

C Y REVERSING (parte 5) por Ricnar

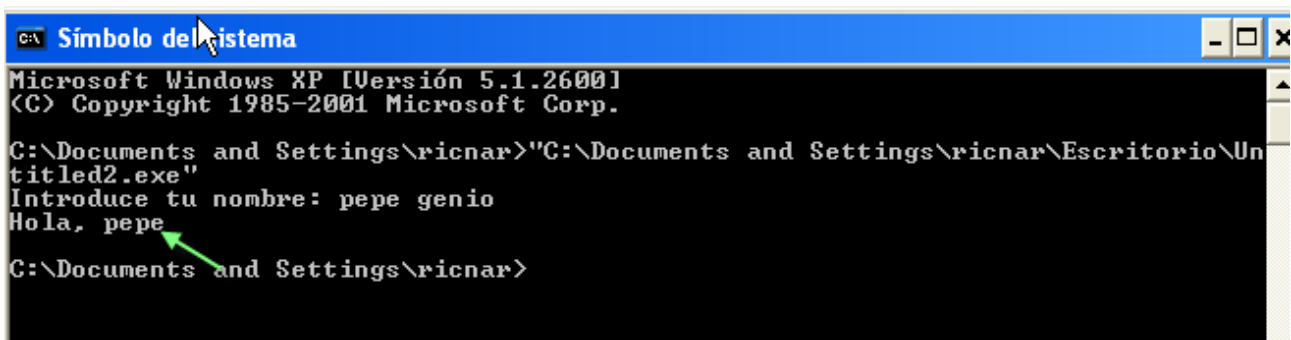
Si no se dieron cuenta en los ejercicios anteriores pueden probar y verán que **scanf** a pesar de que te deja tipear después de un espacio, solo lee hasta allí, por ejemplo.

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

```
main(){  
    funcion();  
    getchar();  
    getchar();  
    getchar();  
}
```

```
funcion(){  
  
    char texto[40];    /* Para guardar hasta 39 letras */  
  
    printf("Introduce tu nombre: ");  
    scanf("%s", &texto);  
    printf("Hola, %s\n", texto);  
}
```

Si ejecuto esto y tipeo **pepe genio**



```
C:\ Símbolo del sistema  
Microsoft Windows XP [Versión 5.1.2600]  
<C> Copyright 1985-2001 Microsoft Corp.  
  
C:\Documents and Settings\ricnar>"C:\Documents and Settings\ricnar\Escritorio\Un  
titled2.exe"  
Introduce tu nombre: pepe genio  
Hola, pepe  
C:\Documents and Settings\ricnar>
```

Veo que imprime solo **pepe**.

Si queremos que un programa haga entrada por teclado sin detenerse en un espacio deberemos usar **gets()** en vez de **scanf**, aquí el ejemplo y para tener una mejor salida si es solo imprimir en pantalla y no necesita format string, podemos usar **puts()** en vez de **printf**.

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

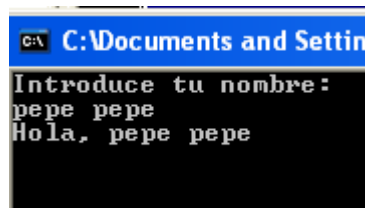
```
main(){  
    funcion();  
    getchar();  
  
}
```

```
funcion(){
```

```
char texto[40];    /* Para guardar hasta 39 letras */
```

```
puts("Introduce tu nombre: ");  
gets(texto);  
printf("Hola, %s\n", texto);  
}
```

El **printf** final como no es solo imprimir en pantalla sino que tiene un format string, no podemos reemplazarlo por un **puts**, si corremos este ejemplo vemos que ahora respeta los espacios.



Si lo vemos en IDA.

```
; Attributes: bp-based frame  
funcion_investigada proc near  
    texto= byte ptr -38h  
  
    push    ebp  
    mov     ebp, esp  
    sub     esp, 48h  
    mov     dword ptr [esp], offset aIntroduceTuNom ; "Introduce tu nombre: "  
    call    puts  
    lea     eax, [ebp+texto]  
    mov     [esp], eax ; char *  
    call    gets  
    lea     eax, [ebp+texto]  
    mov     [esp+4], eax  
    mov     dword ptr [esp], offset aHolaS ; "Hola, %s\n"  
    call    printf  
    leave  
    retn  
funcion_investigada endp
```

Vemos en amarillo que le pasa como argumento la dirección donde se encuentra la string al **puts**, aunque no lo hayamos especificado con **&** en el código fuente, y que el **gets** también usa la dirección de nuestra variable **texto** como argumento, en estos casos aun sin ser necesario en el código fuente aclararlo ya **gets** y **puts** interpretan que necesitan la dirección de la string, por supuesto el **printf** para hacer el format string con un **%s** necesita también la dirección de la misma o sea de la variable **texto** y también usa **lea** para hallarla.

Si le agregamos la siguiente linea

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

```
main(){  
    funcion();  
    getchar();  
}
```

```
}
```

```
funcion(){
```

```
    char texto[40];    /* Para guardar hasta 39 letras */
```

```
    puts("Introduce tu nombre: ");
```

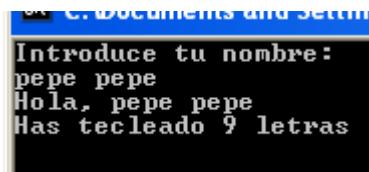
```
    gets(texto);
```

```
    printf("Hola, %s\n", texto);
```

```
    printf("Has tecleado %d letras", strlen(texto));
```

```
}
```

Vemos que usara **strlen** para hallar el largo de la string que tipeamos y luego pasara ese valor numérico como format string **%d** dentro del mensaje **Has tecleado XXX letras**.



```
Introduce tu nombre:
pepe pepe
Hola, pepe pepe
Has tecleado 9 letras
```

Por supuesto el espacio vacío es un carácter valido por eso cuenta 9 letras.

```
funcion_investigada proc near
```

```
    texto= byte ptr -38h
```

```
    push    ebp
```

```
    mov     ebp, esp
```

```
    sub     esp, 48h
```

```
    mov     dword ptr [esp], offset aIntroduceTuNom ; "Introduce tu nomb
```

```
    call    puts
```

```
    lea     eax, [ebp+texto]
```

```
    mov     [esp], eax ; char *
```

```
    call    gets
```

```
    lea     eax, [ebp+texto]
```

```
    mov     [esp+4], eax
```

```
    mov     dword ptr [esp], offset aHolaS ; "Hola, %s\n"
```

```
    call    printf
```

```
    lea     eax, [ebp+texto]
```

```
    mov     [esp], eax ; char *
```

```
    call    strlen
```

```
    mov     [esp+4], eax
```

```
    mov     dword ptr [esp], offset aHasTecleadoDLe ; "Has tecleado %d l
```

```
    call    printf
```

```
    leave
```

```
    retn
```

```
funcion_investigada endp
```

Bueno a **strlen** se le pasa también la dirección de nuestra variable **texto**, y el resultado que devuelve EAX en rosado, lo pasa como argumento de **printf** para hacer el format string e imprimir el numero de caracteres con **%d**.

Copio este texto del curso de Cabanes que explica como copiar cadenas y esta claro

Asignando a una cadena el valor de otra: strcpy, strncpy; strcat

Cuando queremos dar a una variable el valor de otra, normalmente usamos construcciones como `a = 2`, o como `a = b`. Pero en el caso de las cadenas de texto, esta NO es la forma correcta, no podemos hacer algo como `saludo="hola"` ni algo como `texto1=texto2`. Si hacemos algo así, haremos que las dos cadenas estén en la misma posición de memoria, y que los cambios que hagamos a una de ellas se reflejen también en la otra. La forma correcta de guardar en una cadena de texto un cierto valor es:

strcpy (destino, origen);

Es decir, debemos usar una función llamada “**strcpy**” (string copy, copiar cadena), que se encuentra también en “string.h”. Vamos a ver dos ejemplos de su uso:

strcpy (saludo, "hola");

Es nuestra responsabilidad que en la cadena de destino haya suficiente espacio reservado para copiar lo que queremos. Si no es así, estaremos sobrescribiendo direcciones de memoria en las que no sabemos qué hay.

Para evitar este problema, tenemos una forma de indicar que queremos copiar sólo los primeros **n bytes** de origen, usando la función “**strncpy**”, así:

strncpy (destino, origen, n);

Vamos a ver un ejemplo, que nos pida que tecleemos una frase y guarde en otra variable sólo las 4 primeras letras:

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

```
#include <string.h>
```

```
main(){  
    funcion();  
    getchar();
```

```
}
```

```
funcion(){
```

```
    char texto1[40], texto2[40], texto3[10];
```

```
    printf("Introduce un frase: ");
```

```
    gets(texto1);
```

```
    strcpy(texto2, texto1);
```

```
    printf("Una copia de tu texto es %s\n", texto2);
```

```
    strncpy(texto3, texto1, 4);
```

```
    printf("Y sus 4 primeras letras son %s\n", texto3);
```

```
}
```

```
C:\Documents and Settings\ricnar\Escritorio\Un
Introduce un frase: supertexto
Una copia de tu texto es supertexto
Y sus 4 primeras letras son supeAAAA
```

El problema con **strncpy** es que copia solo 4 caracteres no pone el cero final como si hace **strcpy** que copia hasta que encuentra el cero final incluyéndolo y de esta forma si en la variable había basura se mostrara como vemos en el ejemplo, después del cuarto carácter.

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

```
main(){
    funcion();
    getchar();
}
```

```
funcion(){

    char texto1[40], texto2[40], texto3[10];

    printf("Introduce un frase: ");
    gets(texto1);

    strcpy(texto2, texto1);
    printf("Una copia de tu texto es %s\n", texto2);
    strncpy(texto3, texto1, 4);
    texto3[4] = '\0';
    printf("Y sus 4 primeras letras son %s\n", texto3);

}
```

Allí le ponemos el cero final manualmente y si lo ejecutamos.

```
C:\Documents and Settings\ricnar\Escritorio\Un
Introduce un frase: supertexto
Una copia de tu texto es supertexto
Y sus 4 primeras letras son supe
```

Vemos que el cero que colocamos al final corta la string, de esta forma no importa que haya habido basura anterior.

```

; Attributes: bp-based frame

funcion_investigada proc near

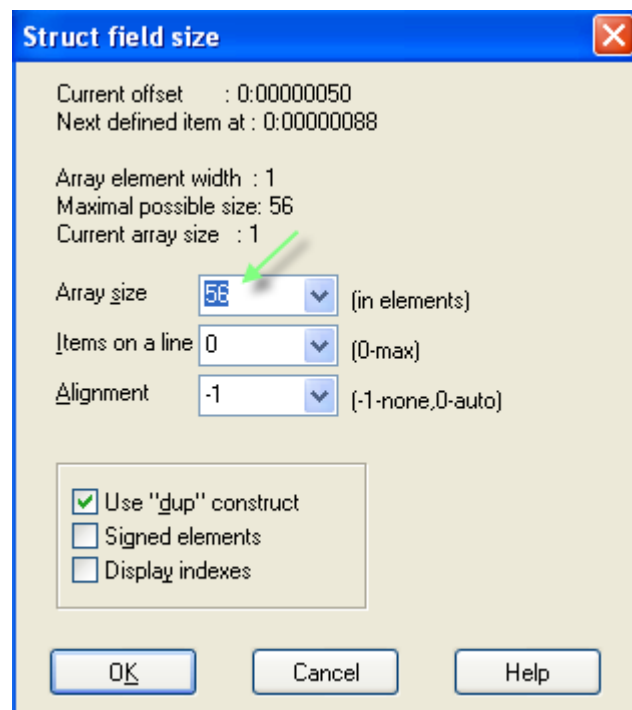
var_78= byte ptr -78h
var_74= byte ptr -74h
var_68= byte ptr -68h
var_38= byte ptr -38h

push    ebp
mov     ebp, esp
sub     esp, 88h
mov     dword ptr [esp], offset aIntroduceUnFra ;
call    printf
lea     eax, [ebp+var_38]
mov     [esp], eax      ; char *
call    gets
lea     eax, [ebp+var_38]
mov     [esp+4], eax    ; char *
lea     eax, [ebp+var_68]

```

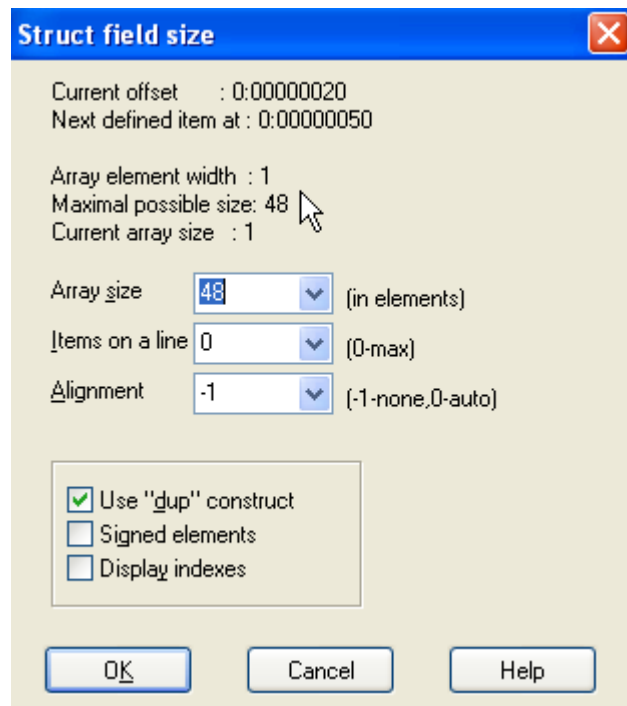
Aquí vemos cuatro variables, nosotros creamos 3 que son arrays de caracteres, así que vayamos observando con cuidado para crear los arrays en forma correcta, la primera variable que vemos que usa es **var_38** que es para guardar el primer texto que tipea el usuario, en nuestro código dicha variable se llama **texto1** y tiene 40 caracteres de largo incluyendo el cero final.

Así que vamos a la tabla de variables con doble click en una de ellas.



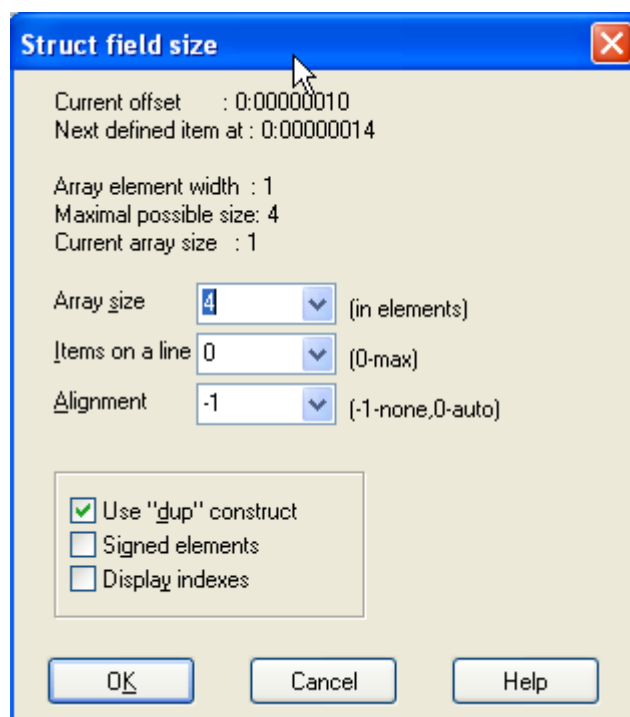
Vemos que usando todo el espacio disponible hasta el **stored ebp**, este array podría tener de largo 56 y no habría problema, pues no pisa nada, si reverseamos y no sabemos el largo exacto porque no tenemos el código fuente, poner 56 sería correcto, pues abarcamos todo el espacio disponible y funcionaría igual, como sabemos que es 40 el largo lo haremos así pero ambas posibilidades son correctas.

La siguiente variable que usa es **var_68** que la usa en el **strcpy** para obtener una copia del texto tipeado en otra variable, en nuestro código se llama **texto2** y también su largo es 40.



El espacio máximo posible es 48 sin pisar nada, podríamos dejarlo así, pero ponemos 40 para quedar igual al código fuente.

La tercer variable es la que guardara los 4 bytes y esa es **texto3** que tiene largo de 10 caracteres.



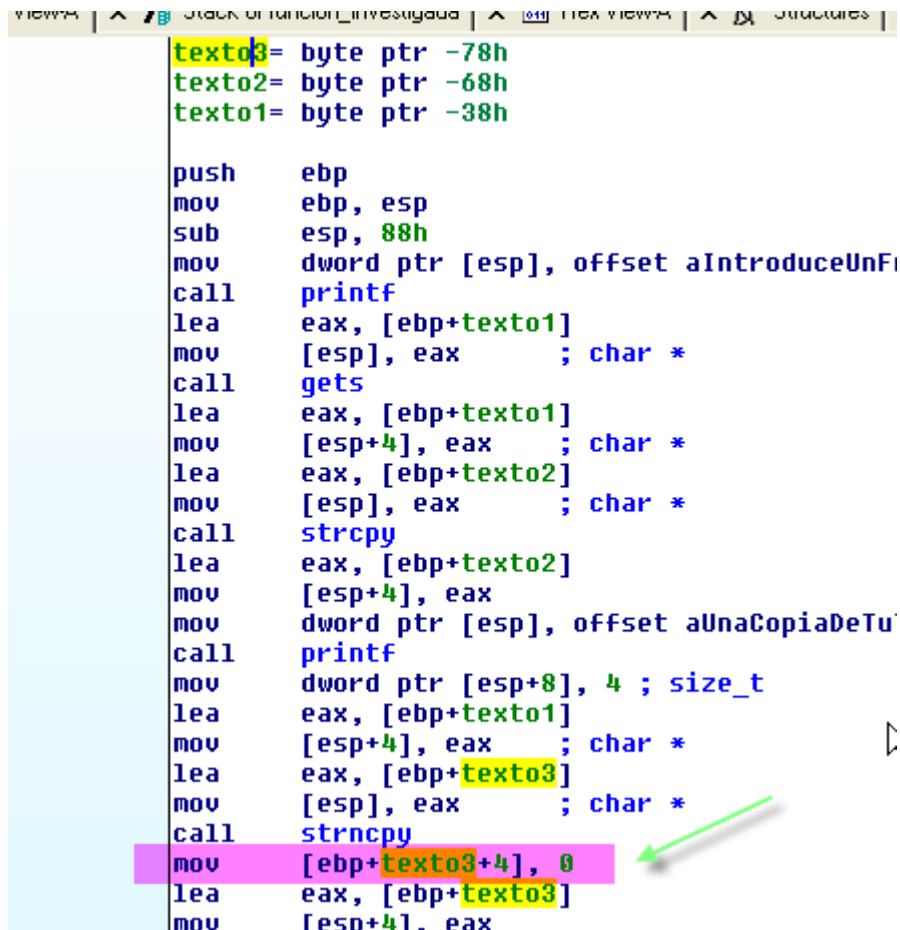
Vemos que sin pisar otra variable tendría como largo 4 ya que a continuación viene la **var_74** que en realidad es el quinto carácter de nuestro array **texto3** que usa para ponerle el cero final, así que si abarcamos con el tamaño igual a 10, nos quedaremos con solo 3 variables, y la cuarta **var_74** desaparecerá y pasara a ser un campo del array, donde se pone el cero.

```

00000079      db ? ; undefined
-00000079      db ? ; undefined
-00000078 texto3      db 10 dup(?)
-0000006E      db ? ; undefined
-0000006D      db ? ; undefined
-0000006C      db ? ; undefined
-0000006B      db ? ; undefined
-0000006A      db ? ; undefined
-00000069      db ? ; undefined
-00000068 texto2      db 40 dup(?)
-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 texto1      db 40 dup(?)
-00000010      db ? ; undefined

```

Ahora si vemos el código:



```

view74  Stack of function Investigada  0049 Hex view74  Structures
texto3= byte ptr -78h
texto2= byte ptr -68h
texto1= byte ptr -38h

push    ebp
mov     ebp, esp
sub     esp, 88h
mov     dword ptr [esp], offset aIntroduceUnFi
call    printf
lea     eax, [ebp+texto1]
mov     [esp], eax          ; char *
call    gets
lea     eax, [ebp+texto1]
mov     [esp+4], eax        ; char *
lea     eax, [ebp+texto2]
mov     [esp], eax          ; char *
call    strcpy
lea     eax, [ebp+texto2]
mov     [esp+4], eax
mov     dword ptr [esp], offset aUnaCopiaDeTu
call    printf
mov     dword ptr [esp+8], 4 ; size_t
lea     eax, [ebp+texto1]
mov     [esp+4], eax        ; char *
lea     eax, [ebp+texto3]
mov     [esp], eax          ; char *
call    strncpy
mov     [ebp+texto3+4], 0
lea     eax, [ebp+texto3]
mov     [esp+4], eax

```

Vemos que ahora si tenemos nuestras tres variables tipo array como en el código fuente y al convertir **texto3** en array desapareció la variable de mas que usaba para modificar un campo intermedio del mismo para poner un cero en el 5 lugar, ahora mostrándose como campo del mismo array.

Puse dos ejercicios con arrays para reversear a ver quien me manda a mi privado el código fuente

aproximado reverseado.

Hasta la parte siguiente
Ricardo Narvaja