Lenguaje de máquina: procedimientos (ABI)

Organización del computador - FIUBA

1.er cuatrimestre de 2024

Última modificación: Sun Oct 8 20:51:04 2023 -0300

Créditos

Para armar las presentaciones del curso utilizamos:

R. E. Bryant and D. R. O'Hallaron, *Computer systems: a programmer's perspective*, Third edition, Global edition. Boston Columbus Hoboken Indianapolis New York San Francisco Cape Town: Pearson, 2015.

D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, RISC-V edition. Cambridge, Massachusetts: Morgan Kaufmann Publishers, an imprint of Elsevier, 2017.

J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. 2017.

El contenido de los slides está basado en las presentaciones de Patricio Moreno y de Organización del Computador I - FCEN.

Tabla de contenidos

Tabla de contenidos

Tabla de contenidos

Soporte para procedimientos

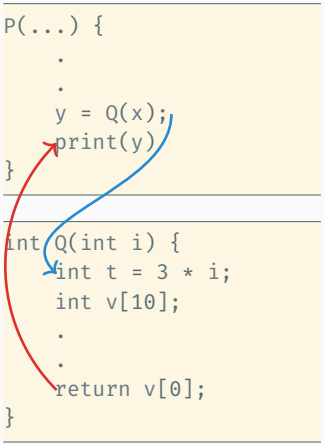
- Transferencia del control
  - Necesario para “saltar” al código del procedimiento
  - Retorno al punto del salto
- Pasaje de datos
  - Argumentos
  - Valor de retorno
- Manejo de la memoria
  - Reservar lo necesario en el procedimiento
  - Liberar lo pedido al retornar
- Todos los mecanimos se implementan a través de instrucciones
- Todo procedimiento en x86-64 utiliza esos mecanismos

```
P(...) {
    .
    .
    y = Q(x);
    print(y)
}
```

```
int Q(int i) {
    int t = 3 * i;
    int v[10];
    .
    .
    return v[0];
}
```

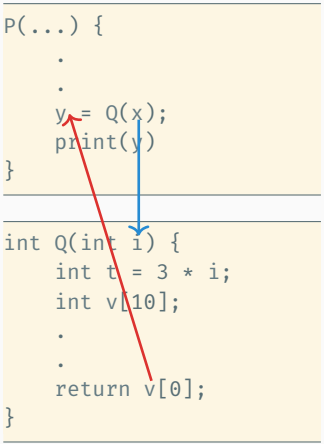
Soporte para procedimientos

- Transferencia del control
  - Necesario para “saltar” al código del procedimiento
  - Retorno al punto del salto
- Pasaje de datos
  - Argumentos
  - Valor de retorno
- Manejo de la memoria
  - Reservar lo necesario en el procedimiento
  - Liberar lo pedido al retornar
- Todos los mecanimos se implementan a través de instrucciones
- Todo procedimiento en x86-64 utiliza esos mecanismos



Soporte para procedimientos

- Transferencia del control
  - Necesario para “saltar” al código del procedimiento
  - Retorno al punto del salto
- Pasaje de datos
  - Argumentos
  - Valor de retorno
- Manejo de la memoria
  - Reservar lo necesario en el procedimiento
  - Liberar lo pedido al retornar
- Todos los mecanimos se implementan a través de instrucciones
- Todo procedimiento en x86-64 utiliza esos mecanismos



Soporte para procedimientos

- Transferencia del control
  - Necesario para “saltar” al código del procedimiento
  - Retorno al punto del salto
- Pasaje de datos
  - Argumentos
  - Valor de retorno
- Manejo de la memoria
  - Reservar lo necesario en el procedimiento
  - Liberar lo pedido al retornar
- Todos los mecanimos se implementan a través de instrucciones
- Todo procedimiento en x86-64 utiliza esos mecanismos

```
P(...) {
    .
    .
    y = Q(x);
    print(y)
}
```

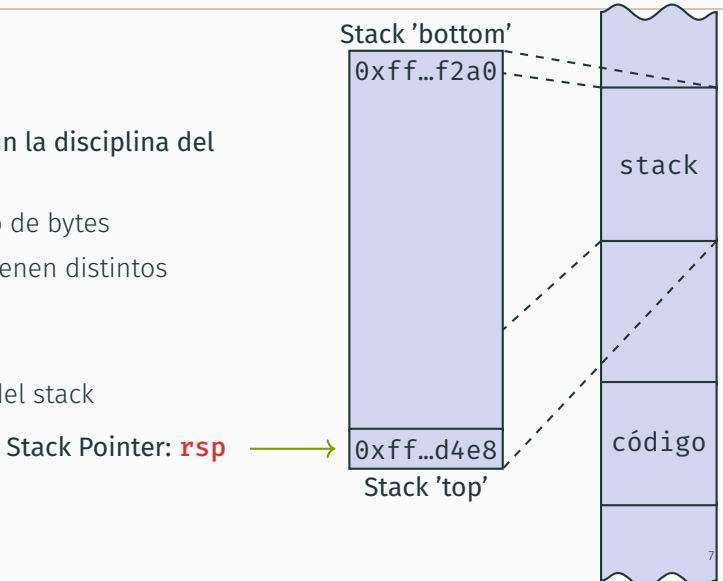
```
int Q(int i) {
    int t = 3 * i;
    int v[10];
    .
    .
    return v[0];
}
```

Tabla de contenidos

Pila (*Stack*) x86-64

Región de memoria administrada según la disciplina del stack

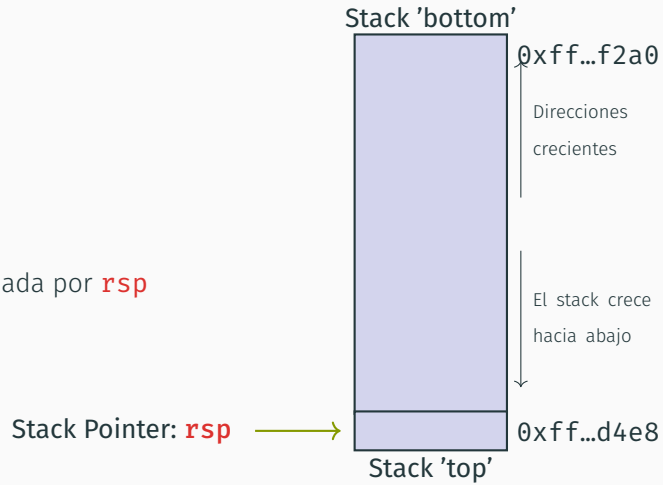
- La memoria se ve como un arreglo de bytes
- Diferentes regiones de la misma tienen distintos propósitos
- Crece hacia direcciones menores
- `rsp` contiene la menor dirección del stack



Stack x86-64: push

`pushq origen`

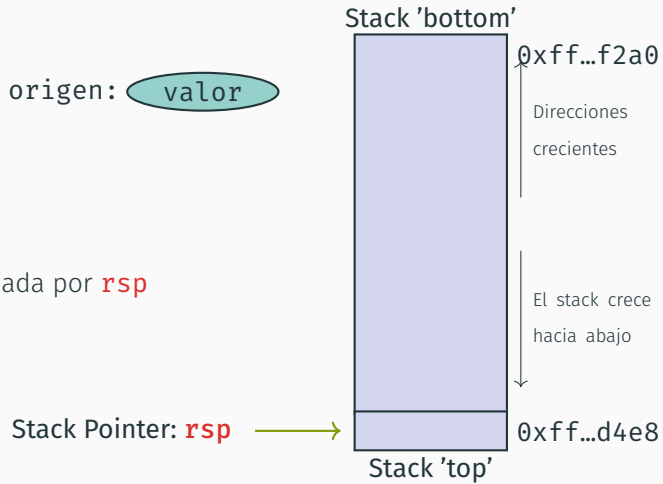
- obtener el dato de **`origen`**
- decrementar `rsp` en 8
- guardar el dato en la dirección dada por `rsp`



Stack x86-64: push

`pushq origen`

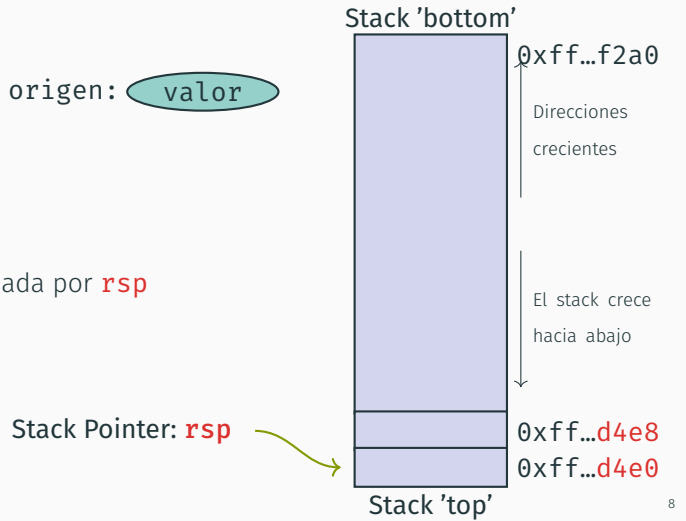
- obtener el dato de `origen`
- decrementar `rsp` en 8
- guardar el dato en la dirección dada por `rsp`



Stack x86-64: push

`pushq origen`

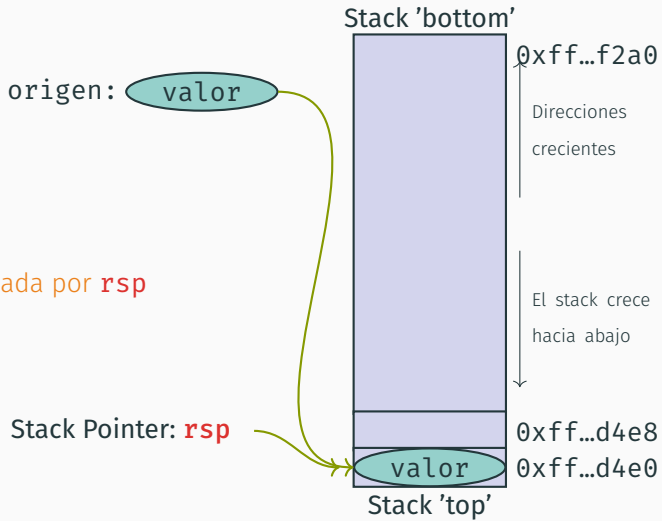
- obtener el dato de **`origen`**
- decrementar **`rsp`** en 8
- guardar el dato en la dirección dada por **`rsp`**



Stack x86-64: push

`pushq origen`

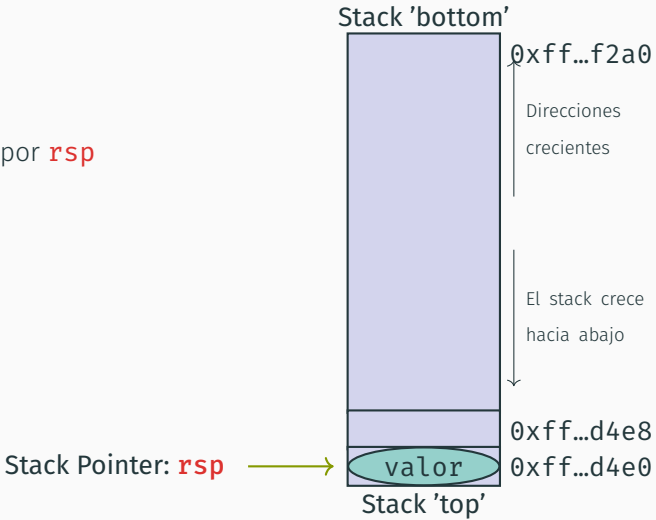
- obtener el dato de **`origen`**
- decrementar `rsp` en 8
- guardar el dato en la dirección dada por `rsp`



Stack x86-64: pop

`pop destino`

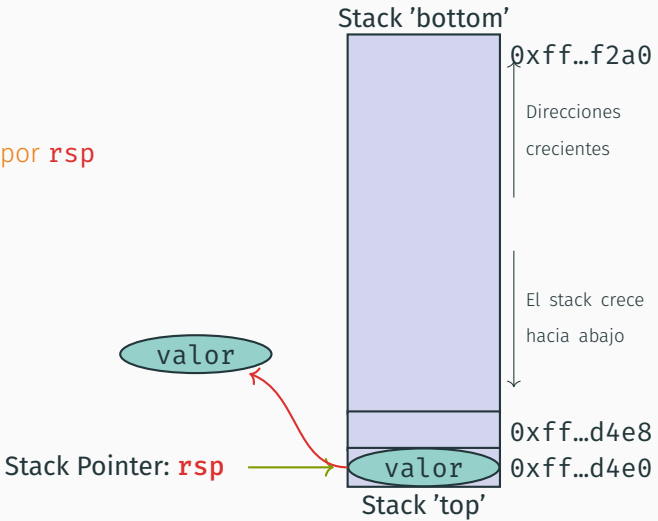
- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`



Stack x86-64: pop

`pop destino`

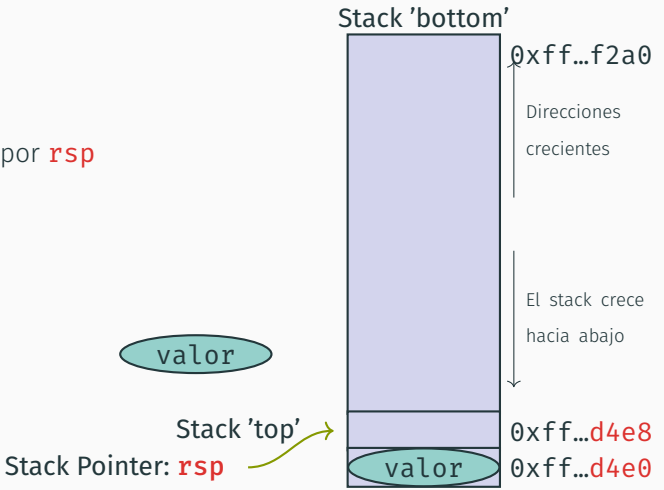
- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`



Stack x86-64: pop

`pop destino`

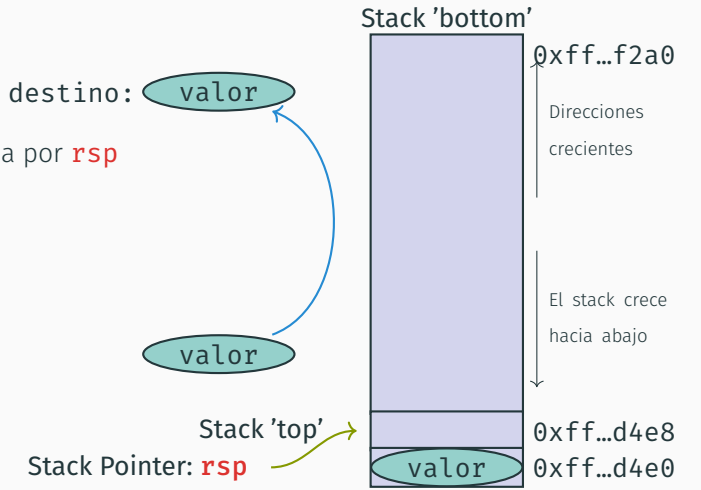
- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`



Stack x86-64: pop

`pop destino`

- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`



Stack x86-64: pop

`pop destino`

- leer el dato en la dirección dada por `rsp`
- incrementar `rsp` en 8
- guardar el dato en `destino`
- No cambia lo almacenado en memoria, sólo el valor de `rsp`

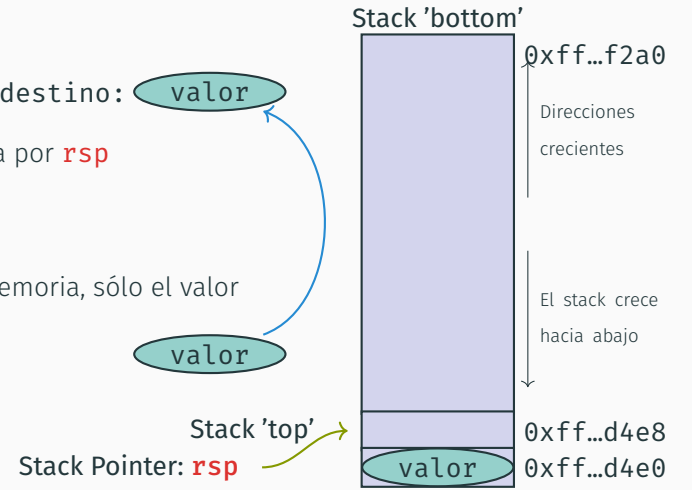


Tabla de contenidos

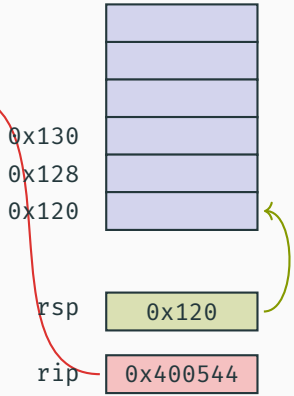
Transferencia de control

- Usa el *stack* para dar soporte a las llamadas y retornos de procedimientos
- Llamada a procedimientos/funciones: `call etiqueta`
  - Hacer un `push` de la *dirección de retorno*
  - "Saltar" a la `etiqueta`
- *Dirección de retorno*
  - Dirección de la instrucción siguiente (inmediata) a la instrucción `call`
- Retorno de procedimientos/funciones: `ret`
  - Hacer un `pop` de la dirección de retorno
  - "Saltar" a dicha dirección

Transferencia de control: ejemplo

```
0000000000400540 <multstore>:
  …
  …
  400544: call 400550 <mult2>
  400549: mov   [rbx], rax
  …
  …
```

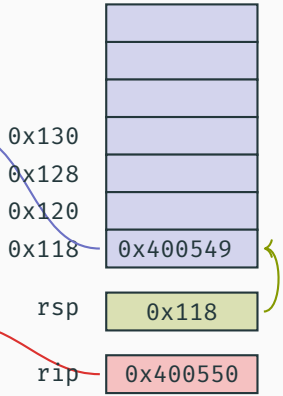
```
0000000000400550 <mult2>:
  400550: mov   rax, rdi
  …
  400557: ret
```



Transferencia de control: ejemplo

```
0000000000400540 <multstore>:
  …
  …
  400544: call 400550 <mult2>
  400549: mov   [rbx], rax
  …
  …
```

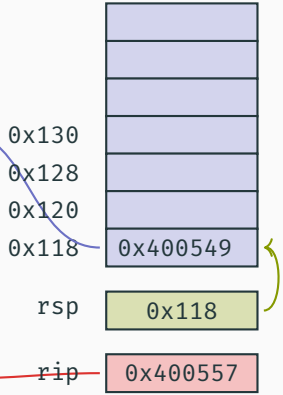
```
0000000000400550 <mult2>:
  400550: mov   rax, rdi
  …
  400557: ret
```



Transferencia de control: ejemplo

```
0000000000400540 <multstore>:
  …
  …
  400544: call 400550 <mult2>
  400549: mov   [rbx], rax
  …
  …
```

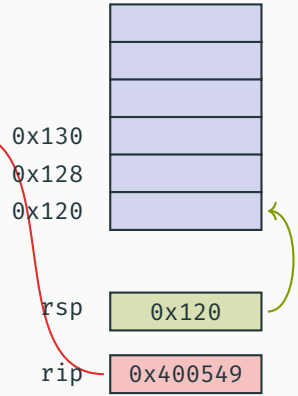
```
0000000000400550 <mult2>:
  400550: mov   rax, rdi
  …
  400557: ret
```



Transferencia de control: ejemplo

```
0000000000400540 <multstore>:
  …
  …
  400544: call 400550 <mult2>
  400549: mov   [rbx], rax
  …
  …
```

```
0000000000400550 <mult2>:
  400550: mov   rax, rdi
  …
  400557: ret
```



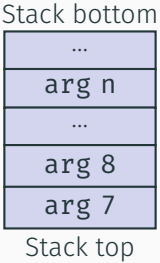
Pasaje de datos

- Los argumentos se pasan por registros o usando el stack
  - en x86 (32 bits) únicamente usando la pila
- **Primeros 6 argumentos**

rdi	arg 1
rsi	arg 2
rdx	arg 3
rsx	arg 4
r8	arg 5
r9	arg 6

- **Valor de retorno**

rax



- **Argumentos siguientes**
- El espacio para los argumentos se reserva únicamente si es necesario

Pasaje de datos: ejemplo

```
void multstore
(long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
# x en rdi, y en rsi, dest en rdx
    ...
    400541: mov     rbx, rdx # guarda dest
    400544: call 400550 <mult2>
    # t en rax
    400549: mov    [rbx], rax
```

```
long mult2
(long a, long b)
{
    long s = a * b;
    return s;
}
```

```
0000000000400550 <mult2>:
    # a en rdi, b en rsi
    400550: mov     rax, rdi
    400553: imul    rax, rsi
    # s en rax
    400557: ret
```

Lenguajes basados en pilas

- Lenguajes que soportan recursividad
  - El código debe ser reentrante (*reentrant*)
    - Para soportar instanciaciones múltiples de un mismo procedimiento
  - Requiere de espacio para almacenar el estado de cada instancia
    - Argumentos
    - Variables locales
    - Retorno
- Disciplina del stack
  - Necesita el estado de un procedimiento durante un tiempo finito
    - Desde que se lo llama hasta que termina
  - El proceso invocado finaliza **antes** que el invocante
- El stack se reserva de a **frames**
  - Guarda el estado de una única instancia de un procedimiento

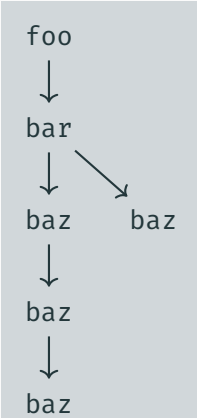
Ejemplo de cadena de invocaciones

```
foo (…)
{
  …
  …
bar (…);
  …
  …
}
```

```
bar (…)
{
  …
baz (…);
  …
baz (…);
  …
}
```

```
baz (…)
{
  …
  …
baz (…);
  …
  …
}
```

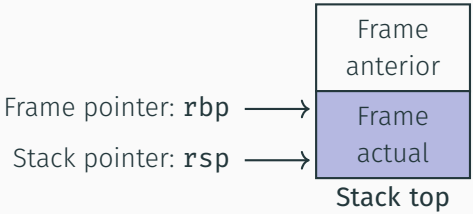
Árbol de llamadas



`baz()` es recursivo

Stack frames

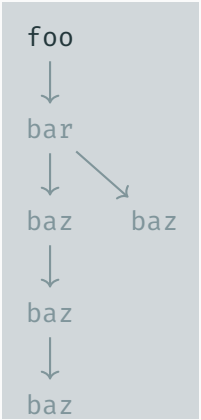
- **Contiene:**
  - Información de retorno
  - Almacenamiento local
  - Espacio temporal

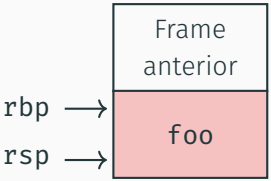


- **Administración:**
  - El espacio se reserva al entrar
    - Requiere código de inicialización
    - Incluye el `push` de la instrucción `call`
  - El espacio se retorna al salir
    - Requiere código de finalización
    - Incluye el `pop` de la instrucción `ret`

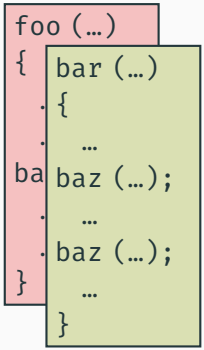
Stack frames: ejemplo

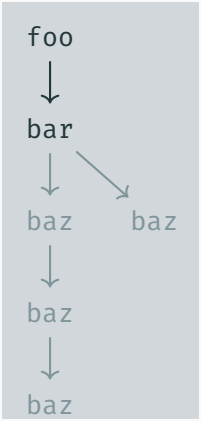
```
foo (…)
{
  …
  …
bar (…);
  …
  …
}
```

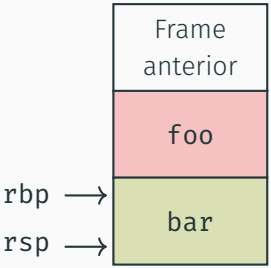




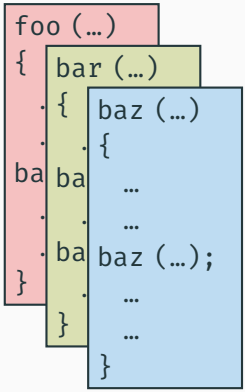
Stack frames: ejemplo

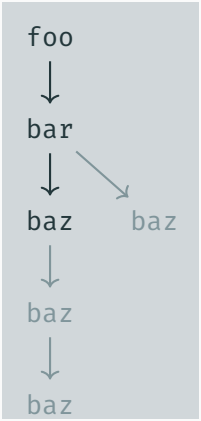


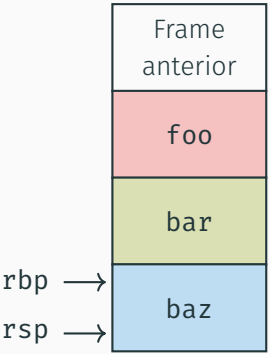




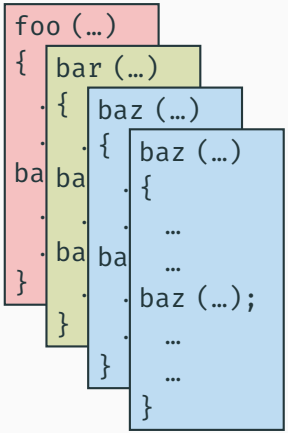
Stack frames: ejemplo

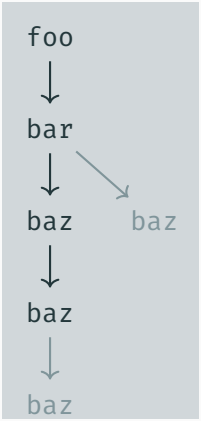


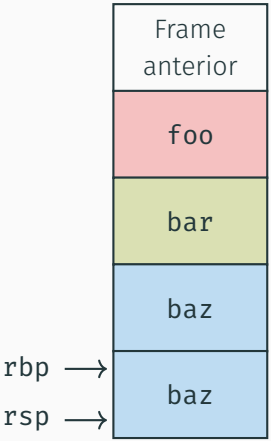




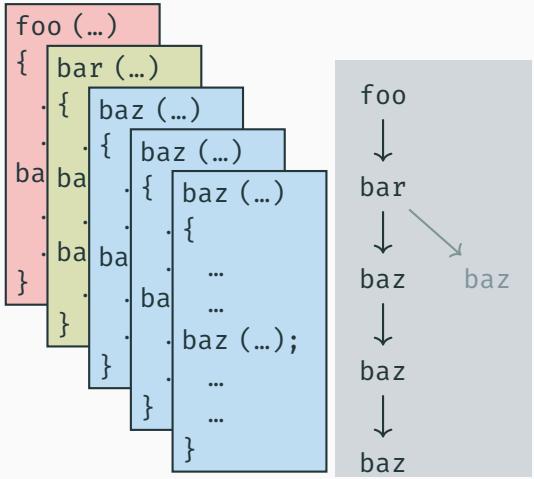
Stack frames: ejemplo



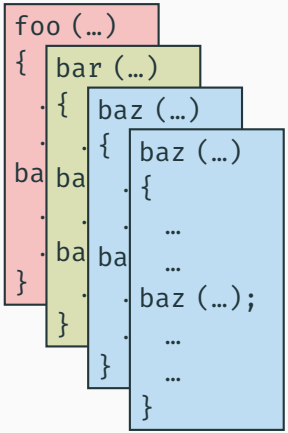


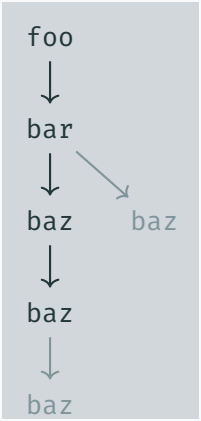


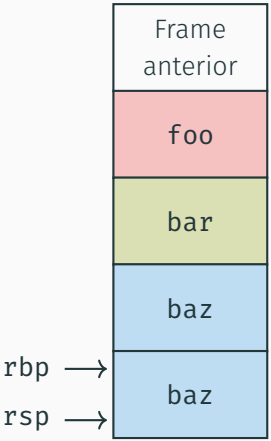
Stack frames: ejemplo



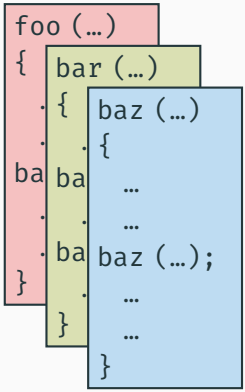
Stack frames: ejemplo

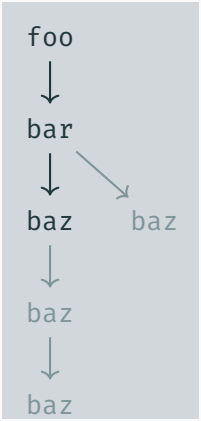


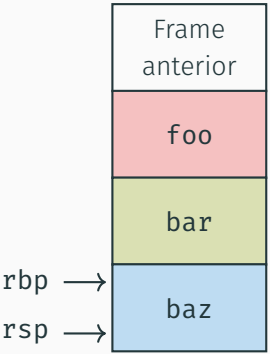




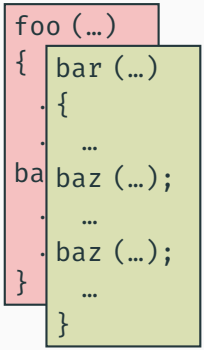
Stack frames: ejemplo

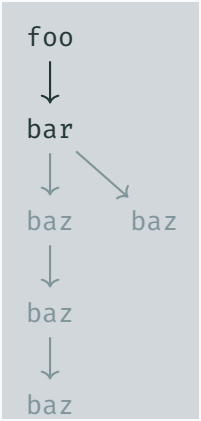


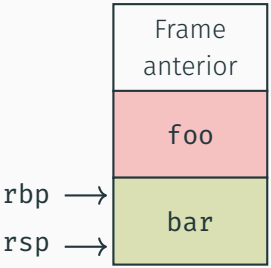




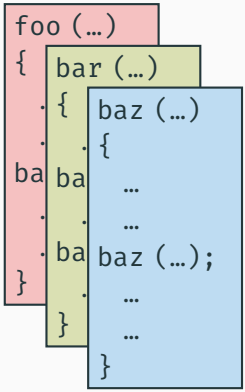
Stack frames: ejemplo

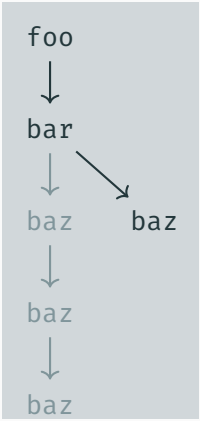


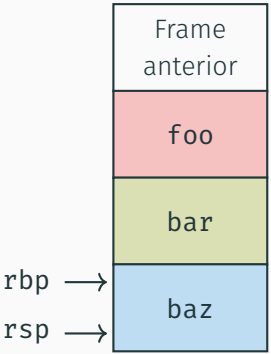




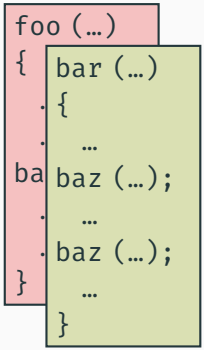
Stack frames: ejemplo

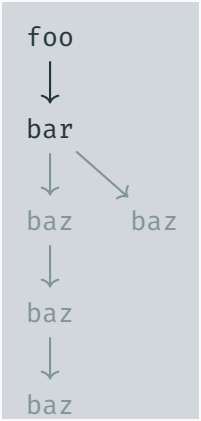


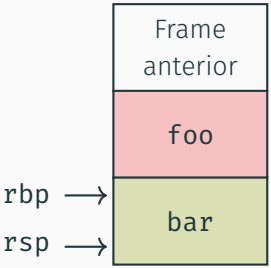




Stack frames: ejemplo

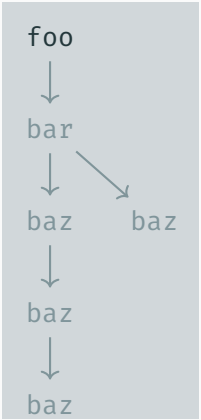


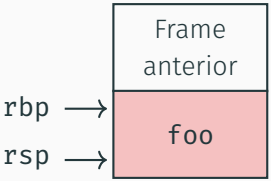




Stack frames: ejemplo

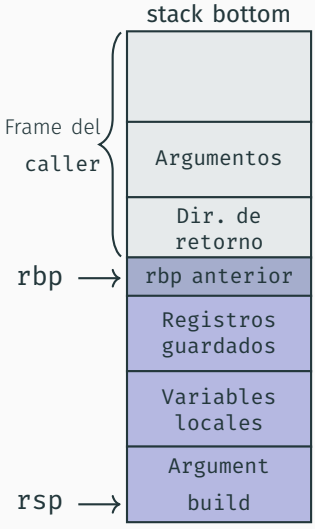
```
foo (…)
{
  …
  …
bar (…);
  …
  …
}
```





Linux Stack Frame

- Frame actual (top a bottom)
  - *argument build*: parámetros de una función a ser invocada
  - variables locales (si no alcanzan los registros)
  - Registros guardados
  - *frame pointer* anterior
- Frame de la función invocante
  - dirección de retorno
    - **push**eada por `call`
  - argumentos para esta función
    - En x86_64 (64 bits): del séptimo en adelante
    - En x86 (32 bits): todos



Convenciones para registros

- Cuando `foo` llama a `bar`:
  - **`foo`** se llama **caller** o invocante
  - **`bar`** se llama **callee** o invocada
- ¿Qué ocurre con los registros usados como temporales?

```
foo:
    ...
    mov    rdx, 0xb00710ad
    call   bar
    add    rax rdx
    ...
    ret
```

```
bar:
    ...
    sub    rdx, 0xdeadbeef
    ...
    ret
```

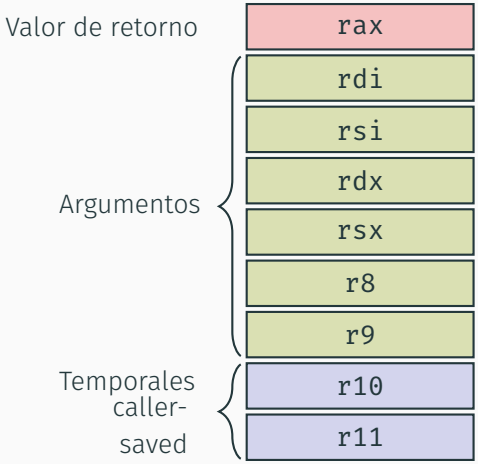
  - El contenido de `rdx` es sobreescrito por `bar`
  - Es necesario algún arreglo de partes para que funcione correctamente

Convenciones para registros

- Cuando `foo` llama a `bar`:
  - **`foo`** se llama **caller** o invocante
  - **`bar`** se llama **callee** o invocada
- ¿Qué ocurre con los registros usados como temporales?
  - El contenido de éstos puede ser sobreescrito por la función callee
  - Es necesario algún arreglo de partes para que funcione correctamente
- Convenciones (*calling conventions*):
  - Caller saved
    - la función *caller* guarda los registros en el stack antes de la invocación
  - Callee saved
    - la función *callee* guarda los registros en el stack antes de **modificarlos**
    - la función *callee* reestable los valores de los registros modificados antes de retornar

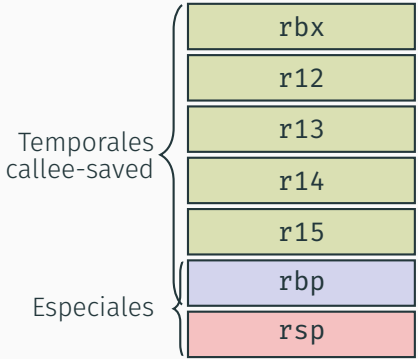
Convenciones para registros caller-saved en x86_64

- **rax**
  - valor de retorno
  - caller-saved
  - un procedimiento puede modificarlo
- **rdi, ..., r9**
  - argumentos
  - caller-saved
  - un procedimiento puede modificarlos
- **r10, r11**
  - caller-saved
  - un procedimiento puede modificarlos

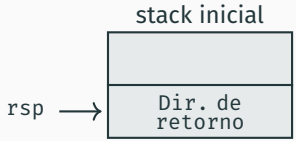


Convenciones para registros callee-saved en x86_64

- **rbx, r12, r13, r14, r15**
  - callee-saved
  - la función callee debe guardarlos y restaurarlos
- **rbp**
  - callee-saved
  - la función callee debe guardarlos y restaurarlos
  - opcionalmente puede usarse como frame pointer
- **rsp**
  - callee-saved especial
  - se restaura a su valor original al retornar del procedimiento



Ejemplo de *calling conventions*



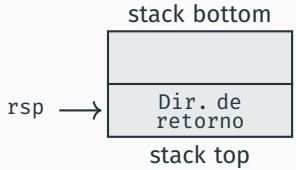

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

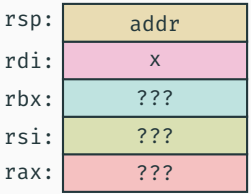
- `x` se pasa en `rdi`
- `rdi` se necesita para llamar a `fun()`
- `x` (`rdi`) se necesita *después* de llamar a `fun()`, *che facciamo?*

Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

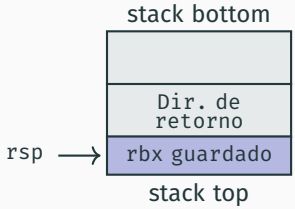




Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

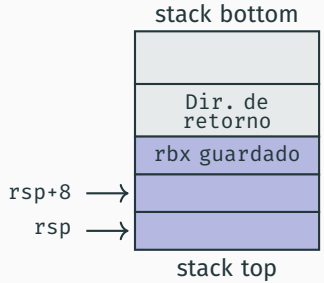
```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```



Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

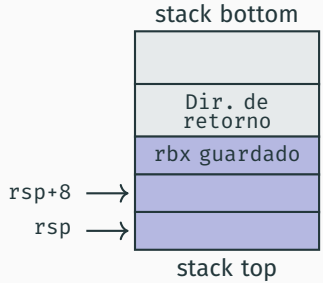
```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

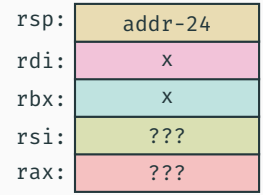


Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

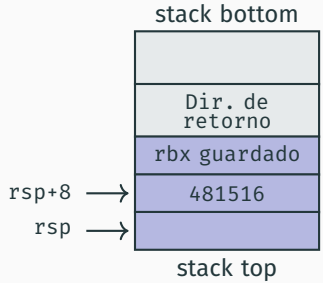


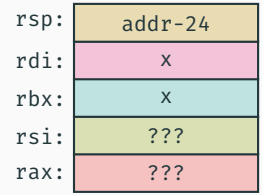


Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

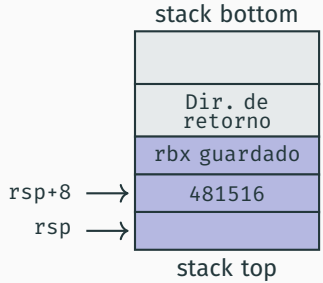


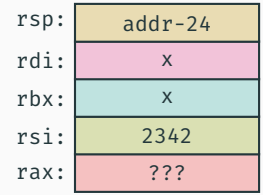


Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

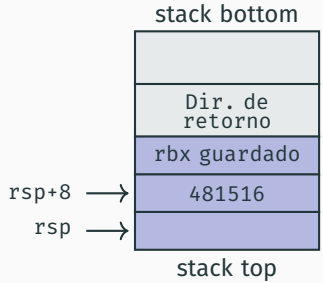


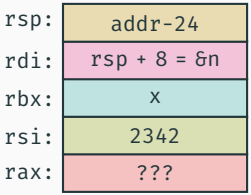


Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

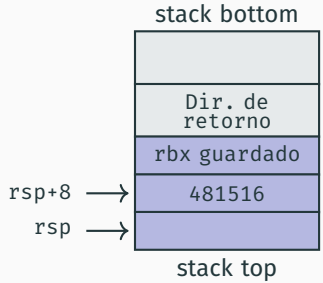


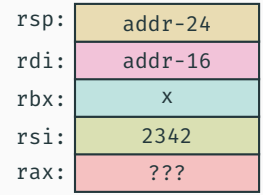


Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

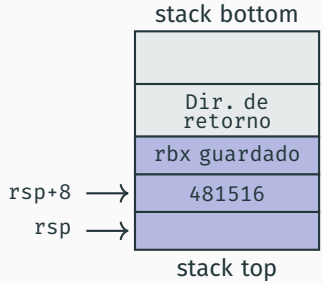


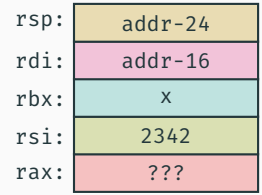


Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```





Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

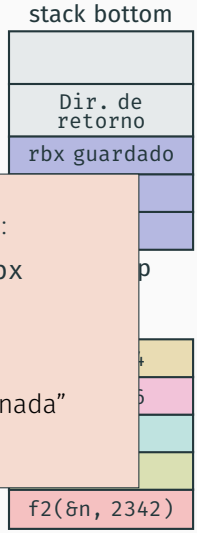
Se ejecuta la llamada a `f2()`:

- `x` está almacenado en `rbx`

Al retornar:

- `y` está en `rax`

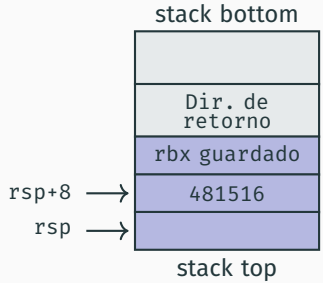
Por cómo funciona el stack, “nada” más cambió.

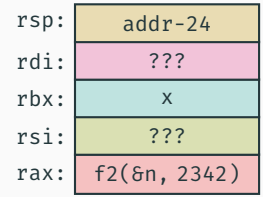


Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

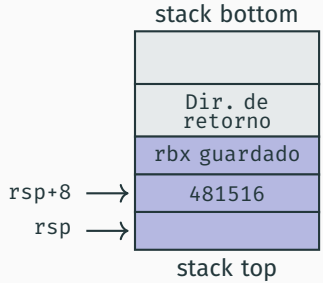


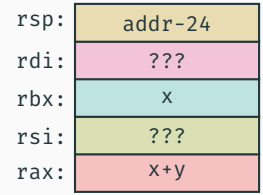


Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```

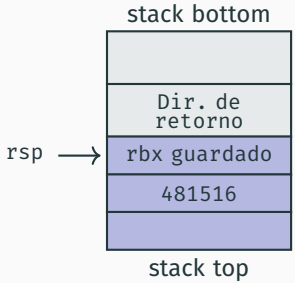




Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

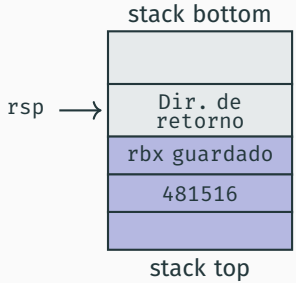
```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```



Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

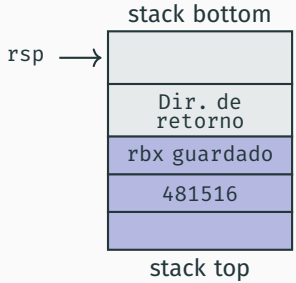
```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```



Ejemplo de *calling conventions*

```
long f1(long x) {
    long n = 481516;
    long y = f2(&n, 2342)
    return x + y
}
```

```
f1:
  push rbx
  sub rsp, 16
  mov rbx, rdi
  mov [rsp +8], 481516
  mov esi, 2342
  lea rdi, [rsp +8]
  call f2
  add rax, rbx
  add rsp, 16
  pop rbx
  ret
```



Licencia del estilo de beamer

Obtén el código de este estilo y la presentación demo en

`github.com/pamoreno/mtheme`

El estilo *en sí* está licenciado bajo la Creative Commons Attribution-ShareAlike 4.0 International License. El estilo es una modificación del creado por Matthias Vogelgesang, disponible en

`github.com/matze/mtheme`