

FUNCIONES-PARAMETROS-VARIABLES LOCALES Y GLOBALES – VALORES DE RETORNO.

Hemos realizado algunos ejemplos hasta ahora con una función principal main que llamaba a otra función que hacia todo el trabajo, pero en general los programas dividen el trabajo en muchas funciones que hacen pequeñas partes cada una.

Del Curso de Cabanes, sobre la importancia de realizar varias funciones que realicen partes del trabajo en vez de una sola muy complicada.

A partir de ahora vamos a empezar a intentar descomponer los problemas en trozos más pequeños, que sean más fáciles de resolver. Esto nos puede suponer varias ventajas:

- Cada "trozo de programa" independiente será más fácil de programar, al realizar una función breve y concreta.
- El "programa principal" será más fácil de leer, porque no necesitará contener todos los detalles de cómo se hace cada cosa.
- Podremos repartir el trabajo, para que cada persona se encargue de realizar un "trozo de programa", y finalmente se integrará el trabajo individual de cada persona.

Esos "trozos" de programa son lo que suele llamar "subrutinas", "procedimientos" o "funciones". En el lenguaje C, el nombre que más se usa es el de **funciones**.

Veamos como ejemplo un programa como el que hemos realizado anteriormente:

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

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

```
funcion{
```

```
    int suma;           /* Un entero que será la suma */
```

```
    int numero0 = 200;   /* Les damos valores */
```

```
    int numero1 = 150;
```

```
    int numero2 = 100;
```

```
    int numero3 = -50;
```

```
    int numero4 = 300;
```

```
    suma = numero0 + numero1 + numero2 + numero3 + numero4;
```

```
    printf("Su suma es %d", suma);
```

```
}
```

Vemos que se llama a funcion dentro de main y la misma esta declarada después, en DEV C++ esto funciona perfectamente pero es bueno saber que otros compiladores no permiten hacer esto, y la forma correcta para que funcione es siempre es declarar las funciones luego de los include..

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

```
funcion(){
```

```
    int suma;          /* Un entero que será la suma */
```

```
    int numero0 = 200;  /* Les damos valores */
```

```
    int numero1 = 150;
```

```
    int numero2 = 100;
```

```
    int numero3 = -50;
```

```
    int numero4 = 300;
```

```
    suma = numero0 + numero1 + numero2 + numero3 + numero4;
```

```
    printf("Su suma es %d", suma);
```

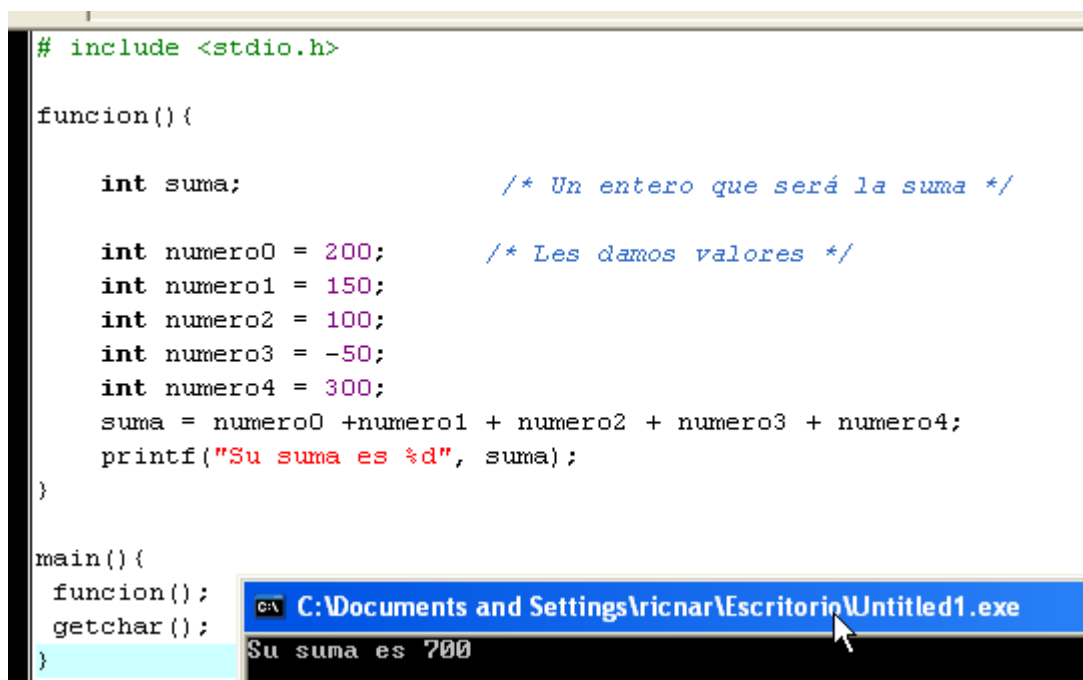
```
}
```

```
main(){
```

```
    funcion();
```

```
    getchar();
```

```
}
```

A screenshot of a C program being executed. The code is displayed in a text editor with syntax highlighting. The program defines a function 'funcion()' that calculates the sum of five integers: numero0 (200), numero1 (150), numero2 (100), numero3 (-50), and numero4 (300). The sum is stored in 'suma' and printed using 'printf'. The 'main()' function calls 'funcion()' and then 'getchar()'. Below the code, a command prompt window shows the execution path 'C:\Documents and Settings\ricnar\Escritorio\Untitled1.exe' and the output 'Su suma es 700'.

Vemos que funciona igual, así que dado que esta forma es mas genérica y aceptada por todos los compiladores, la adoptaremos y definiremos al inicio después de los includes, todas las funciones antes de usarlas.

Si compilan y ven en el IDA verán que quedara similar a como lo hacíamos antes, no cambia nada.

PARAMETROS DE UNA FUNCION

Si trabajamos en forma modular, cada funcion realizara pequeñas tareas, es muy probable que debamos pasarle parametros o argumentos para que realice la misma, veamos este ejemplo, una funcion que le pasamos un numero real y lo imprime en pantalla.

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

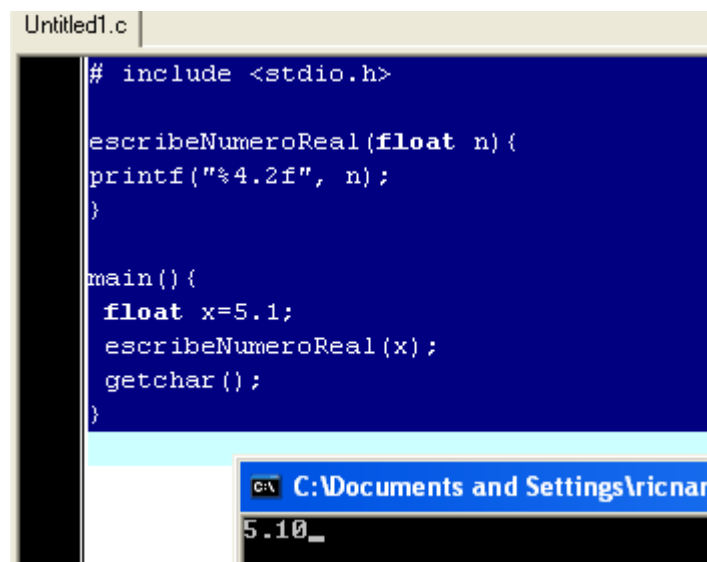
```
escribeNumeroReal(float n){  
printf("%4.2f", n);  
}
```

```
main(){  
float x=5.1;  
escribeNumeroReal(x);  
getchar();  
}
```

Allí vemos que al definir la función **escribeNumeroReal** entre paréntesis se colocara el parámetro que recibirá definiendo al mismo tiempo el tipo de datos del mismo, en este caso **float n**, y cuando se llame a la función de otra parte del programa se colocara entre paréntesis en este caso una variable **float** que le pasamos o una constante que debe coincidir en tipo con la esperada por la función, para que la misma lo acepte y lo imprima, el cual es el parámetro o argumento pasado a la misma.

```
float x=5.1;  
escribeNumeroReal(x);
```

Es necesario que se le puedan pasar parámetros a las funciones, porque como ya vimos las variables locales solo tienen sentido dentro de la función donde están definidas, así que para poder pasar valores entre funciones, se utilizan los parámetros, y también lo que veremos luego, los valores de retorno.

A screenshot of a C program being executed. The top part shows the source code in a text editor window titled 'Untitled1.c'. The code defines a function 'escribeNumeroReal' that takes a 'float n' and prints it with 4.2 decimal places. The 'main' function calls this function with the value 5.1. The bottom part of the image shows the command prompt window with the path 'C:\Documents and Settings\ricnar' and the output '5.10_'.

```
Untitled1.c  
# include <stdio.h>  
  
escribeNumeroReal(float n){  
printf("%4.2f", n);  
}  
  
main(){  
float x=5.1;  
escribeNumeroReal(x);  
getchar();  
}
```

C:\Documents and Settings\ricnar
5.10_

Al compilarlo y ejecutarlo vemos que imprime el valor del **float**, veámoslo en IDA.

Vemos el main y vemos lo resaltado en la imagen en verde que es lo único diferente a los main que veníamos haciendo que solo llamaban a una función sin parámetros.

```

argv= dword ptr 00h
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     eax, 40A33333h
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
mov     [esp], eax ; float
call    sub_401290
call    getchar
leave
retn

```

include <stdio.h>

```

escribeNumeroReal(float n){
printf("%.4f", n);
}

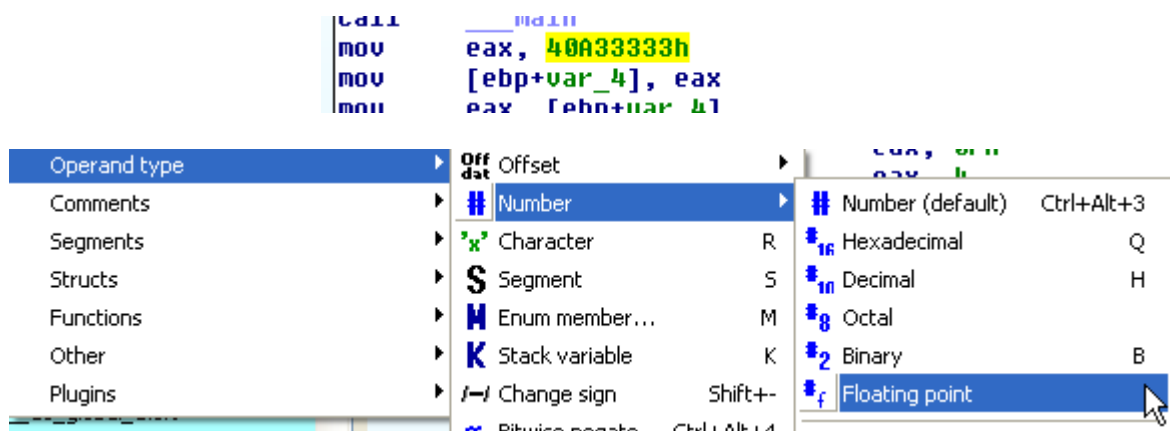
```

```

main(){
float x=5.1;
escribeNumeroReal(x);
getchar();
}

```

Vemos que en el main ademas de las variables y argumentos que creaba el compilador siempre, hay una variable local mas llamada **var_4**, vemos que se inicializa con el valor **40A33333h**, vayamos al menú de IDA,



```

mov     eax, 5.09999999 |
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
mov     [esp], eax      ; float
call    sub_401290
call    getchar

```

Bueno redondeando es el float **5.1** en el código la variable float que se inicializa con dicho valor se llama **x**, así que la renombramos.

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

```

escribeNumeroReal(float n){
printf("%4.2f", n);
}

```

```

main(){
float x=5.1;
escribeNumeroReal(x);
getchar();
}

```

```

call    __main
mov     eax, 5.09999999
mov     [ebp+x], eax
mov     eax, [ebp+x]
mov     [esp], eax      ; float
call    sub_401290
call    getchar

```

Ahora esa variable **x** es una variable absolutamente local que tiene validez solo dentro del main, si dentro de la funcion en **401290** quisiera usar el valor float **5.1**, al llamar a la variable **x**, esta no existirá y me dará error, para pasar valores a funciones como vimos se usan los parámetros, y según el compilador los mismos se pushean antes de la llamada a la funcion, o como este compilador en vez de **pushear** los guarda en el stack, es similar **PUSH EAX** a hacer **MOV [ESP], EAX**, aunque a mi me gustan mas los compiladores que pushean los parametros para mi se ve mas claro.

```

PUSH EBX
PUSH EAX
call    sub_401290

```

Uno ve los PUSH y sabe por la cantidad los parametros de una funcion a simple vista, mientras que pasarlos con

```

mov     [esp+4], eax
mov     eax, [ebp+x]
mov     [esp], eax
call    sub_401290

```

Es menos visual para mi gusto, pero bueno entremos a la funcion.

```

; int __cdecl sub_401290(float)
sub_401290 proc near
    arg_0= dword ptr 8

    push    ebp |
    mov     ebp, esp
    sub     esp, 18h
    fld     [ebp+arg_0]
    fstp    qword ptr [esp+4]
    mov     dword ptr [esp], offset a4_2f ; "%4.2f"
    call    printf
    leave
    retn
sub_401290 endp

```

Si hacemos doble click en **arg_0**, recordemos que la zona de **variables locales** era por encima del **stored_ebp** y del **return address**, en este caso no hay variables locales por eso no hay nada por encima.

```

+00000000 s          db 4 dup(?)
+00000004 r          db 4 dup(?)
+00000008 arg_0      dd ?
+0000000C
+0000000C ; end of stack variables

```

Vemos el primer parametro **arg_0**, la zona de parametros estará siempre por **DEBAJO** del **return address**, lo cual tiene cierta lógica pues si pusheo el parametro antes de entrar a la funcion, este se guarda en el stack, y luego al entrar a la funcion se guarda encima del mismo el **return address** (después de todo es un CALL y al entrar en el siempre se guarda el **return address**), y luego comenzara la funcion generalmente con PUSH EBP, y ahí guarda el stored ebp, y luego declarara las variables arriba del mismo.

Así que si hay un solo parametro IDA lo llamara **arg_0** y sera el que se pushea justo antes de llamar a la funcion, si hay mas parametros como en este otro caso:

```

PUSH EBX
PUSH EAX
call sub_401290

```

PUSH EBX es ejecutado primero por lo tanto su valor estará mas abajo en el stack que PUSH EAX en este caso IDA llamara a este segundo parametro **arg_4** y estará debajo de **arg_0** y así sucesivamente cuantos mas parametros tenga crecerá hacia abajo. (**arg_8**, **arg_C**)

```

+00000000 s          db 4 dup(?)
+00000004 r          db 4 dup(?)
+00000008 arg_0      dd ?
+0000000C [redacted] ← arg_4
+0000000C ; end of stack variables

```

Allí en el rectángulo rojo iría un segundo parametro **arg_4** si lo hubiera, bueno volvamos a nuestra

funcion.

Vimos que los parametros se utilizan para pasar valores entre funciones, ya que las variables locales no sirven para ello, así que al reversear es importante determinar, cual es el valor que se le esta pasando y renombrarla de acuerdo a eso, en este caso sabemos que **arg_0** tiene el mismo valor que la variable **x** del main, así que podría renombrarla como **x** ya que es un parametro que tiene su valor.

```
; int __cdecl sub_401290(float x)
sub_401290 proc near

x= dword ptr 8


push    ebp
mov     ebp, esp
sub     esp, 18h
fld     [ebp+x]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset a4_2f ; "%4.
call    printf
leave
retn
sub_401290 endp
```

Es importante que el nuevo nombre aparezca en la definición de la funcion

```
; int __cdecl sub_401290(float x)
sub_401290 proc near

x= dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 18h
fld     [ebp+x]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset a4_2f ; "%4.2f"
call    printf
leave
retn
sub_401290 endp
```



Si no aparece y queda así

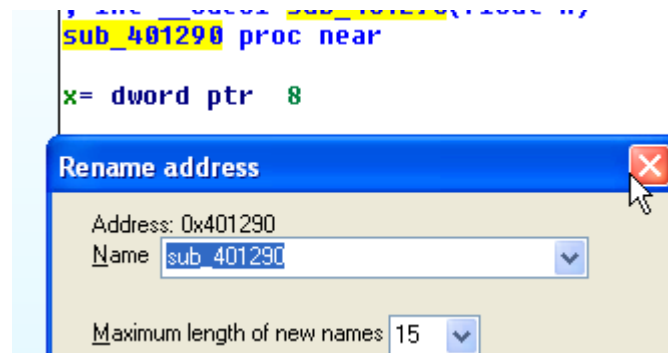
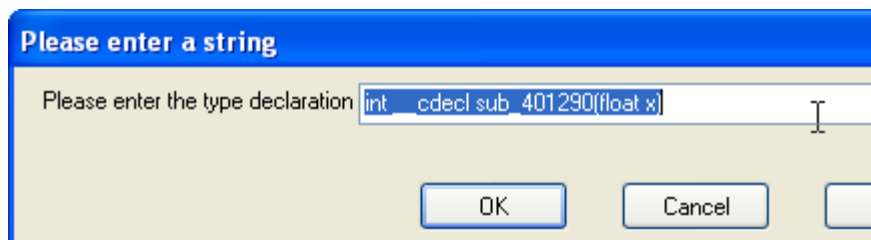
```

sub_401290 proc near
|
x= dword ptr 8

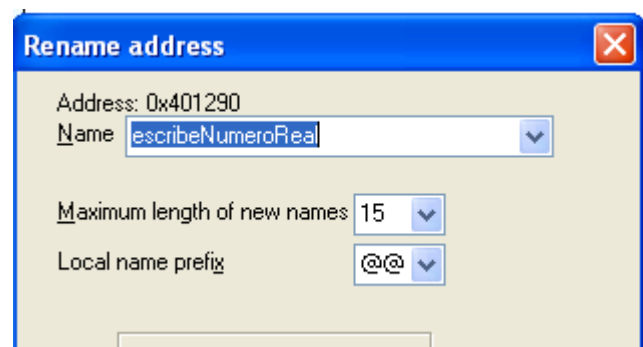
push    ebp
mov     ebp, esp
sub     esp, 18h
fld     [ebp+x]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset a4_2f ; ""4
call    printf
leave
retn
sub_401290 endp

```

Hacemos click derecho SET FUNCION TYPE y lo arreglara tratando de usar los parametros renombrados.



A la vez podemos renombrar la funcion a



y allí quedara mejor


```

; int __cdecl escribeNumeroReal(float x)
escribeNumeroReal proc near

x= dword ptr 8

push    ebp
mov     ebp, esp
sub     esp, 18h
fld     [ebp+x]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset a4_2f ; "%4.2f"
call    printf
leave
retn
escribeNumeroReal endp

```

Si hemos hecho aparecer el nuevo nombre del parametro en la definición de la funcion, al volver al main vemos que nos muestra el nombre que le pusimos dentro al parametro, y que como vemos coincide con el valor de la variable local **x**. De cualquier manera podría ponersele un nombre aclaratorio diferente ya que la funcion puede llamarse desde distintos lugares del programa y con distintos valores, que no siempre serán el de **x**, lo veremos en el proximo ejemplo.

```

mov     eax, [ebp+var_0]
call    __chkstk
call    __main
mov     eax, 5.09999999
mov     [ebp+x], eax
mov     eax, [ebp+x]
mov     [esp], eax ; x
call    escribeNumeroReal
call    getchar
leave
retn
_main endp

```

VALOR DE RETORNO DE UNA FUNCION

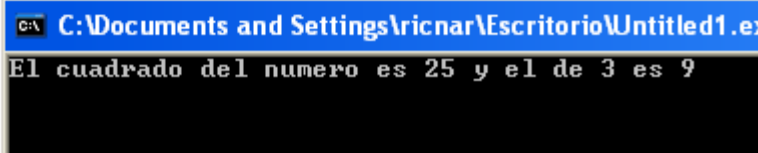
Ya vimos los parametros de una funcion para poder pasarle valores, pero hay casos en que necesitamos ademas de poder pasarle parametros, que la funcion nos devuelva algún resultado que necesitamos usar en otra parte del programa, por ejemplo si tengo una funcion que le paso dos int y la misma multiplica los dos valores, puedo necesitar que me devuelva ese resultado para usarlo.

```
#include <stdio.h>

int cuadrado ( int n ) {
    return n*n;
}

main() {
    int numero;
    int resultado;

    numero= 5;
    resultado = cuadrado(numero);
    printf("El cuadrado del numero es %d", resultado);
    printf(" y el de 3 es %d", cuadrado(3));
    getch();
}
```



Vemos que la funcion llamada **cuadrado**, tiene como parametro un **int n** y el valor de retorno que se encuentra a continuacion de la palabra **return** sera el valor de **n * n** o sea el cuadrado de **n**.

Vemos que la funcion es llamada desde el main desde aquí.

```
numero= 5;
resultado = cuadrado(numero);
```

Se llama a la funcion **cuadrado**, pero se guarda el valor que devuelve o **valor de retorno**, por lo cual cuando se define la funcion cuadrado se debe poner el tipo de valor que devolverá delante del nombre de la misma.

```
int cuadrado ( int n ) {
    return n*n;
}
```

Por lo tanto cuando se almacena la variable resultado debe ser declarada del mismo tipo o sea int también.

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

```
int cuadrado ( int n ) {
    return n*n;
}
```

```
main() {
    int numero;
    int resultado;
```

```

numero= 5;
resultado = cuadrado(numero);
printf("El cuadrado del numero es %d", resultado);
printf(" y el de 3 es %d", cuadrado(3));
getchar();
}

```

Así que deben coincidir el tipo del valor de retorno y el tipo de la variable local donde lo guardara.

Si una funcion esta especificada como tipo **void** significa que no devolverá ningún resultado.

Cuando queremos dejar claro que una función no tiene que devolver ningún valor, podemos hacerlo indicando al principio que el tipo de datos va a ser "void" (nulo). Por ejemplo, nuestra función "saludar", que se limitaba a escribir varios textos en pantalla, quedaría más correcta si fuera así:

```

void saludar() {
    printf("Bienvenido al programa\n");
    printf(" de ejemplo\n");
    printf("Bienvenido al programa\n");
}

```

Bueno volvamos a nuestro ejemplo

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

```

int cuadrado ( int n ) {
return n*n;
}

```

```

main() {
int numero;
int resultado;

```

```

numero= 5;
resultado = cuadrado(numero);
printf("El cuadrado del numero es %d", resultado);
printf(" y el de 3 es %d", cuadrado(3));
getchar();
}

```

Vemos que hay una segunda llamada a la funcion **cuadrado**, pasandole la constante 3 como parametro, en este caso no se le pasa el valor de una variable sino directamente una constante, que lógicamente debe coincidir con el tipo esperado por la funcion.

El valor devuelto por la misma en este caso no es almacenado sino usado directamente dentro del **printf** que espera el valor de retorno para realizar el **format string** e imprimir el resultado del cuadrado de 3.

```

numero= 5;
resultado = cuadrado(numero);
printf("El cuadrado del numero es %d", resultado);
printf(" y el de 3 es %d", cuadrado(3));
getchar();

```

C:\Documents and Settings\ricnar\Escritorio\Untitled1.exe

El cuadrado del numero es 25 y el de 3 es 9

Compilemos y veamoslo en IDA.

```

call    __main
mov     [ebp+var_4], 5
mov     eax, [ebp+var_4]
mov     [esp], eax
call    sub_401290
mov     [ebp+var_8], eax
mov     eax, [ebp+var_8]
mov     [esp+4], eax
mov     dword ptr [esp], offset aElCuadradoDe1N ; "El cuadrado de
call    printf
mov     dword ptr [esp], 3
call    sub_401290
mov     [esp+4], eax
mov     dword ptr [esp], offset aYElDe3EsD ; " y el de 3 es %d"
call    printf

```

Vemos allí que la funcion es llamada dos veces, la primera es aquí y **var_4** corresponde a la variable **numero** que se inicializa con el valor 5 y se le pasa como parametro.

```

call    __main
mov     [ebp+var_4], 5
mov     eax, [ebp+var_4]
mov     [esp], eax
call    sub_401290

```

Renombrando

```

call    __main
mov     [ebp+numero], 5
mov     eax, [ebp+numero]
mov     [esp], eax
call    sub_401290

```

Vemos que le pasa como parametro el valor de **numero** que es 5, entremos a la funcion.

```

; Attributes: bp-based frame

sub_401290 proc near
    arg_0= dword ptr 8

    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+arg_0]
    imul    eax, [ebp+arg_0]
    pop     ebp
    retn
sub_401290 endp

```

Vemos que el parametro único sera **arg_0** la funcion lee este valor lo pasa a EAX y luego lo multiplica por si mismo devolviendo en **EAX** el valor de retorno. Podemos renombrar a la funcion como cuadrado.

```

; Attributes: bp-based frame

cuadrado proc near
    arg_0= dword ptr 8

    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+arg_0]
    imul    eax, [ebp+arg_0]
    pop     ebp
    retn
cuadrado endp

```

Allí vemos el valor de retorno que devolverá en EAX.

Ahora el tema es el nombre del parametro ya que si le pongo el nombre **numero** pues en la primera llamada tiene ese valor, en la segunda vez que la llame cuando tenga el valor 3, ya no coincidirá.

```

call    __call
mov     [ebp+numero], 5
mov     eax, [ebp+numero]
mov     [esp], eax
call    cuadrado
mov     [ebp+var_8], eax
mov     eax, [ebp+var_8]
mov     [esp+4], eax
mov     dword ptr [esp], offset a
call    printf
mov     dword ptr [esp], 3
call    cuadrado
mov     [esp+4], eax

```

En estos casos es difícil aconsejar pues a veces nosotros al reversear necesitamos seguir un parametro que vemos en una funcion, hacia las funciones padres y muchas veces conviene ponerle un nombre e ir hacia atrás con el mismo nombre, para no perdernos y poder seguir identificando el mismo valor que nos interesa.

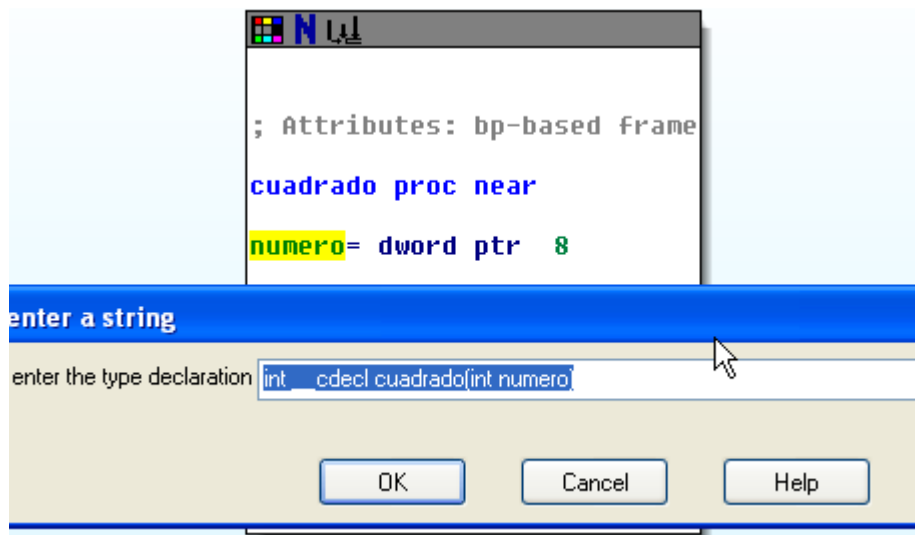
```

cuadrado proc near
    numero= dword ptr 8

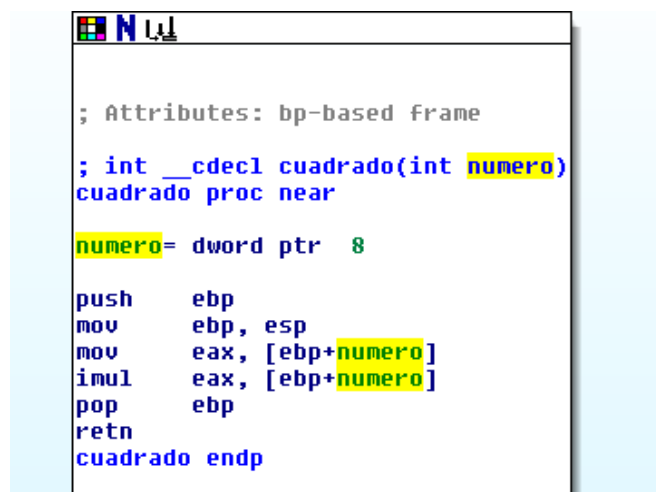
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+numero]
    imul    eax, [ebp+numero]
    pop     ebp
    retn
cuadrado endp

```

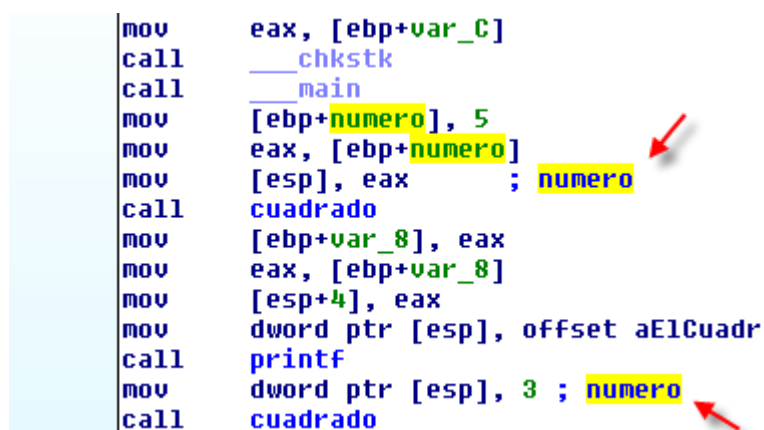
Ahora propago el nombre haciendo click derecho – SET UNION TYPE.



Quedara asi



Si vamos al main aparece el nombre que le pusimos, y en la primera llamada coinciden ya que toma el valor de la variable **numero**, en la segunda bueno, igual me da la idea que al parametro **numero** interno se le pasa el valor 3, ya se que el nombre del parametro interno es una ayuda y que no necesariamente tiene que coincidir 100% con las variables locales del main, pero es bueno que tenga el nombre alguna relación para orientarnos.



Hay reversers que no le ponen exactamente el mismo nombre sino que le ponen por ejemplo con

mayúsculas.

```
; Attributes: bp-based frame

; int __cdecl cuadrado(int NUMERO)
cuadrado proc near
    NUMERO = dword ptr 8

    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+NUMERO]
    imul    eax, [ebp+NUMERO]
    pop     ebp
    retn
cuadrado endp
```

```
call    __chkstk
call    __main
mov     [ebp+numero], 5
mov     eax, [ebp+numero]
mov     [esp], eax ; NUMERO
call    cuadrado
mov     [ebp+var_8], eax
mov     eax, [ebp+var_8]
mov     [esp+4], eax
mov     dword ptr [esp], offset aElCu
call    printf
mov     dword ptr [esp], 3 ; NUMERO
call    cuadrado
mov     [esp+4], eax
mov     dword ptr [esp], offset aYE1D
```

O por ahí alguna convención propia como **Arg_numero** o algo así.

Vemos que en la primera llamada el valor de retorno de la función que devuelve en **EAX** lo guarda en **var_8** así que renombraremos la misma a **resultado**.

```
call    __chkstk
call    __main
mov     [ebp+numero], 5
mov     eax, [ebp+numero]
mov     [esp], eax ; NUMERO
call    cuadrado
mov     [ebp+var_8], eax
mov     eax, [ebp+var_8]
mov     [esp+4], eax
mov     dword ptr [esp], offset aElCuadradoDe1N ; "El cuadrado de
call    printf
mov     dword ptr [esp], 3 ; NUMERO
call    cuadrado
mov     [esp+4], eax
mov     dword ptr [esp], offset aYE1De3EsD ; " y el de 3 es %d"
call    printf
```

Luego ese valor guardado en **resultado** lo pasa como parametro de **printf** para imprimirlo.


```

call    _main
mov     [ebp+numero], 5
mov     eax, [ebp+numero]
mov     [esp], eax      ; NUMERO
call    cuadrado
mov     [ebp+resultado], eax
mov     eax, [ebp+resultado]
mov     [esp+4], eax
mov     dword ptr [esp], offset aElCuadradoDe1N ; "El cuadrado del
--

```

En la segunda llamada vemos que no guarda el valor de retorno en ninguna variable directamente lo guarda en el stack como parametro de **printf** directamente, que era lo que pasaba en nuestro código fuente.

```

call    printf
mov     dword ptr [esp], 3 ; NUMERO
call    cuadrado
mov     [esp+4], eax
mov     dword ptr [esp], offset aYElDe3EsD ; " y el de 3 es %d"
call    printf

```

printf(" y el de 3 es %d", cuadrado(3));

La siguiente definición se la copiamos a Cabanes

Variables locales y variables globales

Las variables se pueden declarar dentro de un bloque (una función), y entonces sólo ese bloque las conocerá, no se podrán usar desde ningún otro bloque del programa. Es lo que llamaremos “**variables locales**”.

Por el contrario, si declaramos una variable al comienzo del programa, fuera de todos los “bloques” de programa, será una “**variable global**”, a la que se podrá acceder desde cualquier parte.

En general, deberemos intentar que la mayor cantidad de variables posible sean locales (lo ideal sería que todas lo fueran). Así hacemos que cada parte del programa trabaje con sus propios datos, y ayudamos a evitar que un error en un trozo de programa pueda afectar al resto. La forma correcta de pasar datos entre distintos trozos de programa es usando los parámetros de cada función y los valores de retorno.

En el ejemplo anterior

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

```

int cuadrado ( int n ) {
return n*n;
}

```

```

main() {
int numero;
int resultado;

```

```

numero= 5;
resultado = cuadrado(numero);
printf("El cuadrado del numero es %d", resultado);
printf(" y el de 3 es %d", cuadrado(3));
getchar();
}

```

si lo cambiamos a

The screenshot shows a C program in a text editor window titled 'Untitled1.c'. The code defines a function 'cuadrado' that takes an integer 'n' and returns 'n*n'. In the 'main' function, 'numero' is set to 5, 'resultado' is calculated as 'cuadrado(numero)', and two printf statements are used to display the results. The output window shows the execution of 'C:\Documents and Settings\ricnar\Escritorio\Untitled1.exe' with the output: 'El cuadrado del numero es 25 y el de 3 es 9'.

```

#include <stdio.h>

int numero;
int resultado;
int cuadrado ( int n ) {
    return n*n;
}

main() {

    numero= 5;
    resultado = cuadrado(numero);
    printf("El cuadrado del numero es %d", resultado);
    printf(" y el de 3 es %d", cuadrado(3));
    getchar();
}

```

C:\Documents and Settings\ricnar\Escritorio\Untitled1.exe
El cuadrado del numero es 25 y el de 3 es 9

Vemos que **numero** y **resultado** son **variables globales** por eso son reconocidas en cualquier parte del programa.

The screenshot shows a snippet of assembly code. It includes instructions for stack setup, calling the main function, and moving values into registers. The key part is the declaration of global variables: 'ds:dword_404060, 5' and 'ds:dword_404070, eax', which correspond to the 'numero' and 'resultado' variables in the C code.

```

mov     eax, [esp+4]
call    __chkstk
call    __main
mov     ds:dword_404060, 5
mov     eax, ds:dword_404060
mov     [esp], eax
call    sub_401290
mov     ds:dword_404070, eax
mov     eax, ds:dword_404070
mov     [esp+4], eax

```

Allí vemos las dos variables globales, podemos renombrarlas igual, si hacemos doble click en el nombre nos lleva a la sección **bss** adonde se guardan.

```

call    _main
mov     ds:numero, 5
mov     eax, ds:numero
mov     [esp], eax
call    sub_401290
mov     ds:resultado, eax
mov     eax, ds:resultado
; .bss:00404070

; .bss:00404054          align 10h
; .bss:00404060 numero  dd ?           ; DATA XREF: _main+2A↑w
; .bss:00404060          ; _main+34↑r
; .bss:00404064          align 10h
; .bss:00404070 resultado dd ?         ; DATA XREF: _main+41↑w
; .bss:00404070          ; _main+46↑r
; .bss:00404074          align 10h

```

La funcion cuadrado no ha cambiado ya que se le pasan los valores como parametro.

```

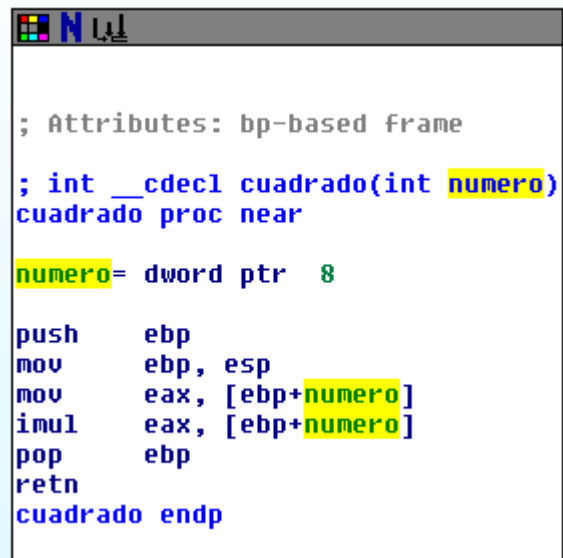
; Attributes: bp-based frame

sub_401290 proc near
    arg_0= dword ptr 8

    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+arg_0]
    imul    eax, [ebp+arg_0]
    pop     ebp
    retn
sub_401290 endp

```

Arreglándola



```
; Attributes: bp-based frame

; int __cdecl cuadrado(int numero)
cuadrado proc near

numero= dword ptr 8

push    ebp
mov     ebp, esp
mov     eax, [ebp+numero]
imul    eax, [ebp+numero]
pop     ebp
retn
cuadrado endp
```

Volviendo al main el valor de retorno se guarda en resultado.

```

mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    ___chkstk
call    ___main
mov     ds:numero, 5
mov     eax, ds:numero
mov     [esp], eax      ; numero
call    cuadrado
mov     ds:resultado, eax
mov     eax, ds:resultado
mov     [esp+4], eax
mov     dword ptr [esp], offset aElCuadrado
call    printf
mov     dword ptr [esp], 3 ; numero
call    cuadrado
mov     [esp+4], eax
mov     dword ptr [esp], offset aYE1De3EsD
call    printf
call    getchar
leave

```

De cualquier forma dentro de cuadrado podríamos acceder a las variables globales directamente.

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

```

int numero;
int resultado;
void cuadrado(){
printf("El cuadrado del numero es %d", (numero*numero));
}

```

```
main() {
```

```

numero= 5;
cuadrado();
getchar();
}

```

En este ejemplo **numero** es una variable global y la funcion **cuadrado** no tiene parametros ni valor de retorno, si lo compilo y veo en IDA.

```

call    ___chkstk
call    ___main
mov     ds:dword_404060, 5
call    sub_401290
call    getchar
leave
retn

```

Allí veo la variable global **numero** la renombro.

```

mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    ___chkstk
call    __main
mov     ds:numero, 5
call    sub_401290
call    getchar
leave
retn
_main endp

```

Veo que la funcion cuadrado no tiene parametros si entro en ella,

```

; Attributes: bp-based frame

cuadrado proc near
push    ebp
mov     ebp, esp
sub     esp, 8
mov     eax, ds:numero
imul    eax, ds:numero
mov     [esp+4], eax
mov     dword ptr [esp], offset aElCuadradoDe1N ; "El cuadrado del numero es %d"
call    printf
leave
retn
cuadrado endp

```

Veo que no tiene argumentos ni variables y que usa el valor de la variable global **numero** que esta guardado en la sección **bss**.

.bss:00404050	ddword_404050	dd ?	; DATA XREF: __w32_snareqptr
.bss:00404054		align 10h	
.bss:00404060	numero	dd ?	; DATA XREF: cuadrado+61r
.bss:00404060			; cuadrado+B1r ...
.bss:00404064		db ? ;	
.bss:00404065		db ? ;	
bss:00404066		db ? ;	

También me muestra a la derecha la referencia desde donde es llamada dicha variable.

```

* .bss:00404050 dword_404050 dd ? ; DATA XREF: __w32_sharedpti
* .bss:00404054 align 10h
* .bss:00404060 numero | dd ? ; DATA XREF: cuadrado+01r
; ===== S U B R O U T I N E =====
; Attributes: bp-based frame

cuadrado proc near ; CODE XREF: _main+34↓p
    push ebp
    mov ebp, esp
    sub esp, 8
    mov eax, ds:numero
    imul eax, ds:numero
    .bss:0040406E db ? ;
    .bss:0040406F db ? ;
    .bss:00404070 db ? ;

```

Bueno esto es todo por ahora con el tema funciones no les daré ejercicios aun, los dejare descansar un poco.

Ricardo Narvaja