

C Y REVERSING (parte 7) por Ricnar

Tablas bidimensionales

Podemos declarar tablas de dos o más dimensiones. Por ejemplo, si queremos guardar datos de dos grupos de alumnos, cada uno de los cuales tiene 20 alumnos, tenemos dos opciones:

- 1) Podemos usar **int datosAlumnos[40]** y entonces debemos recordar que los 20 primeros datos corresponden realmente a un grupo de alumnos y los 20 siguientes a otro grupo.
- 2) O bien podemos emplear **int datosAlumnos[2][20]** y entonces sabemos que los datos de la forma **datosAlumnos[0][i]** son los del primer grupo, y los **datosAlumnos[1][i]** son los del segundo.

En cualquier caso, si queremos indicar valores iniciales, lo haremos entre llaves, igual que si fuera una tabla de una única dimensión. Vamos a verlo con un ejemplo de su uso:

PD :Esta explicación la copie del curso de C de Cabanes el que necesita la explicación completa la puede obtener aquí:

<http://www.nachocabanes.com/c/curso/cc05.php>

El primer ejemplo es este:

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

```
main(){
```

```
    funcion1();
```

```
    getchar();
```

```
}
```

```
funcion1(){
```

```
    int notas[2][10] =
```

```
    { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

```
      11, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
```

```
    printf("La nota del tercer alumno del grupo 1 es %d",
```

```
          notas[0][2]);
```

```
}
```

Vemos que crea una tabla de dos dimensiones o sea 2 x 10, y le asigna los 20 valores tipo int, los 10 primeros corresponden al primer grupo de alumnos, y los 10 siguientes al segundo grupo de alumnos, luego imprime la nota del tercer alumno del primer grupo que como los campos se numeran desde cero, sera el valor de la tabla **notas (0,2)**.

```

sub_4012C6 proc near

var_58= byte ptr -58h
var_50= dword ptr -50h

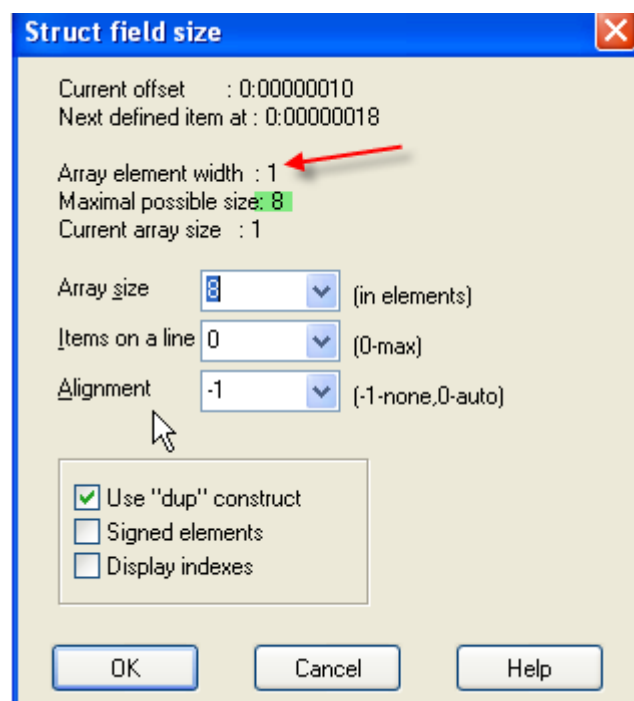
push    ebp
mov     ebp, esp
sub     esp, 68h
lea     ecx, [ebp+var_58]
mov     edx, offset unk_402000
mov     eax, 50h
mov     [esp+8], eax    ; size_t
mov     [esp+4], edx    ; void *
mov     [esp], ecx     ; void *
call    memcpy
mov     eax, [ebp+var_50]
mov     [esp+4], eax
mov     dword ptr [esp], offset aLaNotaDeITe
call    printf
leave
retn
sub_4012C6 endp

```

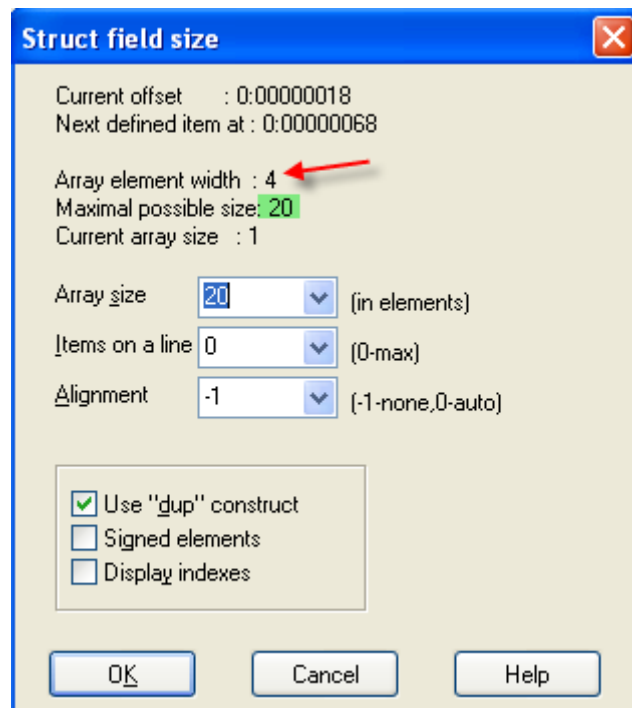
Bueno aquí el IDA la embarro bastante, por supuesto no interpreto como array de dos dimensiones o tabla, si no pescaba los array de una dimensión menos pescara los de dos jeje.

Vemos que creo una variable de un byte y otra de un word, si vemos el espacio que hay entre variables, vemos que es correcto, pues.

Vemos apretando asterisco en la primera variable, que la misma podría llegar a extenderse 8 bytes hasta llegar a la otra, ya que como IDA la definió como byte, el máximo largo 8 se refiere a 8 del mismo tipo o sea 8 bytes.



Si hacemos lo mismo en la siguiente que esta definida como dword.



Vemos que su largo es 20 dwords o sea 80 bytes o sea que el espacio total es 80.

Bueno cambiaremos la primera variable superior a dword y luego apretamos asterisco y le damos el espacio que sea la suma de los 2 dwords de la primera (8 bytes) mas los 20 dwords de la segunda variable (80 bytes) así que en total serán 22 dwords o sea 88 bytes un poquito mas largo que el nuestro pero no hay problema mientras no pisemos el stored_ebp y el return address.

Para cambiar a dword una variable en IDA, nos situamos en la definición de la misma donde dice **db (byte)** y apretamos la tecla D, primero cambiara a **dw(no confundir eso es word)** y luego cambia a **dd (dword)**

```
00000059      db ? ; undefined
00000058  var_58  db ? ; undefined
00000057      db ? ; undefined
00000056      db ? ; undefined
-----
```

Apretando la tecla D vemos que quedo **dd** como la otra variable

```
-00000059      db ? ; undefined
-00000058  var_58  dd ? ; undefined
-00000054      db ? ; undefined
-00000053      db ? ; undefined
-00000052      db ? ; undefined
-00000051      db ? ; undefined
-00000050  var_50  dd ? ; undefined
-----
```

Si vemos en el código:

```

sub_4012C6 proc near

var_58= dword ptr -58h
var_50= dword ptr -50h

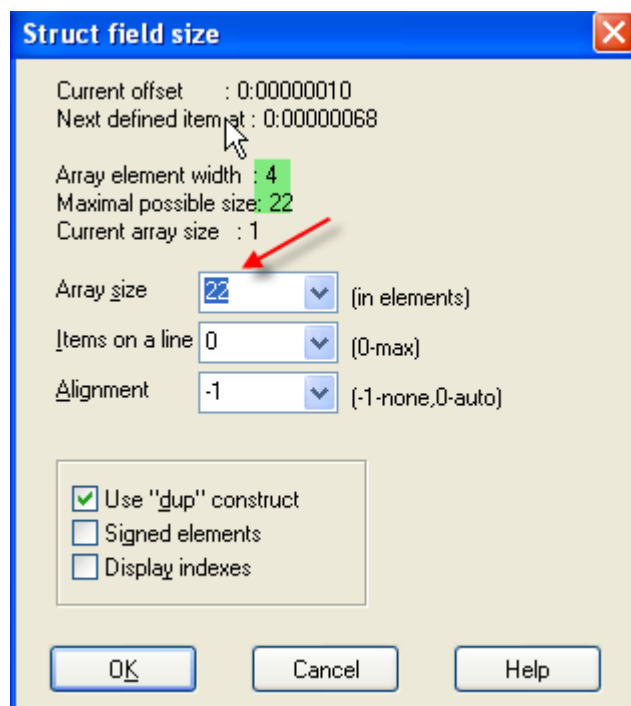
push    ebp
mov     ebp, esp
sub     esp, 68h
lea     ecx, [ebp+var_58]
mov     edx, offset unk_402000
mov     esi, 50h

```

Ahora las dos variables son DWORD, ahora volvamos a la tabla de variables y apretemos * y pongamos el largo máximo 22 dwords.

Me olvide de decir que otra forma de saber cuanto es el largo máximo desde esta primera variable, hasta el stored_ebp sin pisarlo, sin hacer cuentas es eliminar la segunda variable ya que la vamos a pisar, haciendo click derecho y undefine, así desaparece.

Ahora si apretamos asterisco en la primera vemos que nos dice 22 dwords el espacio máximo ya que no cuenta la segunda variable como pisada ya que no existe y cuenta hasta el stored_ebp.



Ahora si quedo una sola variable y todo completo.

```

-00000059          db ? ; undefined
-00000058 var_58    dd 22 dup(?)
+00000000          db 4 dup(?)
+00000004          db 4 dup(?)
+00000008
+00000008 ; end of stack variables

```

De cualquier forma vemos que a diferencia de nuestro código quedo un array unidimensional con un solo index que lo recorre entero, mientras que en nuestro código fuente era un array bidimensional con dos indices que recorrían los dos grupos de 10 alumnos cada uno.

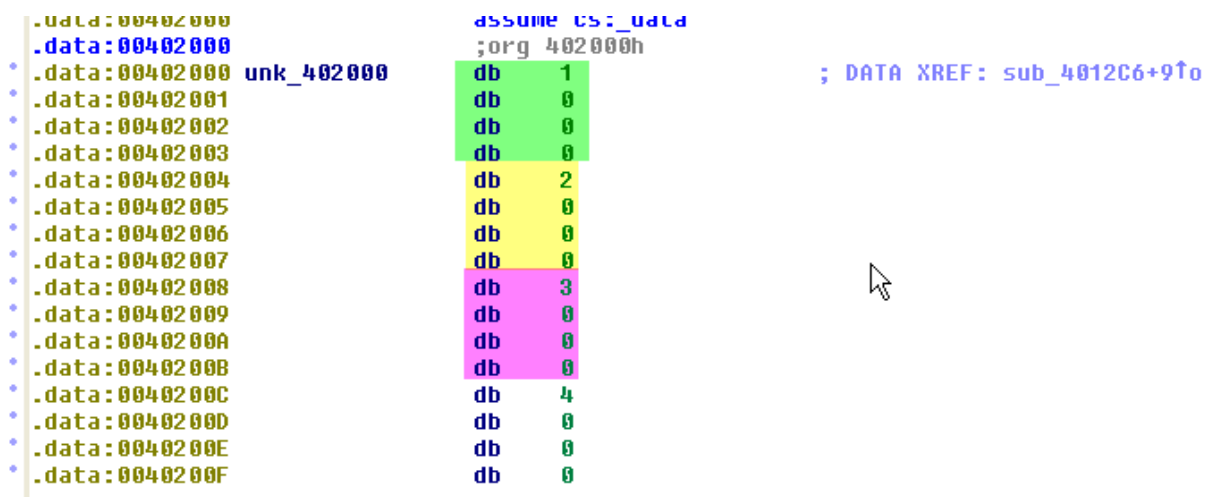
Quiere decir que donde nuestro código muestra imprimir el elemento (0,2) de una tabla de dos dimensiones de 10 elementos cada campo, en el código del IDA sera un solo indice de 0 a 19 y el buscado sera el tercer elemento de este array.

Veamos como terminar de interpretarlo así, y luego veremos que podemos hacer para poner doble indice ya que el IDA no nos deja la posibilidad usando array.

Vemos que inicializara usando **memcpy**, que necesita tres argumentos, el tamaño de lo que va a copiar en bytes, la dirección desde donde copiar o fuente, y la dirección de destino adonde copiara o destination.

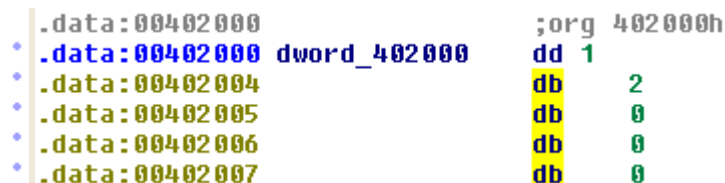
```
sub esp, 08h
lea ecx, [ebp+notas]
mov edx, offset unk_402000
mov eax, 50h
mov [esp+8], eax ; size_t
mov [esp+4], edx ; void *
mov [esp], ecx ; void *
call memcpy
```

Sera el size de la copia 50h o sea 80 bytes que es el largo de las 20 notas en dword, si vamos a 402000 que es la dirección de la fuente veremos las notas.



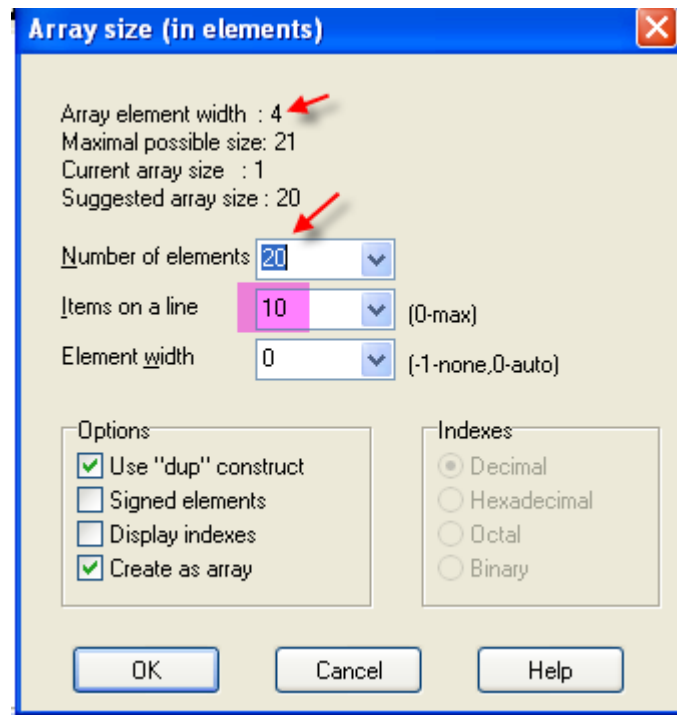
Address	Disassembly	Comment
.data:00402000	assume cs:_data	
.data:00402000	org 402000h	
.data:00402000	unk_402000 db 1	; DATA XREF: sub_4012C6+9↑o
.data:00402001	db 0	
.data:00402002	db 0	
.data:00402003	db 0	
.data:00402004	db 2	
.data:00402005	db 0	
.data:00402006	db 0	
.data:00402007	db 0	
.data:00402008	db 3	
.data:00402009	db 0	
.data:0040200A	db 0	
.data:0040200B	db 0	
.data:0040200C	db 4	
.data:0040200D	db 0	
.data:0040200E	db 0	
.data:0040200F	db 0	

Allí las vemos, podemos también aquí darle formato de array, primero cambiamos **db** por **dd** ya que son dwords allí en 402000 apretando D dos veces.

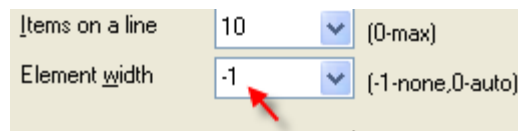


Address	Disassembly	Comment
.data:00402000	assume cs:_data	
.data:00402000	org 402000h	
.data:00402000	dword_402000 dd 1	
.data:00402004	db 2	
.data:00402005	db 0	
.data:00402006	db 0	
.data:00402007	db 0	

Ahora apretamos asterisco.



El array element es 4 porque cada elemento es un dword, el máximo posible size antes de pisar algo (en este caso otra referencia) es 21 dwords, pero como sabemos que son 20 los datos, cambiamos a 20, y como son dos grupos de a 10, podemos poner en la segunda línea 10, para que muestre como dos grupos de a 10, lo cual no tiene efecto en el index, seguirá siendo un array unidimensional pero aquí en la memoria la creación de arrays es mas flexible al no haber variables que respetar así que lo muestra correctamente en dos líneas.



Por ultimo al poner -1 en el ancho mostrado de cada elemento los agrupa para que se vean mas compactos y no deja espacio entre cada uno.

```

:00402000 dword_402000 dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 0Ah; 0
:00402000 ; DATA XREF: sub_4012C6+9↑to
:00402000 dd 0Bh, 0Ch, 0Dh, 0Eh, 0Fh, 10h, 11h, 12h, 13h, 14h; 10

```

Así se ve mejor en la memoria acomodados los datos.

Si posamos el mouse en la flechita que nos muestra la referencia desde donde es llamado este array vemos el código en el popup y marcado cuando llama a 402000 para pasarlo a EDX y usarlo como dirección de la fuente de datos del **memcpy**, por supuesto el destination o donde copiara estos datos sera nuestra la dirección de la variable **notas** en el stack, y el largo 80 bytes o sea 50h.

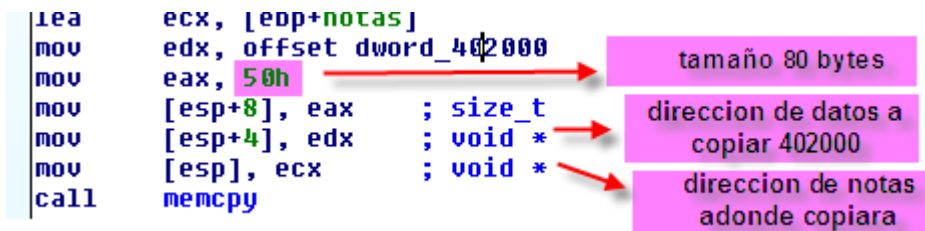
```

:00402000 ; Segment type: Pure data
:00402000 ; Segment permissions: Read/Write
:00402000 _data segment para public 'DATA' use32
:00402000 assume cs:_data
:00402000 ;org 402000h
* :00402000 dword_402000 dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 0Ah; 0
:00402000 ; DATA XREF: sub_4012C6+91fo
- 00402000 dd 0Ah 0Ch 0Dh 0Fh 0Fh 10h 11h 12h 13h 14h 15h 16h 17h 18h 19h 1Ah 1Bh 1Ch 1Dh 1Eh 1Fh
; CODE XREF: _main+2A1p

sub_4012C6 proc near
    notas = dword ptr -58h

    push ebp
    mov ebp, esp
    sub esp, 68h
    lea ecx, [ebp+notas]
    mov edx, offset dword_402000
    mov eax, 50h

```



Luego de copiar la data leerá la tercer nota, ya que desde el inicio de **notas**, como son dwords, la tercera estará 8 bytes mas adelante y luego imprimirá ese valor.

```
mov    eax, [ebp+notas+8]
```

Si alguien probó vemos que cambiando en la memoria la forma de visualización por dos líneas de 10, mejoro la misma, pero en el stack cuando creamos nuestra variable **notas** si pones 10 en el mismo casillero no hace un array bidimensional ni modifica la forma de usar los índices, así que veremos si podemos acomodarlo de otra forma para que se adapte mejor a nuestro código ya que revisando los libros no encontré forma de hacerlo con arrays, así que les mostrare como lo hago yo, no se si es la mejor forma pero al menos quedara con doble subíndice de la misma forma que el código original.

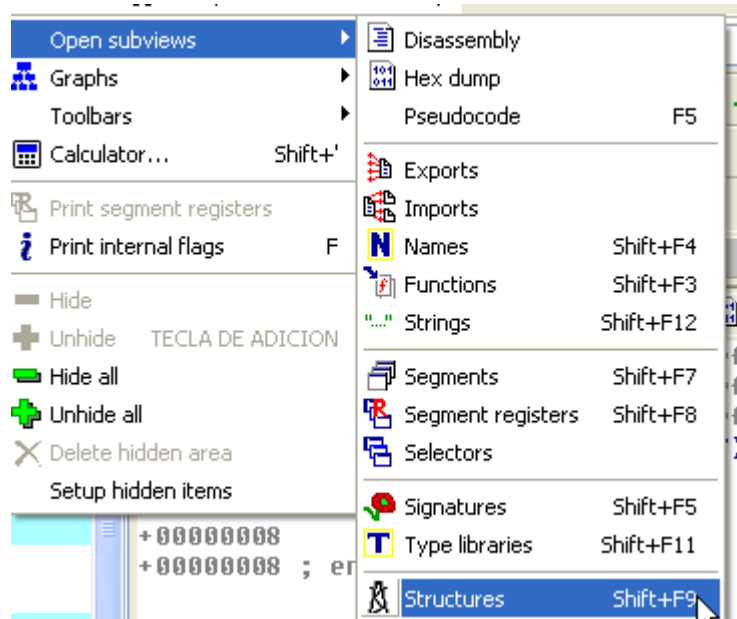
```

-00000000  ;
-00000058  notas
+00000000  s
+00000004  r
+00000008
+00000008 ; end of stack variables
uu ? ; undefined
dd 22 dup(?)
db 4 dup(?)
db 4 dup(?)

```

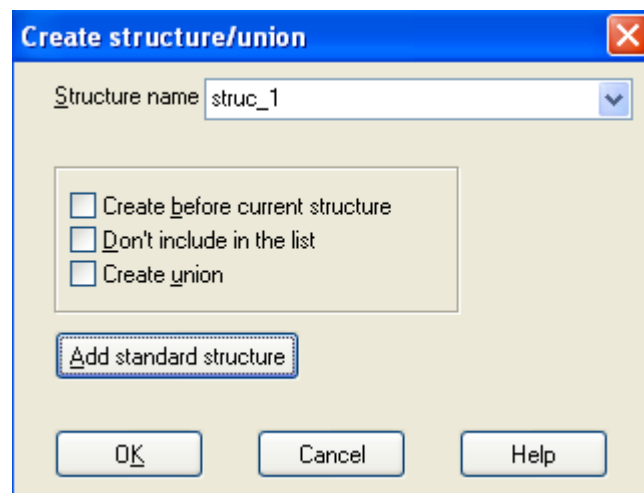
Cambiamos **notas** de nuevo a que sea una variable dword como era originalmente, este método se puede aplicar sin necesidad de hacer desaparecer la otra variable que había, funciona igual y la pisa tal cual como los arrays.

Abramos la ventana de estructuras, no tiemblen que no es tan difícil como lo pintan jeje



```
; Ins/Del : create/delete structure
; D/A/*   : create structure member (data/ascii/array)
; N       : rename structure or structure member
; U       : delete structure member
```

Ahí vemos que apretando **INS** o la tecla INSERT creamos una estructura nueva, lo hacemos con estructuras porque un array bidimensional es una forma de estructura, lo cual no hemos visto aun, pero sabemos que una estructura puede contener diferentes tipos de campos mezclados, y en nuestro caso contendrá dos arrays de 10 dwords cada campo.



No importa el nombre demos OK.

D/A/* : create structure member (data/ascii/array)

Vemos en la aclaración que para agregar algo debemos apretar **D** para datos, **A** para ASCII y **asterisco** para arrays, hagamos esto ultimo.


```

00000000
00000000 struc_1      struc ; (sizeof=0x1)
00000000 field_0      db ?
00000001 struc_1      ends
00000001

```

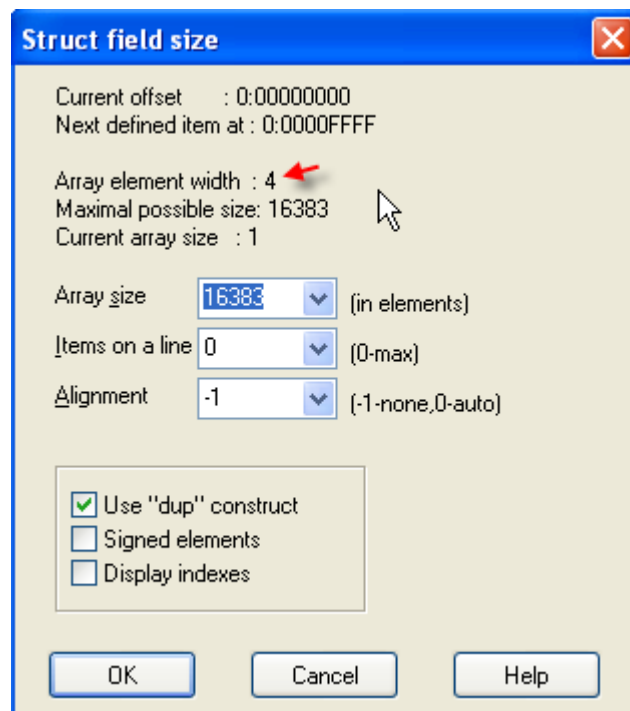
Vemos que nos creó un array de un solo byte arriba nos dice el tamaño de la estructura que por ahora es uno.

```

00000000
00000000 struc_1      struc ; (sizeof=0x4
00000000 field_0      dd ?
00000004 struc_1      en
00000004

```

Marcando allí y apretando la D nos deja cambiarlo a DWORD ahora apretamos asterisco de nuevo.



Allí vemos que cada elemento será un dword y que el tamaño máximo aquí no está limitado pues estamos en una definición de estructura fuera del programa, igual sabemos que cada array debe tener 10 dwords así que ponemos 10.

```

00000000
00000000 struc_1      struc ; (sizeof=0x28)
00000000 field_0      dd 10 dup(?)
00000028 struc_1      ends
00000028

```

ya tenemos el primer array para agregar un segundo debemos ir al final a **ends** y apretar de nuevo asterisco y repetir lo mismo quedará así.

```

00000000
00000000 struc_1      struc ; (sizeof=0x50)
00000000 field_0      dd 10 dup(?)
00000028 field_28     dd 10 dup(?)
00000050 struc_1      ends
00000050

```

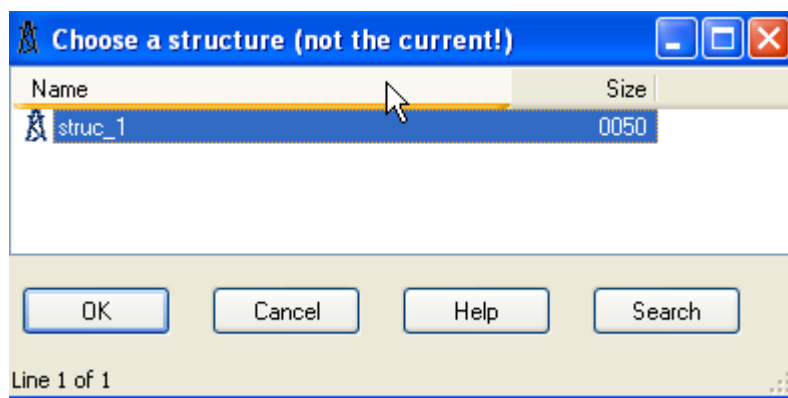
Listo ya esta creada ahora volvamos a las variables.

```

-00000059          db ? ; undefined
-00000058 notas    dd 22 dup(?)
+00000000          db 4 dup(?)
+00000004          db 4 dup(?)
+00000008
+00000008 ; end of stack variables

```

Marquemos notas y apretemos ALT mas Q.



Ahí sale la que creamos para elegir, damos OK.

```

-0000005A          db ? ; undefined
-00000059          db ? ; undefined
-00000058 notas    struc_1 ?
-00000008          db ? ; undefined
-00000007          00000000 ; Ins/Del : create/delete structure
-00000006          00000000 ; D/A/* : create structure member (data/ascii/array)
-00000005          00000000 ; N : rename structure or structure member
-00000004          00000000 ; U : delete structure member
-00000003          00000000 ; -----
-00000002          00000000
-00000001          00000000 struc_1      struc ; (sizeof=0x50)
+00000000          s 00000000 field_0      dd 10 dup(?)
+00000004          r 00000028 field_28     dd 10 dup(?)
+00000008          ...
+00000008 ; end of stack variables

```

Si pasamos el mouse por notas vemos que ahora nos muestra la definición de la estructura.

```

notas= struc_1 ptr -58h

push    ebp
mov     ebp, esp
sub     esp, 68h
lea     ecx, [ebp+notas]
mov     edx, offset dword_402000
mov     eax, 50h
mov     [esp+8], eax      ; size_t
mov     [esp+4], edx      ; void *
mov     [esp], ecx        ; void *
call    memcpy
mov     eax, [ebp+notas.field_0+8]
mov     [esp+4], eax
mov     dword ptr [esp], offset aLaNotaDel
call    printf
leave
retn
sub_4012C6 endp

```

Vemos que ahora si la variable notas tiene doble subindice, ya que el primer grupo de 10 se llama field0, y el segundo field1, podemos renombrarlos en la misma estructura.

```

IDA View-A | Stack of sub_4012C6 | Hex View-A | Structures | En
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=0x50)
00000000 grupo_1      dd 10 dup(?)
00000028 grupo_2      dd 10 dup(?)
00000050 struc_1      ends
00000050
00000050

```

```

notas= struc_1 ptr -58h

push    ebp
mov     ebp, esp
sub     esp, 68h
lea     ecx, [ebp+notas]
mov     edx, offset dword_402000
mov     eax, 50h
mov     [esp+8], eax    ; size_t
mov     [esp+4], edx    ; void *
mov     [esp], ecx      ; void *
call    memcpy
mov     eax, [ebp+notas.grupo_1+8]
mov     [esp+4], eax
mov     dword ptr [esp], offset aLaNotaDelTerce ; "La nota de:
call    printf
leave
retn
sub_4012C6 endp

```

Ahora si vemos una estructura que es un array bidimensional con dos indices, ahí vemos que al leer la nota del primer grupo, usa el primer array **grupo_1** y indexa por este hasta el tercer elemento del mismo que sera el que esta 8 bytes mas adelante, no imprimimos un elemento del segundo grupo pero por ejemplo si lo hubiéramos hecho este aparecería como **notas.grupo_2+ x**, usando el otro indice.

Lo mas bonito es que como ya tenemos definida la estructura podemos ir a 402000 y cambiar a que en vez de un array sea la misma estructura esta, apretando ALT mas Q y seleccionándola, quedara así.

```

ta:00402000          ;org 402000h
ta:00402000 stru_402000 dd 1, 2, 3, 4, 5, 6, 7, 8, 9, 0Ah; grupo_1
ta:00402000          ; DATA XREF: sub_4012C6+9↑o
ta:00402000          dd 0Bh, 0Ch, 0Dh, 0Eh, 0Fh, 10h, 11h, 12h, 13h, 14h; grupo_2

```

Vemos que marca al final de cada uno el nombre de los arrays **grupo_1** y **grupo_2**.

Bueno esto ademas de servirnos para ver arrays bi o multidimensionales, no sirvió para iniciarnos con simples estructuras, las cuales profundizaremos en la parte siguiente.

Hasta la próxima
Ricardo Narvaja