

# Práctica de Organización del Computador II

## Stack Frame

---

Segundo Cuatrimestre 2022

Organización del Computador II  
DC - UBA

## Uso de la pila y ejemplos de interacción

---

Anteriormente, cuando hablamos de la temporalidad de los datos, mencionamos que íbamos a tener datos temporales que iban a ubicarse en la pila.

Anteriormente, cuando hablamos de la temporalidad de los datos, mencionamos que íbamos a tener datos temporales que iban a ubicarse en la pila.

Ahora vamos a ver que la forma de acceder a los datos temporales y a los registros pasados por pila va a ser en direcciones relativas al registro RBP ,que apunta a la base actual de la pila.

Vamos a referirnos a dos partes del código de nuestra función de ASM con nombres distinguidos:

Vamos a referirnos a dos partes del código de nuestra función de ASM con nombres distinguidos:

- Prólogo: es donde se reserva espacio en la pila para datos temporales, se agrega padding para mantenerla alineada a 16 bytes y se preserva los valores de los registros no volátiles.

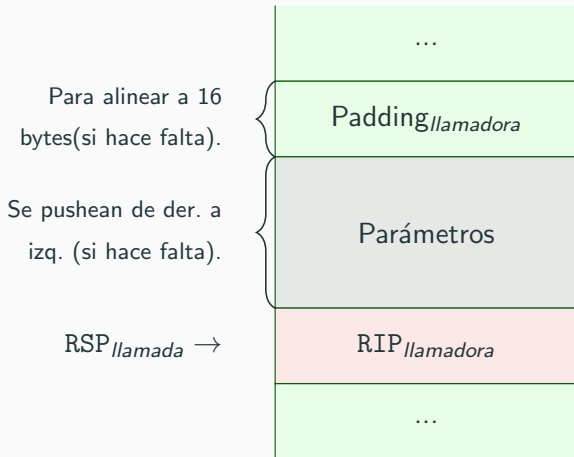
Vamos a referirnos a dos partes del código de nuestra función de ASM con nombres distinguidos:

- Prólogo: es donde se reserva espacio en la pila para datos temporales, se agrega padding para mantenerla alineada a 16 bytes y se preserva los valores de los registros no volátiles.
- Epílogo: es donde restauramos los valores de los registros no volátiles y devolvemos la pila a su estado inicial.

Veamos como recibimos los datos en relación a la pila:

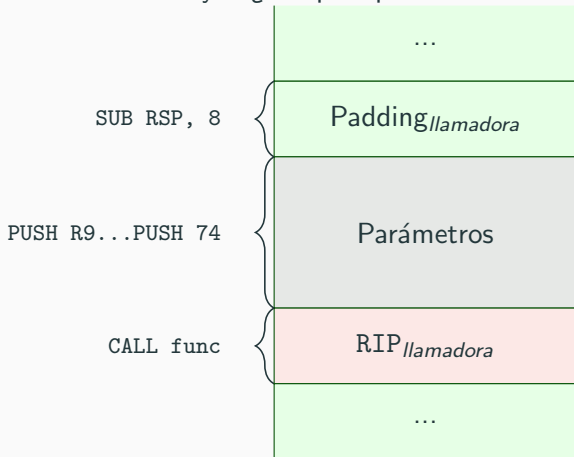


Veamos como recibimos los datos en relación a la pila:



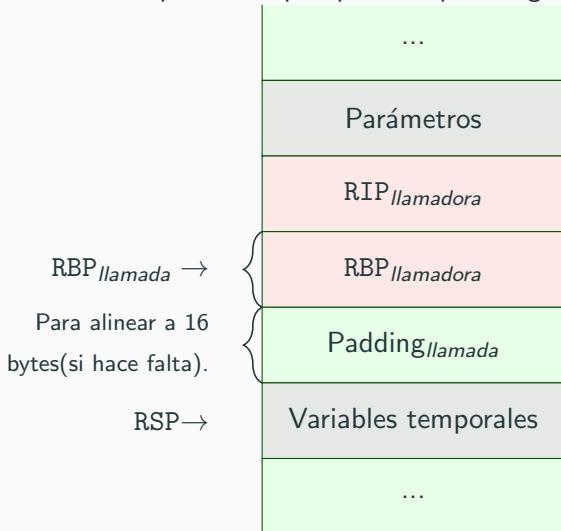
¿Cómo se construye? ¿De qué operación son resultado?

¿Cómo se construye? ¿De qué operación son resultado?



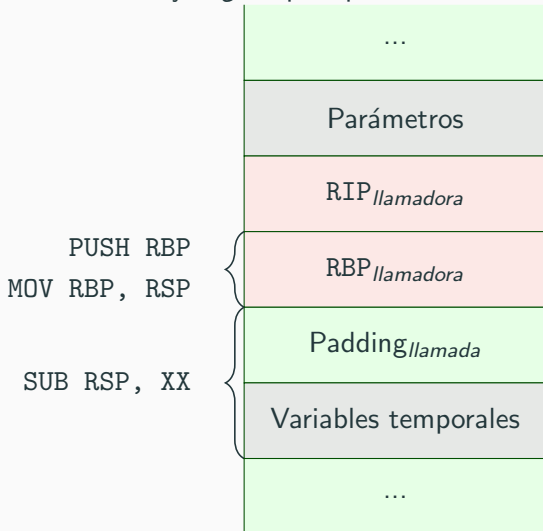
Veamos como queremos que quede la pila luego del prólogo:

Veamos como queremos que quede la pila luego del prólogo:



¿Cómo se construye? ¿De qué operación son resultado?

¿Cómo se construye? ¿De qué operación son resultado?



Supongan que deseamos llamar a la siguiente función desde main:

```
uint32_t sum_9(uint8_t x1, uint8_t x2, uint8_t x3,  
uint8_t x4, uint8_t x5, uint8_t x6, uint8_t x7,  
uint8_t x8, uint8_t x9)
```

¿Cómo pasamos los parámetros? En registros:



Supongan que deseamos llamar a la siguiente función desde main:

```
uint32_t sum_9(uint8_t x1, uint8_t x2, uint8_t x3,  
uint8_t x4, uint8_t x5, uint8_t x6, uint8_t x7,  
uint8_t x8, uint8_t x9)
```

¿Cómo pasamos los parámetros? En registros:

x1 → RDI	x2 → RSI	x3 → RDX	x4 → RCX
x5 → R8	x6 → R9		

Supongan que deseamos llamar a la siguiente función desde main:

```
uint32_t sum_9(uint8_t x1, uint8_t x2, uint8_t x3,  
uint8_t x4, uint8_t x5, uint8_t x6, uint8_t x7,  
uint8_t x8, uint8_t x9)
```

¿Cómo pasamos los parámetros? En registros:

x1 → RDI	x2 → RSI	x3 → RDX	x4 → RCX
x5 → R8	x6 → R9		

En la pila:

x7 → RBP + 0x10	x8 → RBP + 0x18
x9 → RBP + 0x20	

Supongan que deseamos llamar a la siguiente función desde `main`:

```
uint32_t sum_9(uint8_t x1, uint8_t x2, uint8_t x3,  
uint8_t x4, uint8_t x5, uint8_t x6, uint8_t x7,  
uint8_t x8, uint8_t x9)
```

Supongan que deseamos llamar a la siguiente función desde `main`:

```
uint32_t sum_9(uint8_t x1, uint8_t x2, uint8_t x3,  
uint8_t x4, uint8_t x5, uint8_t x6, uint8_t x7,  
uint8_t x8, uint8_t x9)
```

Corramos el siguiente comando para conseguir un volcado del código ASM que ejecutaría el procesador:

```
objdump -M intel -d main
```

Supongan que deseamos llamar a la siguiente función desde `main`:

```
uint32_t sum_9(uint8_t x1, uint8_t x2, uint8_t x3,  
uint8_t x4, uint8_t x5, uint8_t x6, uint8_t x7,  
uint8_t x8, uint8_t x9)
```

Corramos el siguiente comando para conseguir un volcado del código ASM que ejecutaría el procesador:

```
objdump -M intel -d main
```

Ahora veamos lo que conseguimos.

```
uint32_t sum_9(uint8_t x1[rdi], uint8_t x2[rsi], uint8_t x3[rdx],  
uint8_t x4[rcx], uint8_t x5[r8], uint8_t x6[r9],  
uint8_t x7[rbp+0x10], uint8_t x8[rbp+0x18], uint8_t x9[rbp+0x20])
```

Veamos la llamada desde main:

```
main:  
push    rbp  
mov     rbp, rsp  
sub     rsp, 0x8  
push    0x9  
push    0x8  
push    0x7  
mov     r9d, 0x6  
mov     r8d, 0x5  
mov     ecx, 0x4  
mov     edx, 0x3  
mov     esi, 0x2  
mov     edi, 0x1  
  
call    sum_9  
add     rsp, 0x20  
movsx   eax, al  
mov     esi, eax  
lea     rdi, [rip+0xde7]  
mov     eax, 0x0  
call    printf  
mov     eax, 0x0  
leave  
ret
```

```
uint32_t sum_9(uint8_t x1[rdi], uint8_t x2[rsi], uint8_t x3[rdx],  
uint8_t x4[rcx], uint8_t x5[r8], uint8_t x6[r9],  
uint8_t x7[rbp+0x10], uint8_t x8[rbp+0x18], uint8_t x9[rbp+0x20])
```

```
main:  
push    rbp  
mov     rbp, rsp
```

La primera parte carga el valor RBP de la función llamadora y asigna el RBP de la función llamada al valor de tope de la pila así como estaba la ingresar a la función.

```
uint32_t sum_9(uint8_t x1[rdi], uint8_t x2[rsi], uint8_t x3[rdx],  
uint8_t x4[rcx], uint8_t x5[r8], uint8_t x6[r9],  
uint8_t x7[rbp+0x10], uint8_t x8[rbp+0x18], uint8_t x9[rbp+0x20])
```

```
main:  
push    rbp  
mov     rbp, rsp
```

Esto es necesario para poder preservar el valor de RBP anterior y utilizar el RBP actual para hacer referencia a los valores temporales y a los parámetros pasados por pila.



```
uint32_t sum_9(uint8_t x1[rdi], uint8_t x2[rsi], uint8_t x3[rdx],  
uint8_t x4[rcx], uint8_t x5[r8], uint8_t x6[r9],  
uint8_t x7[rbp+0x10], uint8_t x8[rbp+0x18], uint8_t x9[rbp+0x20])
```

```
main: ; estaba alineada a 8  
push    rbp ; queda en 16  
mov     rbp, rsp  
sub     rsp, 0x8 ; vuelve a 8  
push    0x9  
push    0x8  
push    0x7  
...
```

Luego se desplaza el tope  
de pila para dejarla  
alineada a 8 bytes.  
¿Por qué?

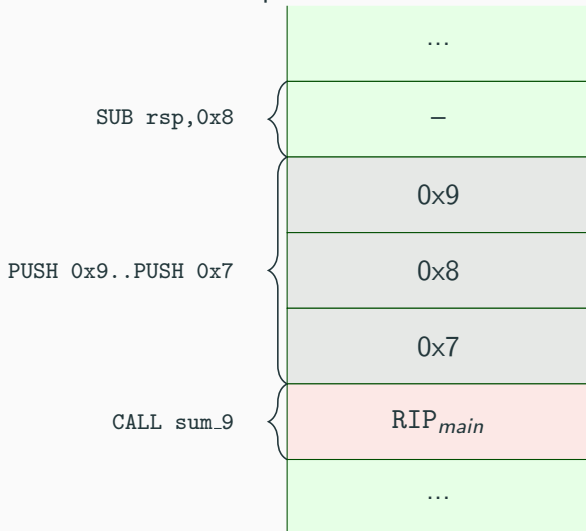
```
uint32_t sum_9(uint8_t x1[rdi], uint8_t x2[rsi], uint8_t x3[rdx],  
uint8_t x4[rcx], uint8_t x5[r8], uint8_t x6[r9],  
uint8_t x7[rbp+0x10], uint8_t x8[rbp+0x18], uint8_t x9[rbp+0x20])
```

```
push    0x9  
push    0x8  
push    0x7  
mov     r9d, 0x6  
mov     r8d, 0x5  
mov     ecx, 0x4  
mov     edx, 0x3  
mov     esi, 0x2  
mov     edi, 0x1  
call    sum_9
```

A continuación se pushean de derecha a izquierda los valores de los parámetros que no caben en los registros, y se asignan aquellos que sí. En la instrucción siguiente realizamos el llamado a la función.

Así recibe `sum_9` a la pila:

Así recibe `sum_9` a la pila:



```
uint32_t sum_9(uint8_t x1[rdi], uint8_t x2[rsi], uint8_t x3[rdx],  
uint8_t x4[rcx], uint8_t x5[r8], uint8_t x6[r9],  
uint8_t x7[rbp+0x10], uint8_t x8[rbp+0x18], uint8_t x9[rbp+0x20])
```

Veamos que pasa en `sum_9`:

```
sum_9:  
push    rbp  
mov     rsp, rsp  
sub     rsp, 0x40  
mov     eax, ecx  
mov     r11d, r8d  
mov     r10d, r9d  
mov     r9d, DWORD PTR [rbp+0x10]  
mov     r8d, DWORD PTR [rbp+0x18]  
mov     ecx, DWORD PTR [rbp+0x20]  
mov     BYTE PTR [rbp-0x14], dil
```

...

Sólo estudiaremos las primeras líneas. Aquí también preserva RBP, actualiza el valor de la base de la pila y luego desplaza el tope en 64 bytes reservando espacio para los valores temporales (`sub rsp, 0x40`).

```
uint32_t sum_9(uint8_t x1[rdi], uint8_t x2[rsi], uint8_t x3[rdx],  
uint8_t x4[rcx], uint8_t x5[r8], uint8_t x6[r9],  
uint8_t x7[rbp+0x10], uint8_t x8[rbp+0x18], uint8_t x9[rbp+0x20])
```

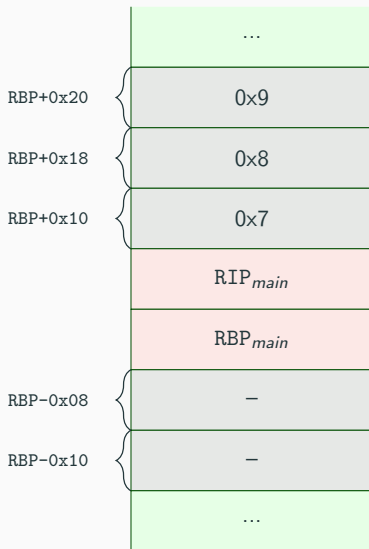
Veamos que pasa en `sum_9`:

```
sum_9:  
push    rbp  
mov     rbp, rsp  
sub     rsp, 0x40  
mov     eax, ecx  
mov     r11d, r8d  
mov     r10d, r9d  
mov     r9d, DWORD PTR [rbp+0x10]  
mov     r8d, DWORD PTR [rbp+0x18]  
mov     ecx, DWORD PTR [rbp+0x20]  
mov     BYTE PTR [rbp-0x14], dil  
...
```

Nota: siempre que vean que se suma un valor al RBP, por ejemplo `RBP + 0x10` deben entender que estamos accediendo a un valor que recibimos de la función llamadora, ya que se pushearon antes de actualizar RBP.

Así prepara `sum_9` la pila:

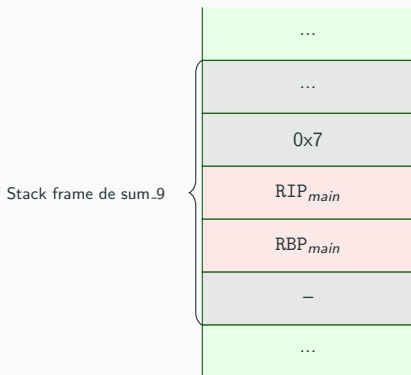
Así prepara `sum_9` la pila:





A la región de la pila comprendida entre los parámetros recibidos y el tope de pila actual le llamaremos stack frame. Durante la ejecución del programa, en nuestra pila suele haber varios stack frames apilados, uno por cada llamada a función de la cual no se regresó aún.

A la región de la pila comprendida entre los parámetros recibidos y el tope de pila actual le llamaremos stack frame. Durante la ejecución del programa, en nuestra pila suele haber varios stack frames apilados, uno por cada llamada a función de la cual no se regresó aún.



Es importante intentar comprender la estructura de la pila, antes, al ingresar y al salir de la función sobre la que estamos trabajando.

Es importante intentar comprender la estructura de la pila, antes, al ingresar y al salir de la función sobre la que estamos trabajando. Junto con los registros conforman los elementos utilizados para pasar información entre funciones, pero a diferencia de los registros se trata de una estructura dinámica y sobre la que hay que considerar las convenciones estructurales y de uso.

Cierre

---

Hoy vimos:

Hoy vimos:

- Estructura de un programa en assembly x86

Hoy vimos:

- Estructura de un programa en assembly x86
- Interpretación de convenciones en funciones como contratos.



Hoy vimos:

- Estructura de un programa en assembly x86
- Interpretación de convenciones en funciones como contratos.
- Uso de la pila y registros en llamadas a funciones.

Hoy vimos:

- Estructura de un programa en assembly x86
- Interpretación de convenciones en funciones como contratos.
- Uso de la pila y registros en llamadas a funciones.
- Definición de stack frame