Argumentos

Tipos enteros (incluye punteros), en registros, en este orden (un argumento por registro):

rdi, rsi, rdx, rcx, r8 y r9

¿más argumentos? En el stack, enviados a la pila de derecha a izquierda.

Retorno: rax

Ejemplo de pasaje de argumentos

- demo_pasaje_args_1.c
- demo_pasaje_args_2.c (y demo_pasaje_args_2.s)

¡Los argumentos se evalúan de derecha a izquierda!

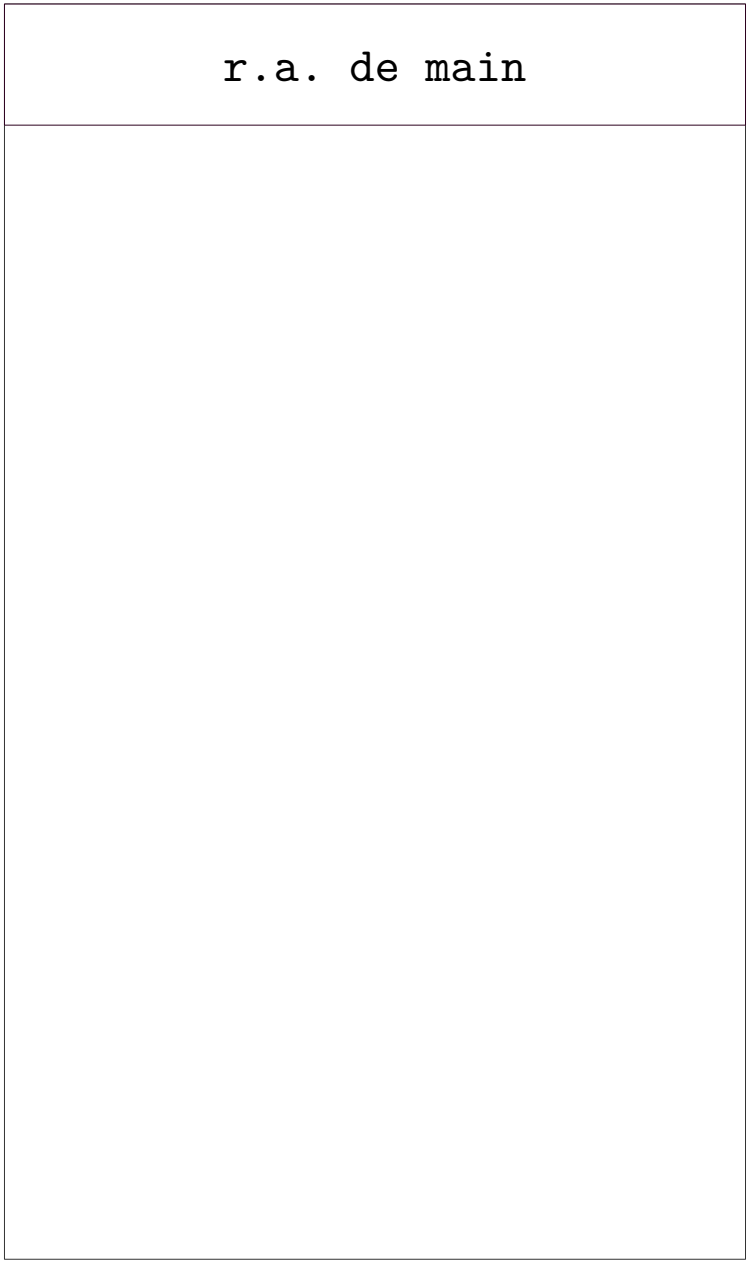
Evolución del stack

- demo_pasaje_args3.s Implementando:

```
f(long a1, long a2, long a3, long a4, long a5,
  long a6, long a7, long a8) {
        return a8;
}
main() {
        return f(1,2,3,4,5,6,7,8);
}
```

Evolución del stack

```
f:
movq 16(%rsp), %rax

ret

.global main
main:
#Cargar args al stack

cargar_pila:    # O (lo que es lo mismo)
pushq $8        #    subq    $16, %rsp
pushq $7        #    movq    $8, 8(%rsp)
                #    movq    $7, (%rsp)
cargar_regs:
movl$6, %r9d; movl $5, %r8d
movl$4, %ecx; movl $3, %edx
movl$2, %esi; movl $1, %edi
callf: call  f

deshace_pila:  # O (lo que es casi lo mismo)
addq$16, %rsp  # popq %rcx
retorna:       # popq %rcx
ret
```



Evolución del stack

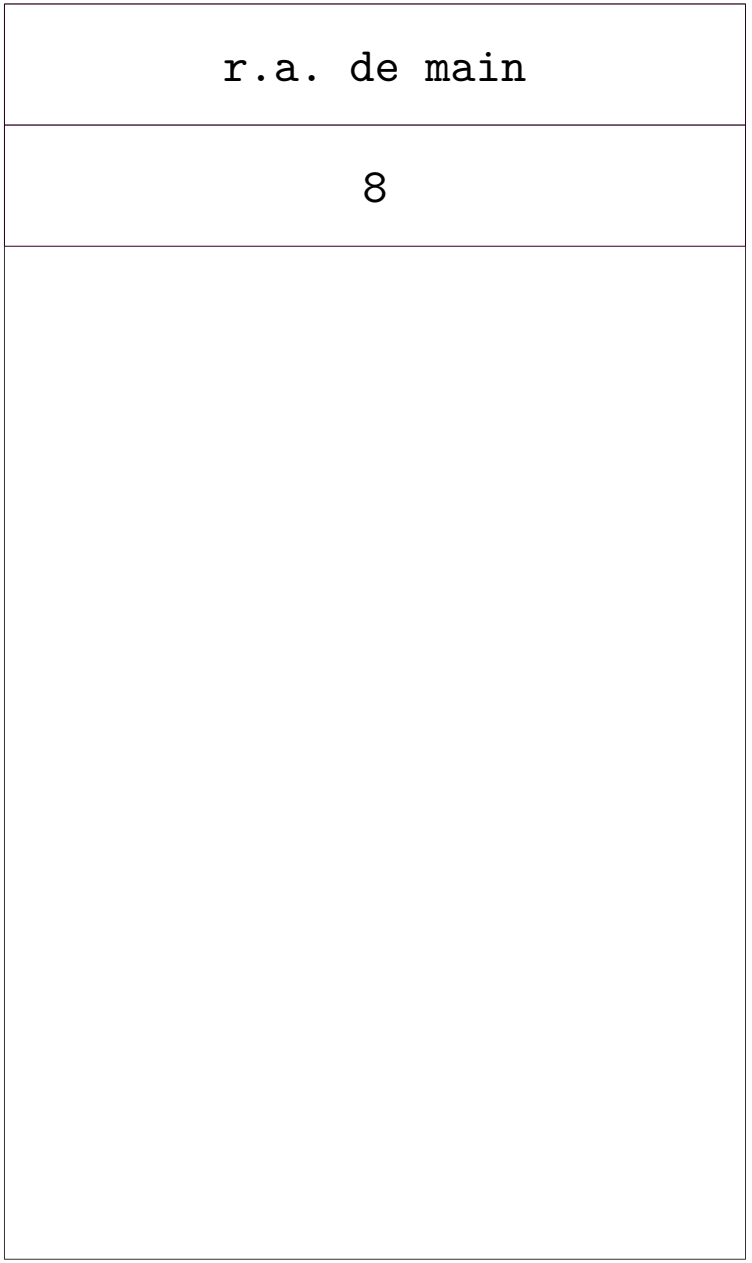
```
f:
movq 16(%rsp), %rax

ret

.global main
main:
#Cargar args al stack

cargar_pila:
pushq  $8
pushq  $7

cargar_regs:
movl$6, %r9d; movl $5, %r8d
movl$4, %ecx; movl $3, %edx
movl$2, %esi; movl $1, %edi
callf: call  f

deshace_pila:
addq$16, %rsp
retorna:
ret
```

```
# O (lo que es lo mismo)
#    subq     $16, %rsp
#    movq     $8, 8(%rsp)
#    movq     $7, (%rsp)
```

```
# O (lo que es casi lo mismo)
# popq %rcx
# popq %rcx
```

(%rsp) ->

Stack



Evolución del stack

```
f:
movq 16(%rsp), %rax
return a8
ret

.global main
main:
#Cargar args al stack

cargar_pila:
pushq  $8
pushq  $7

cargar_regs:
movl$6, %r9d; movl $5, %r8d
movl$4, %ecx; movl $3, %edx
movl$2, %esi; movl $1, %edi
callf: call  f

deshace_pila:
addq$16, %rsp
retorna:
ret
```

```
# O (lo que es lo mismo)
#    subq     $16, %rsp
#    movq     $8, 8(%rsp)
#    movq     $7, (%rsp)
```

```
# O (lo que es casi lo mismo)
# popq %rcx
# popq %rcx
```

(%rsp) ->

Stack

r.a. de main
8
7

Evolución del stack

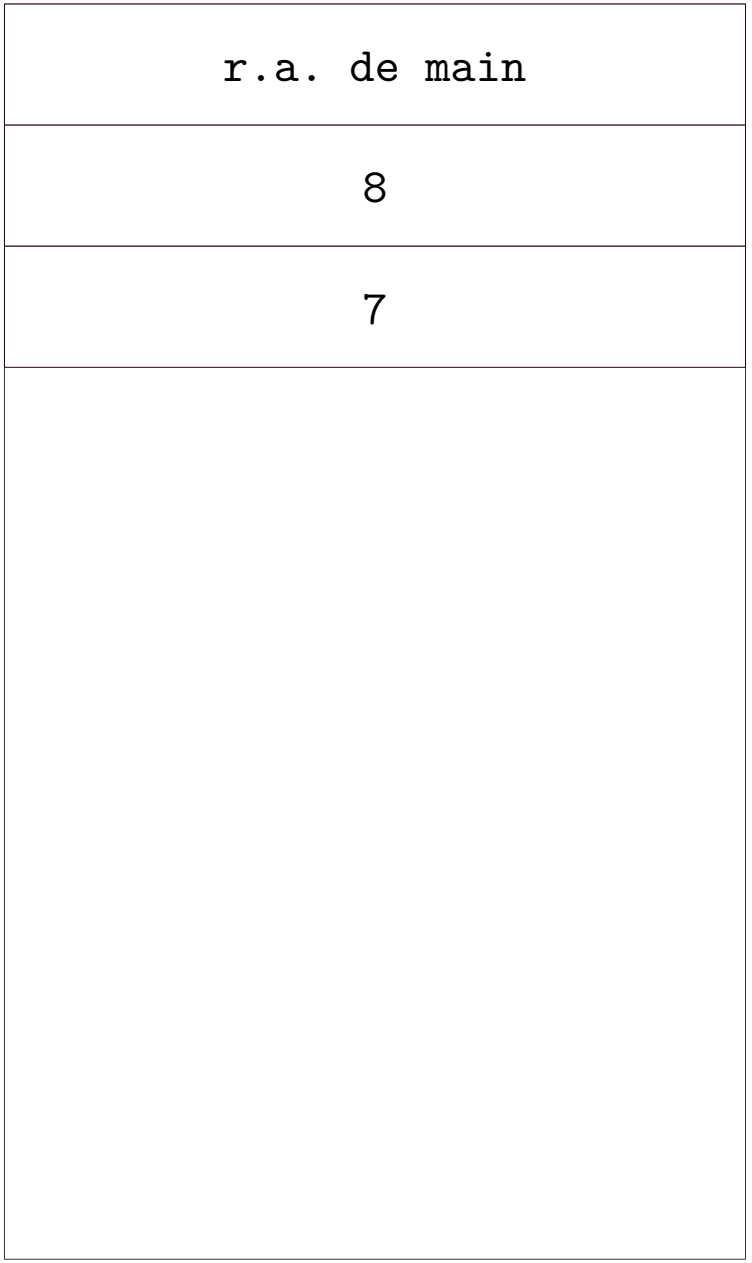
```
f:
movq 16(%rsp), %rax

ret

.global main
main:
#Cargar args al stack

cargar_pila:
pushq  $8
pushq  $7

cargar_regs:
movl$6, %r9d; movl $5, %r8d
movl$4, %ecx; movl $3, %edx
movl$2, %esi; movl $1, %edi
callf: call  f

deshace_pila:
addq$16, %rsp
retorna:
ret
```

```
# O (lo que es lo mismo)
#    subq     $16, %rsp
#    movq     $8, 8(%rsp)
#    movq     $7, (%rsp)
```

```
# O (lo que es casi lo mismo)
# popq %rcx
# popq %rcx
```

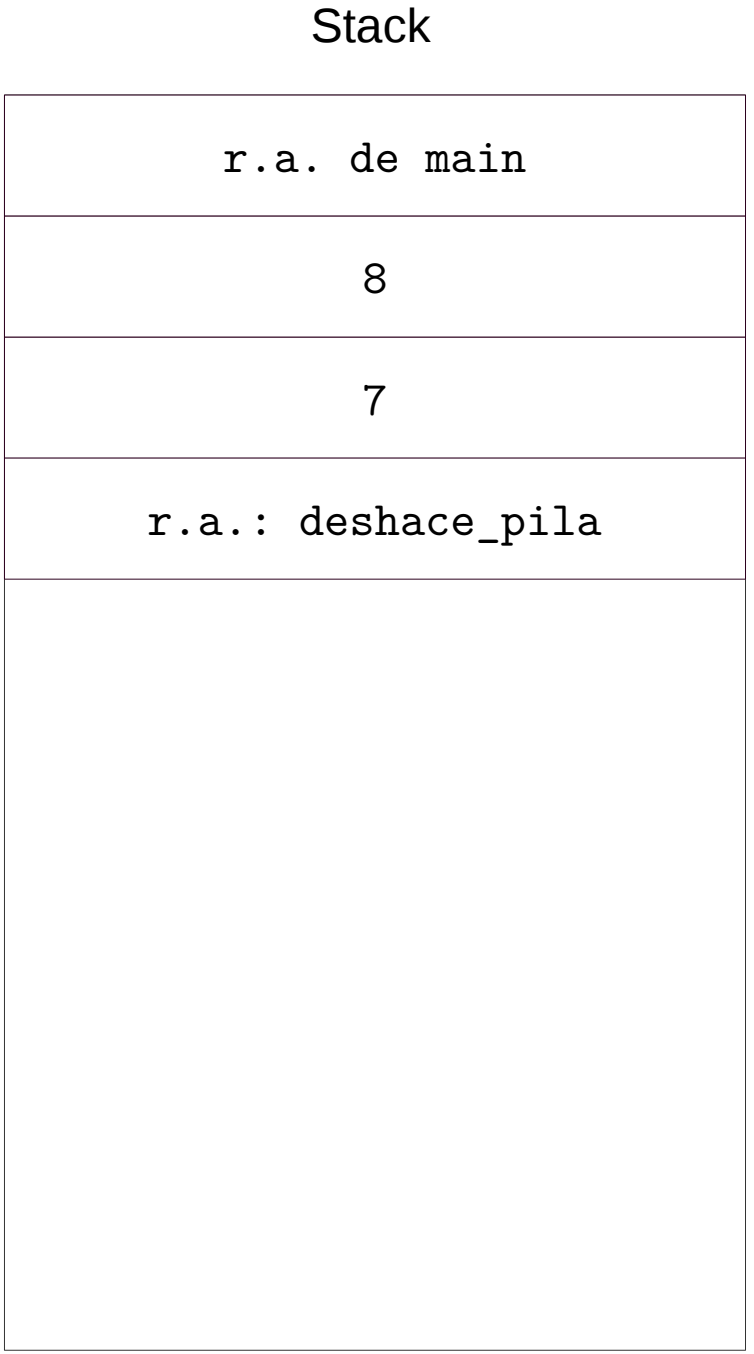
(%rsp) ->

Stack



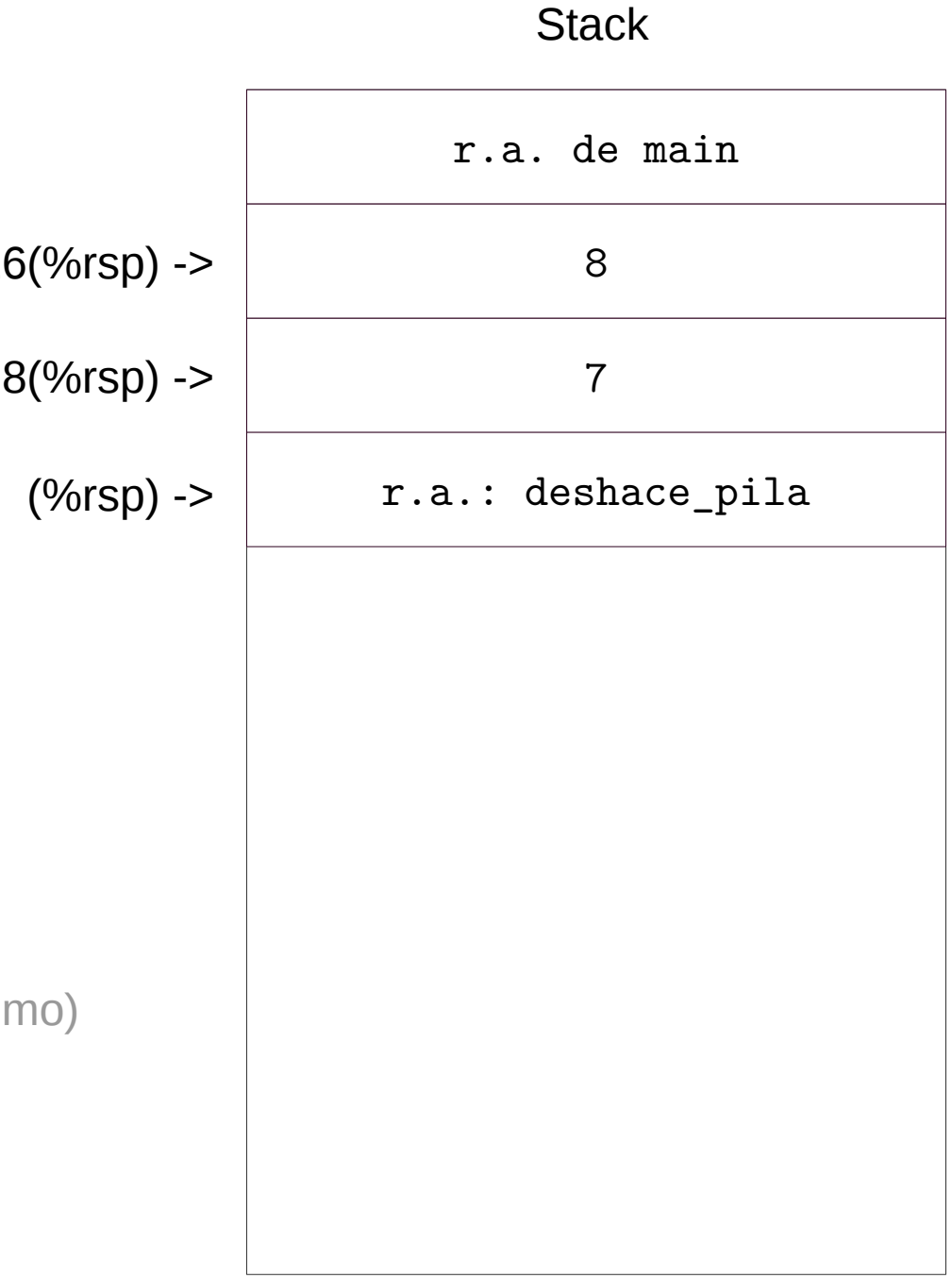
Evolución del stack

```
f:
movq 16(%rsp), %rax

ret

.global main
main:
#Cargar args al stack

cargar_pila:    # O (lo que es lo mismo)
pushq $8        #    subq    $16, %rsp
pushq $7        #    movq    $8, 8(%rsp)
                #    movq    $7, (%rsp)
cargar_regs:
movl$6, %r9d; movl $5, %r8d
movl$4, %ecx; movl $3, %edx
movl$2, %esi; movl $1, %edi
callf: call  f

deshace_pila:  # O (lo que es casi lo mismo)
addq$16, %rsp  # popq %rcx
retorna:       # popq %rcx
ret
```



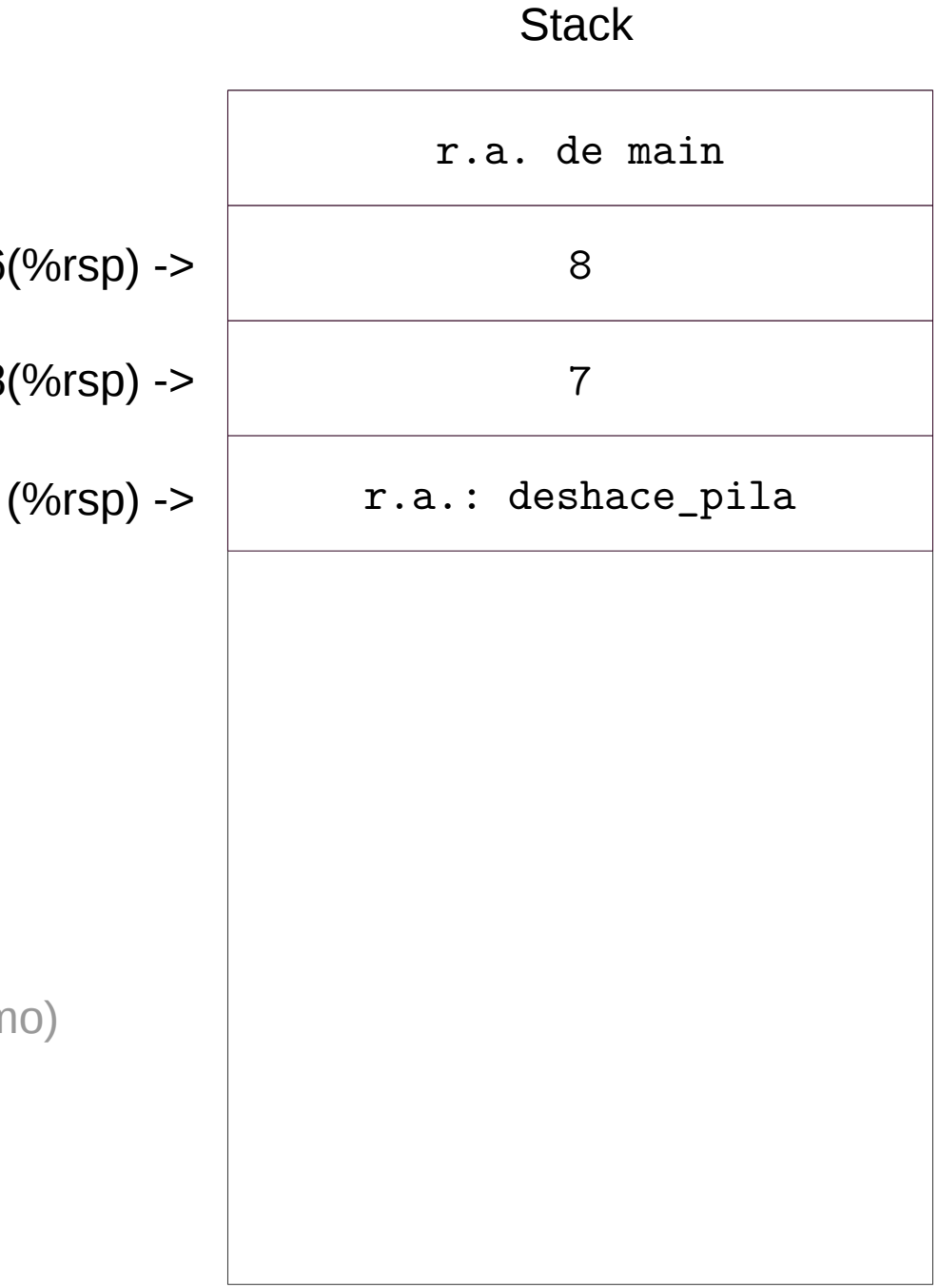
Evolución del stack

```
f:
movq 16(%rsp), %rax

ret

.global main
main:
#Cargar args al stack

cargar_pila:    # O (lo que es lo mismo)
pushq $8        #    subq    $16, %rsp
pushq $7        #    movq    $8, 8(%rsp)
                #    movq    $7, (%rsp)
cargar_regs:
movl$6, %r9d; movl $5, %r8d
movl$4, %ecx; movl $3, %edx
movl$2, %esi; movl $1, %edi
callf: call  f

deshace_pila:  # O (lo que es casi lo mismo)
addq$16, %rsp  # popq %rcx
retorna:       # popq %rcx
ret
```



Evolución del stack

```
f:
movq 16(%rsp), %rax

ret

.global main
main:
#Cargar args al stack

cargar_pila:    # O (lo que es lo mismo)
pushq $8        #   subq    $16, %rsp
pushq $7        #   movq    $8, 8(%rsp)
                #   movq    $7, (%rsp)
cargar_regs:
movl$6, %r9d; movl $5, %r8d
movl$4, %ecx; movl $3, %edx
movl$2, %esi; movl $1, %edi
callf: call f

deshace_pila:  # O (lo que es casi lo mismo)
addq$16, %rsp  # popq %rcx
retorna:       # popq %rcx
ret
```



Evolución del stack

```
f:
movq 16(%rsp), %rax

ret

.global main
main:
#Cargar args al stack

cargar_pila:    # O (lo que es lo mismo)
pushq  $8       #    subq     $16, %rsp
pushq  $7       #    movq     $8, 8(%rsp)
                #    movq     $7, (%rsp)

cargar_regs:
movl$6, %r9d; movl $5, %r8d
movl$4, %ecx; movl $3, %edx
movl$2, %esi; movl $1, %edi
callf: call  f

deshace_pila:   # O (lo que es casi lo mismo)
addq$16, %rsp   # popq %rcx
retorna:        # popq %rcx
ret
```

(%rsp) ->

Stack

Evolución del stack

```
f:
movq 16(%rsp), %rax

ret

.global main
main:
#Cargar args al stack

cargar_pila:    # O (lo que es lo mismo)
pushq  $8       #    subq     $16, %rsp
pushq  $7       #    movq     $8, 8(%rsp)
                #    movq     $7, (%rsp)
cargar_regs:
movl$6, %r9d; movl $5, %r8d
movl$4, %ecx; movl $3, %edx
movl$2, %esi; movl $1, %edi
callf: call  f

deshace_pila:  # O (lo que es casi lo mismo)
addq$16, %rsp  # popq %rcx
retorna:       # popq %rcx
ret
```

¿Qué hace el compilador C?

- demo_pasaje_args_4.s y demo_pasaje_args_4.c

```
main:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movq    $8, 8(%rsp)
        movq    $7, (%rsp)
        movl    $6, %r9d
        movl    $5, %r8d
        movl    $4, %ecx
        movl    $3, %edx
        movl    $2, %esi
        movl    $1, %edi
        call    f
        leave
        ret
```

```
f:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, -8(%rbp)
        movq    %rsi, -16(%rbp)
        movq    %rdx, -24(%rbp)
        movq    %rcx, -32(%rbp)
        movq    %r8, -40(%rbp)
        movq    %r9, -48(%rbp)
        movq    24(%rbp), %rax
        popq    %rbp
        ret
```

Ejemplo más complicado (aun)

```
typedef struct { int a, b; double d;} structparm;
structparm s;
Int e,f,g,h,i,j,k;
long double ld;
double m,n;
__m256 y;

extern void func (int e, int f, strcutparm s, int g, int h, long double
ld, double m, __m256 y, double n, int I, int j, int k);

func(e,f,s,g,h,ld,m,y,n,i,j,k)
```

Registros:

```
%rdi:     e
%rsi:     f
%rdx:     s.a,s.b
%rcx:     g
%r8:      h
%r9:      i
```

Reg. p/flotantes

```
%xmm0: s.d
%xmm1: m
%ymm2: y
%xmm3: n
```

Offset de %rsp

```
0:    ld
16:  j
24:  k
```

Más complicaciones: argumentos variables y retornos complicados

- Si una función puede tomar un número variable de argumentos, en `al` debe ir el número máximo de argumentos de punto flotante pasados en registros (`al` no debe ser mayor a 8).
- Retorno de estructuras: el llamante reserva el lugar y la función llamada tiene un primer argumento oculto (en `rdi`): el puntero a ese lugar. En `rax` retornará el puntero pasado en `rdi`. El primer argumento real irá en rsi (y así sucesivamente)

Variables automáticas – ¿punteros a argumentos?

- `demo_vars_en_funciones.c`
- `demo_dir_args.c`

Convención de uso de registros

- Ver apunte

Casos especiales: main y syscalls

- Main: `demo_args_main.c => retorna un int`
- Syscalls (llamadas al sistema):
  - Nº de syscall en `rax`
  - Argumentos en:

    `rdi  rsi  rdx  r10  r8  r9`
  - Instrucción: `syscall`
  - Ejemplo en C: `demo_syscall_en_c.c`
  - Ejemplo en asm: `demo_syscall_en_asm.s`

Número variable de argumentos

- `demo_var_args.c`
- ¿cómo implementarlo en assembler?
  - `rax` contiene máximo número de reg. de punto flotante (de 0 a 8 inclusive).
  - Se usa una `register_save_area`.
  -
```
typedef struct {
   unsigned int gp_offset;
   unsigned int fp_offset;
   void *overflow_arg_area;
   void *reg_save_area;
   } va_list[1];
```

Register save area:

Reg:	Offset:
%rdi	0
%rsi	8
%rdx	16
%rcx	24
%r8	32
%r9	40
%xmm0	48
%xmm1	64
…	…
%xmm15	288

Número variable de argumentos

- Una macro para cada tipo:

  ej.: `va_arg(l, int)`

```
movl l->gp_offset, %eax
cmpl %48, %eax
jae stack
leal $8(%rax), %edx
addq l->reg_save_area, %rax
movl %edx, l->gp_offset
jmp fetch
```

```
stack:
    movq l->overflow_arg_area, %rax
    leaq 8(%rax), %rdx
    movq %rdx,l->overflow_arg_area
fetch:
    movl (%rax), %eax
```

setjmp y longjmp

```
demo_setjmp.c

demo_setjmp2:

    gcc demo_setjmp2_main.c demo_setjmp2_calculo.c -o demo_setjmp2
```

Corrutinas

- Construyendo corrutinas:
  - Las implementaremos con funciones de C
  - Para cambiar entre corrutinas podríamos usar `setjmp` y `longjmp => TRANSFER(origen, destino)`
  - Se necesita reservar una porción del stack para cada función: `hace_stack`

Otras convenciones

- Previos a `x86_64` había una gran variedad
- `x86_64`: sólo dos convenciones:
  - System V AMD64 ABI (la que ya vimos)
  - Microsoft x64 calling convention:
    - Registros rcx, rdx, r8 y r9 (y punto flotante), 32 bytes de espacio “shadow” en el stack, los registros sólo pueden ser usados por los argumentos 1 a 4.

Funciones anidadas

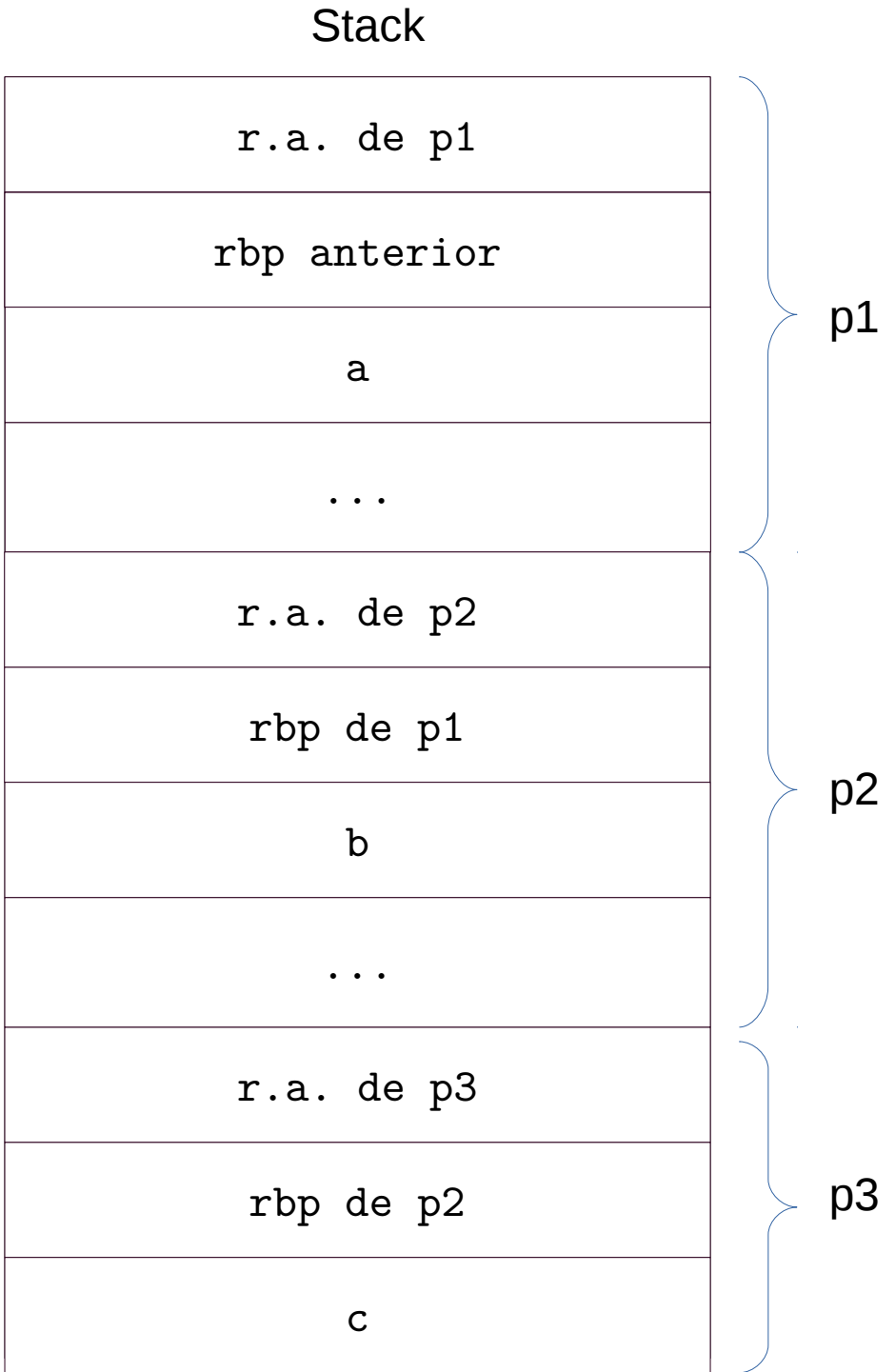
- En C no existen funciones anidadas, pero Pascal (por ejemplo) permite algo como:

```
procedure p1;
    var a: Integer;
    procedure p2;
        var b: Integer;
        procedure p3:
            var c: Integer;
            begin
                c:=b*a;
            end
        begin
            b=a+1;
            p3();
        end
    begin
        a=5;
        p2();
    end;
```

Funcs. Anidadas ¿cómo implementarlas?

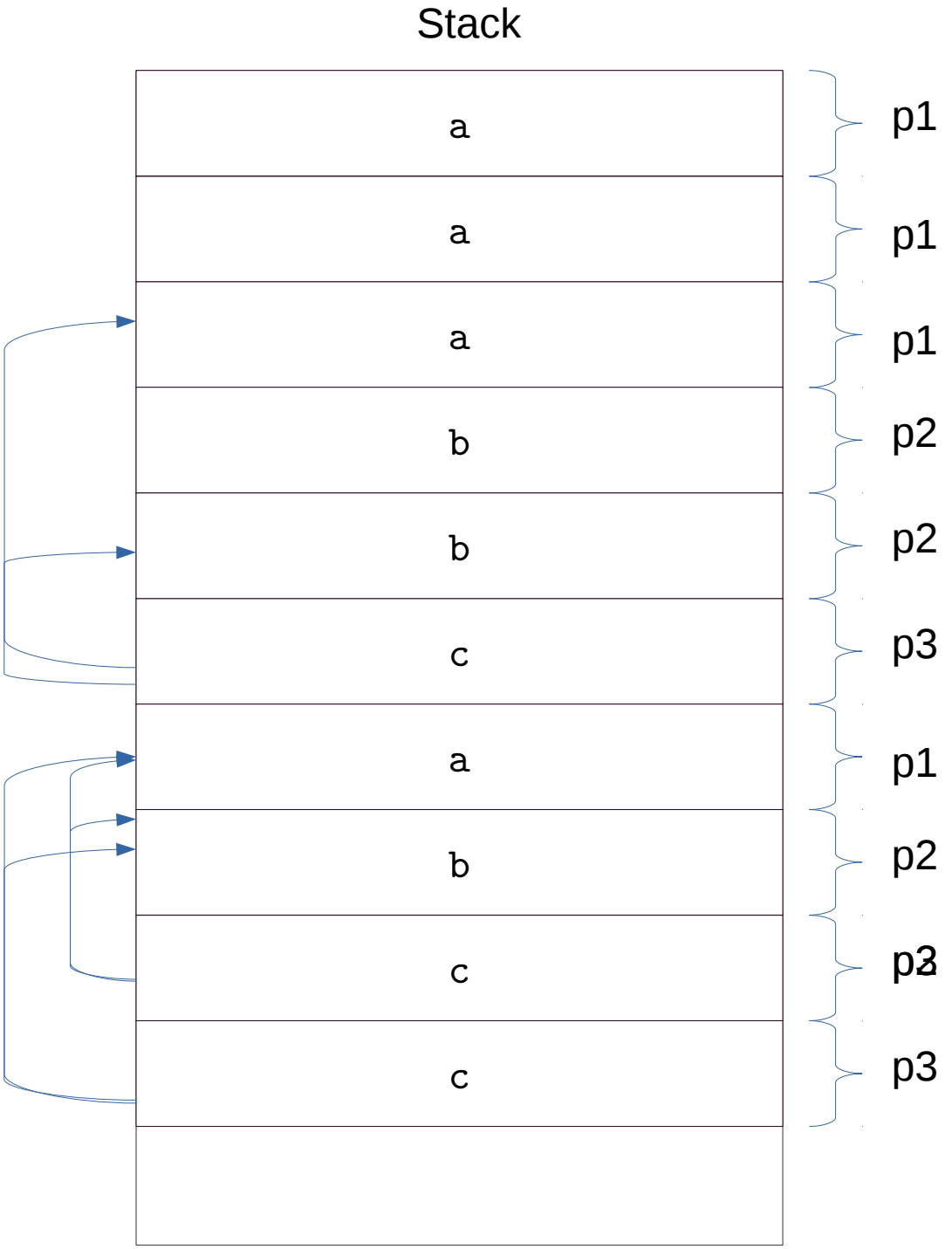
- Intentamos usar el rbp anterior: -8(%rbp DE p1) ->

-8(%rbp DE p2) ->



Funcs. Anidadas ¿recursión?

- p1 podría ser recursiva
- ¡p2 y p3 también!
- ¿cómo se calcula c?



Una solución: usar punteros

```
procedure p1;
    var a: Integer;
    procedure p2;
        var b: Integer;
        procedure p3:
            var c: Integer;
            begin
                c:=b*a;
            end
        begin
            b=a+1;
            p3();
        end
    begin
        a=5;
        p2();
    end;
```

Equivaldría al Código C:

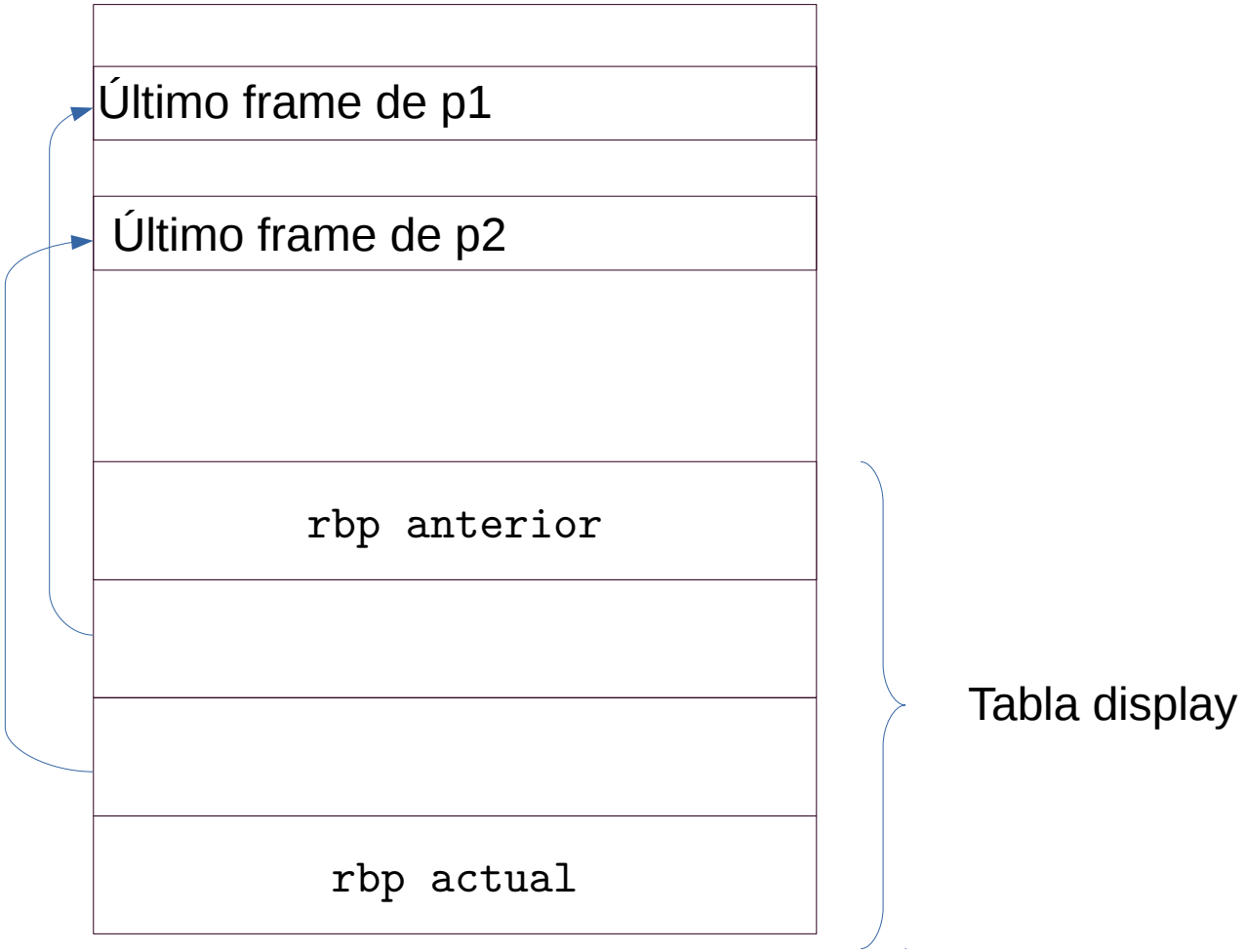
```
void p3(int *a, int *b) {
    int c=(*a)*(*b);
}

void p2(int *a) {
    int b=(*a)+1;
    p3(a,&b);
}

void p1() {
    int a=5;
    p2(&a);
}
```

Mejorando: tabla display

- Notar que:
  - Cada variable se puede calcular con respecto al comienzo del marco de activación correspondiente.
  - Para cada nivel de anidación sólo se necesita el último marco
- O sea: necesitamos un puntero por cada nivel de anidación y con eso se pueden calcular en tiempo de compilación la posición de cualquier dato necesario

Tabla display

- Guardar al comienzo del marco de activación los punteros a la funciones de niveles superiores. A esto (más el rbp anterior) se lo llama “tabla (o area) display”:



Otra solución: static link