que antes, pero en vez de instanciarla en forma estática, lo hacemos en forma dinámica, ahora **persona2** sera un puntero a la estructura.

```
struct datosPersona *persona2;
```

Vemos como reserva el espacio necesario ya que **sizeof** le devuelve al compilador el tamaño de la estructura, así que eso sera una constante, por supuesto **malloc** devolverá un puntero a la zona donde reservo dicho espacio, luego para que no haya problemas al compilar castea el tipo a (**struct datosPersona***) como puntero a esa estructura.

```
persona2 = (struct datosPersona*)  
    malloc (sizeof(struct datosPersona));
```

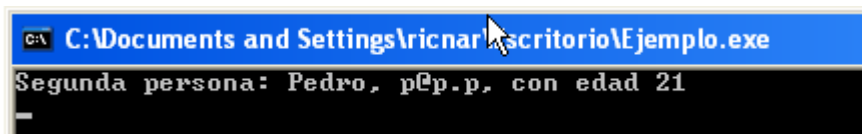
Así que tenemos nuestro puntero en **persona2**, y el espacio necesario reservado al que apunta, como habíamos visto en el caso de las estructuras dinámicas se debe usar la flecha para acceder a los campos.

```
strcpy(persona2->nombre, "Pedro");  
strcpy(persona2->email, "p@p.p");  
persona2->edad = 21;
```

Luego imprime la salida también usando la flecha para acceder a los campos y luego cuando termina libera la memoria reservada.

```
printf("Segunda persona: %s, %s, con edad %d\n",  
    persona2->nombre, persona2->email, persona2->edad);  
free(persona2);
```

Funciona perfectamente



Veamos este nuevo ejemplo en IDA


```

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

var_8= dword ptr -8
var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 18h
and     esp, 0FFFFFF0h
mov     eax, 0
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], 3Ch ; size_t
call    malloc
mov     [ebp+var_4], eax
mov     dword ptr [esp+4], offset aPedro ; "Pedro"
mov     eax, [ebp+var_4]

```

Vemos donde empieza nuestro código debajo de la línea, lo primero que observamos es que no se está reservando lugar en el stack para la estructura como antes, solo una variable **var_4** que es un dword es usada, la otra **var_8** la usa el compilador.

```

IDA View-A | Stack of _main | Hex View-A | Structures | En Enums | Imports |
-00000009          db ? ; undefined
-00000008  var_8      dd ?
-00000004  var_4      dd ? ; offset
+00000000          db 4 dup(?)
+00000004          db 4 dup(?)
+00000008  argc      dd ?
+0000000C  argv      dd ? ; offset
+00000010  envp      dd ? ; offset
+00000014
+00000014 ; end of stack variables

```

Vemos que no hay espacio no definido, las variables están consecutivas, y la **var_4** es un dword y a continuación ya se encuentra el **stored_ebp**, así que no hay lugar vacío para arrays ni estructuras estáticas aquí, solo nuestra **var_4** que ocupa 4 bytes y nada más.

```

mov     eax, [ebp+var_8]
call    __chkstk
call    __main
mov     dword ptr [esp], 3Ch ; size_t
call    malloc
mov     [ebp+var_4], eax

```

Lo primero que hace es llamar a malloc reservando el espacio **3ch** que es el tamaño de nuestra estructura, el puntero lo guarda en la variable que podemos renombrar como **persona2**.

```

call    __main
mov     dword ptr [esp], 3Ch ; size_t
call    malloc
mov     [ebp+persona2], eax

```

Luego trabaja todo con punteros así que aquí no hay que hacer **lea** ni nada, le pasamos el puntero **persona2** a **strcpy**, IDA reconoce que es un puntero y por eso nos muestra **char *** llenamos el campo **nombre**.

```

mov     eax, [ebp+var_8]
call    __chkstk
call    __main
mov     dword ptr [esp], 3Ch ; size_t
call    malloc
mov     [ebp+persona2], eax
mov     dword ptr [esp+4], offset aPedro ; "F
mov     eax, [ebp+persona2]
mov     [esp], eax ; char *
call    strcpy
mov     dword ptr [esp+4], offset a@a_p ; "f
mov     eax, [ebp+persona2]
add     eax, 1Eh
mov     [esp], eax ; char *
call    strcpy
mov     eax, [ebp+persona2]
mov     dword ptr [eax+38h], 15h
mov     eax, [ebp+persona2]
mov     eax, [eax+38h]
mov     [esp+0Ch], eax
mov     eax, [ebp+persona2]
add     eax, 1Eh

```

Luego a dicho puntero le suma **1eh** obteniendo un puntero al segundo campo **email** de la estructura también aquí IDA nos muestra que el argumento es un **char ***.

```

mov     eax, [ebp+persona2]
mov     dword ptr [eax+38h], 15h

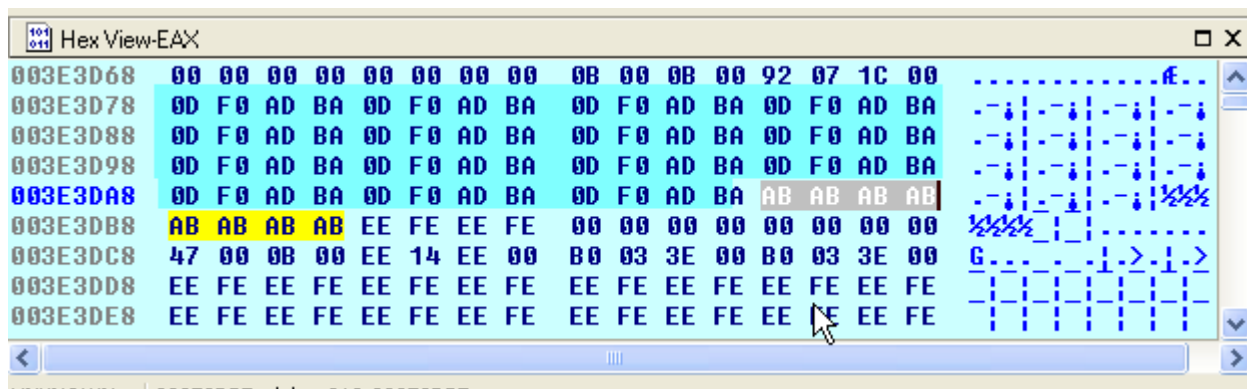
```

Luego lee el puntero a **persona2** nuevamente le suma **38h** y allí asigna el valor 15 al campo **edad**.

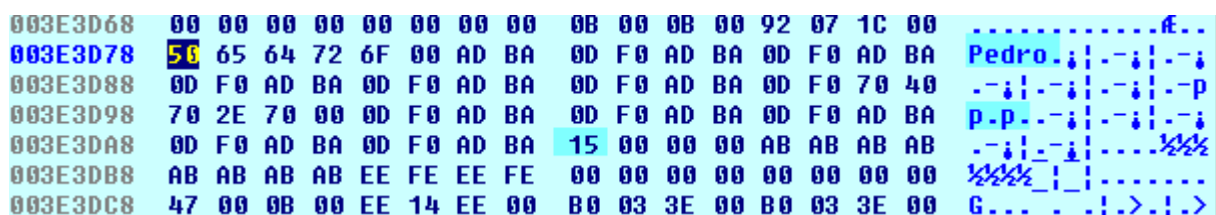
Aquí no podemos en el análisis estático, asignar nuestra estructura a nada, pues no esta en el stack y es dinámica o sea creada al correr el programa, lo debuggaremos para ver si quedo todo bien.

The screenshot shows the IDA Pro interface. On the left, the 'IDA View-EIP' window displays assembly code. The instruction at address 004012B6, 'mov dword ptr [esp], 3Ch ; size_t', is highlighted in red. Below it, the instruction at 004012C1, 'call malloc', is highlighted in blue. On the right, the 'General registers' window shows the current state of the CPU registers. The EAX register contains the value 003E3D78, which is linked to 'debug013:003E3D78'. The ESP register contains 0022FF50, linked to 'Stack[00000BD4]:0022FF50'. The EIP register contains 004012C6, linked to '_main+36'.

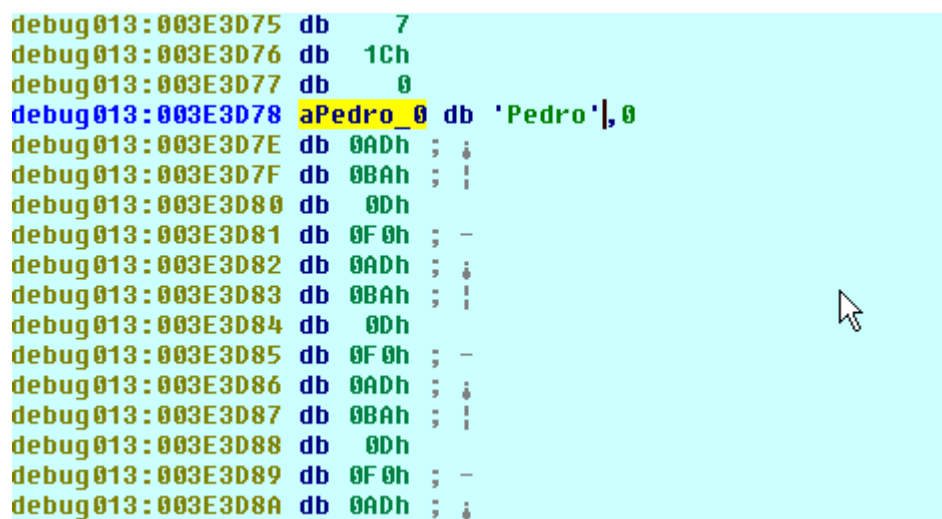
Al pasar **malloc** EAX tiene el valor del puntero en mi maquina 0x3e3d78, allí reservo **3ch** de espacio veamos en el DUMP hagamos click derecho SINCRONIZE WITH EAX.

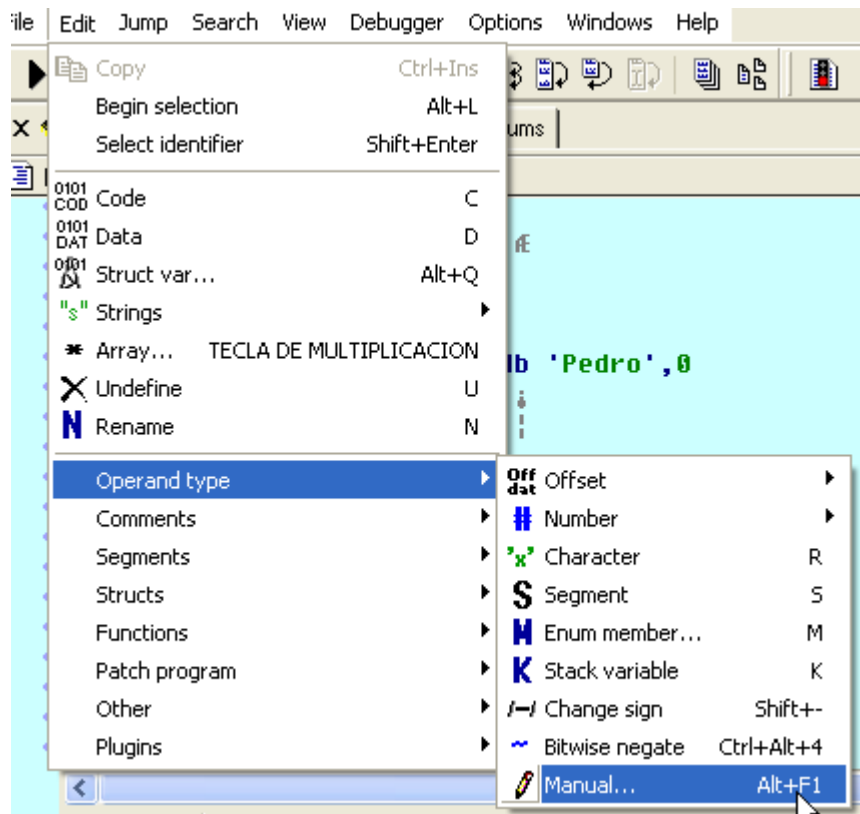


Allí esta, traceemos con f8 y vayamos copiando, vemos como llena los campos de la estructura con las strings Pedro, el mail y el dword 15.



Si vemos la estructura en el desensamblado





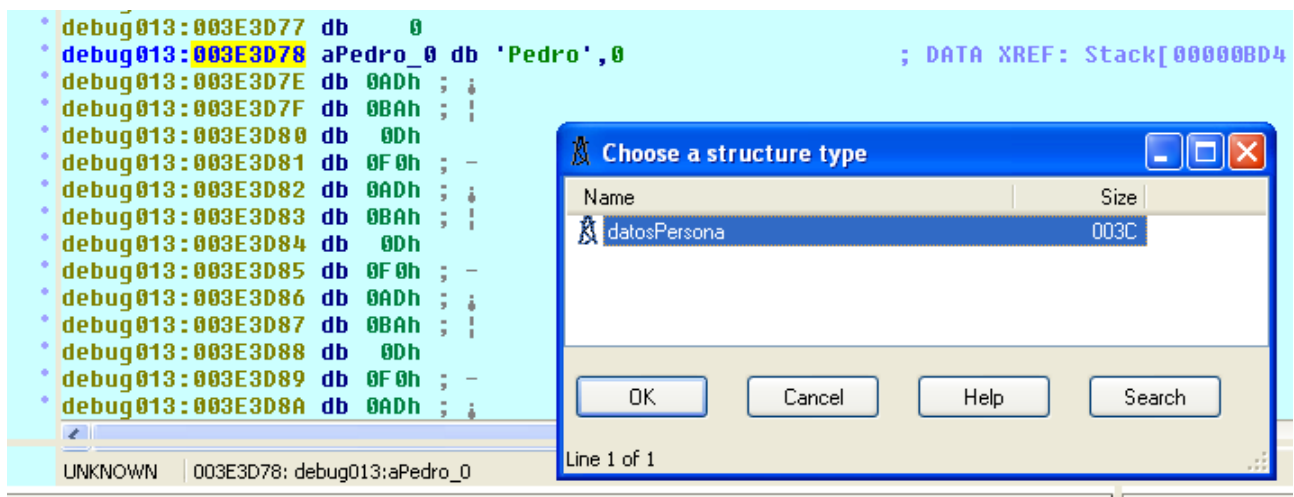
Bueno no me deja me da error, pero a no asustarse que yo estoy temblando seguro que como no la guardamos no nos deja, pero bueno se crea en un minuto de nuevo ya sabemos que si el campo es una string apretamos A en vez de asterisco.

Creamos la estructura nuevamente, usamos A en vez de asterisco en los campos que serán strings.

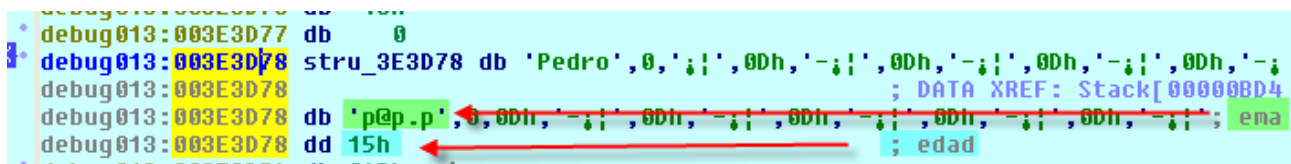
```

00000000 ,
00000000 datosPersona struc ; (sizeof=0x3C)
00000000 nombre db 30 dup(?) ; string(C)
0000001E email db 26 dup(?) ; string(C)
00000038 edad dd ?
0000003C datosPersona ends
0000003C

```



Vemos que los datos corresponden aunque quizás por el mismo motivo que cuando es estática, demasiada basura intermedia, no nos crea el formato mas cómodo, pero vemos bien que esta todo correcto.



Luego primero le pasa como argumento el valor de la edad, que obtiene a partir del puntero **persona2** al cual le suma 38h y obtiene un puntero al campo **edad** y al cual le halla el contenido **15** y lo guarda en el stack, para los otros campos ya que son strings solo es necesario el puntero al inicio de la misma así que le pasa **puntero2** para el **nombre** y le suma **1eh** para apuntar al mail, luego hace **free** de persona2 para liberar la memoria reservada.

```
004012FC mov     eax, [ebp+persona2]
004012FF mov     eax, [eax+38h]
00401302 mov     [esp+0Ch], eax
00401306 mov     eax, [ebp+persona2]
00401309 add     eax, 1Eh
0040130C mov     [esp+8], eax
00401310 mov     eax, [ebp+persona2]
00401313 mov     [esp+4], eax
00401317 mov     dword ptr [esp], offset aSegundaPersona ; "Segunda persona: %s, %s, con
0040131E call    printf
00401323 mov     eax, [ebp+persona2]
```

Hay un ejercicio para solucionar casi igual a ambos de esta parte pero juntos.

Hasta la parte siguiente
Ricardo Narvaja