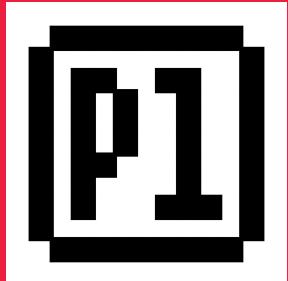


La memoria y los punteros

UNRN

Universidad Nacional
de Río Negro

r20



A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?



Las regiones de memoria

**Todas las variables
tienen un lugar en la
memoria**

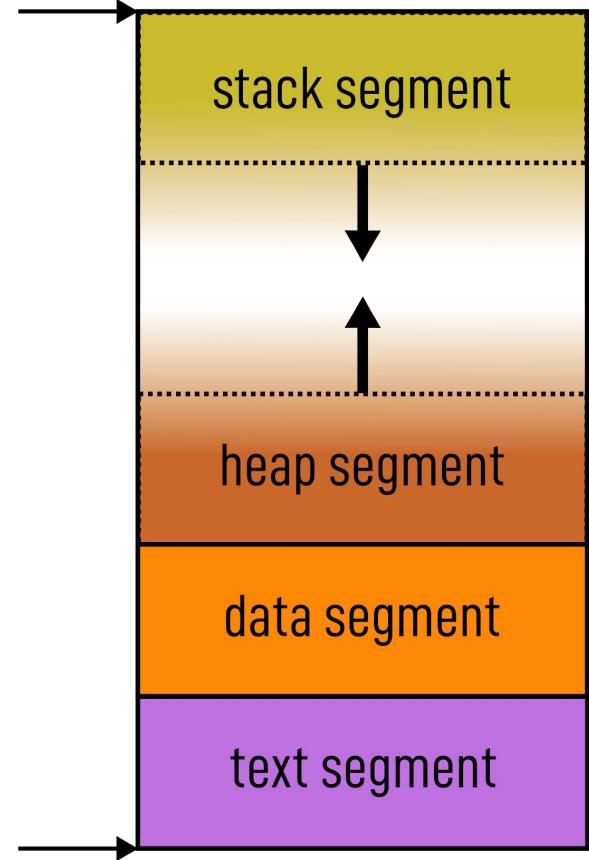
Segmentos de memoria

En donde se alojan
las variables

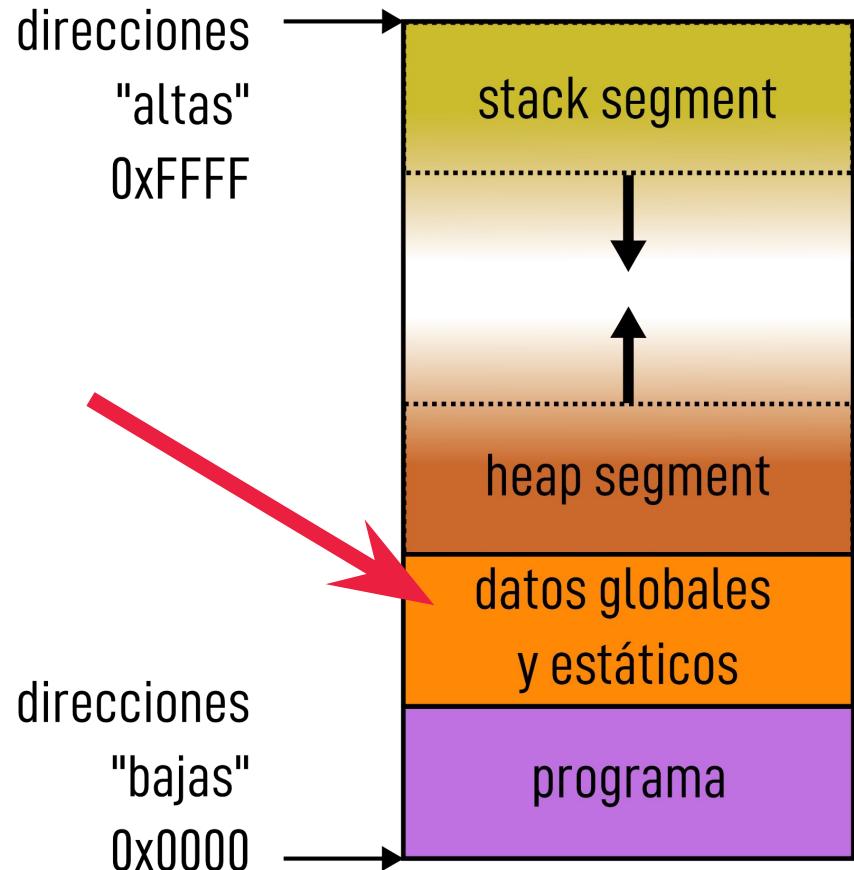
El mapa de la memoria

direcciones
"altas"
0xFFFF

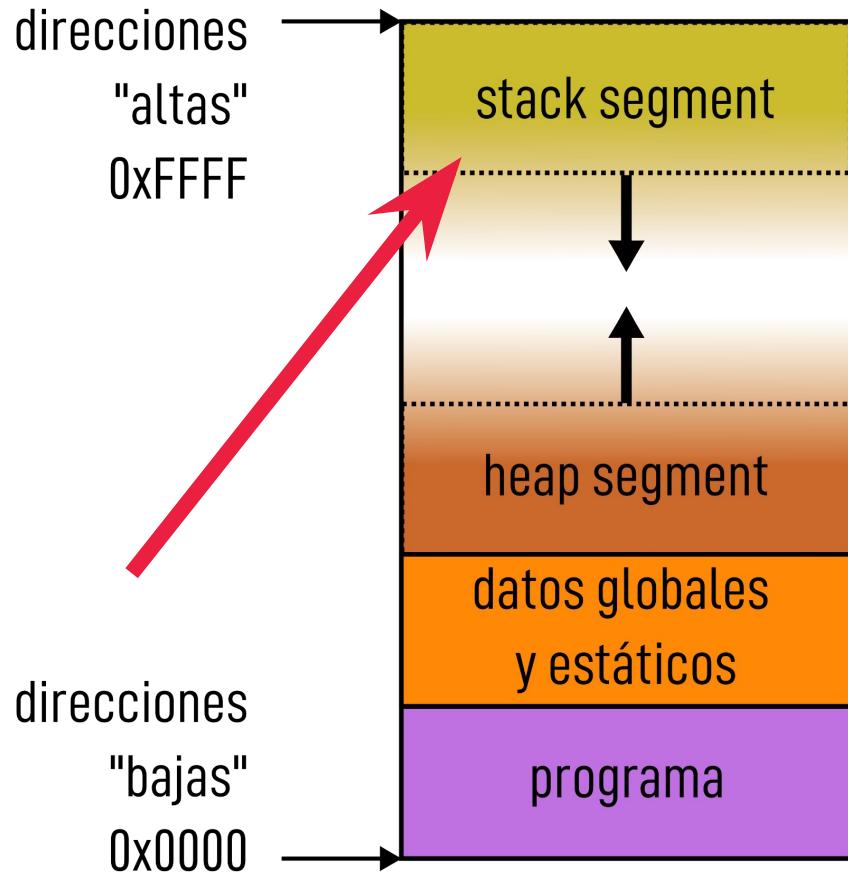
direcciones
"bajas"
0x0000



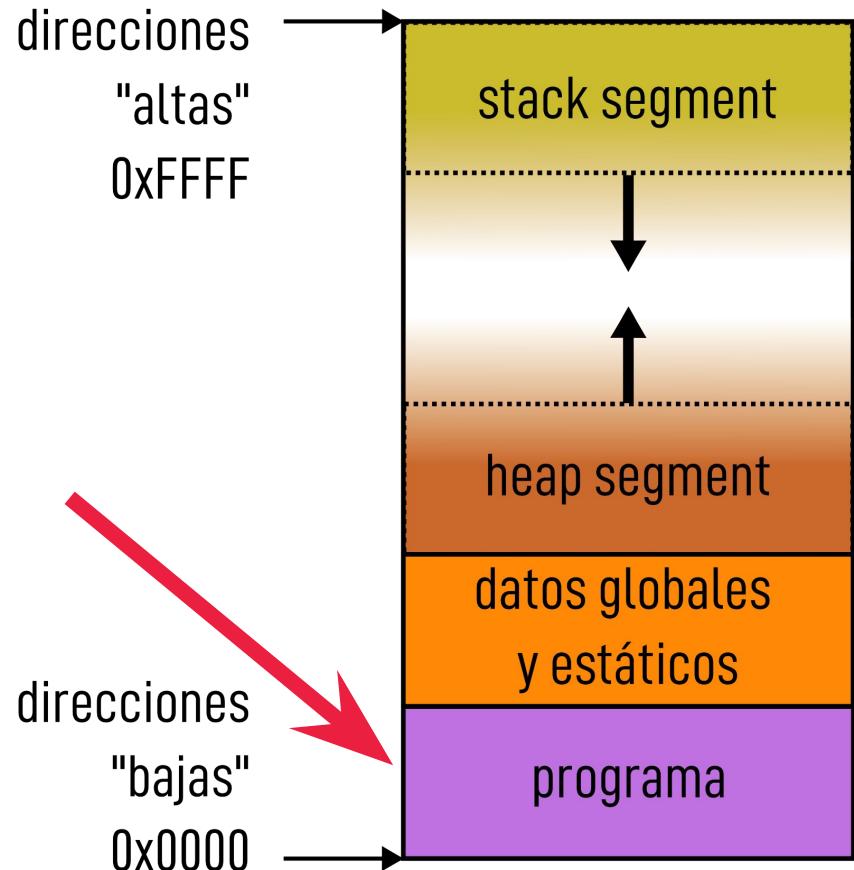
memoria global



memoria automática

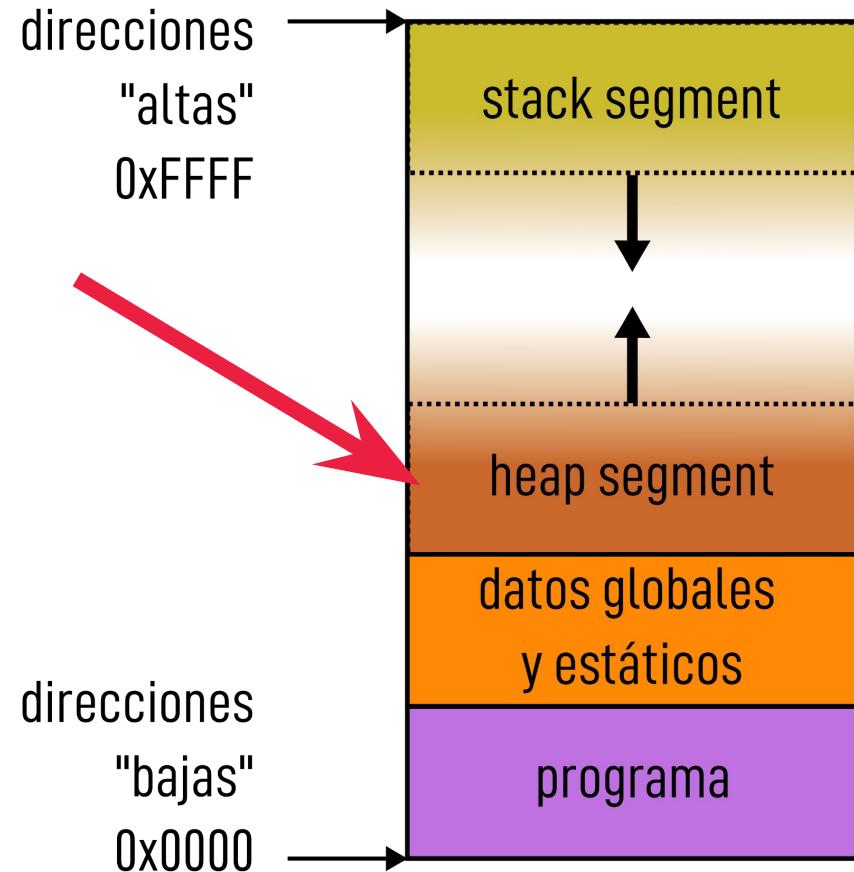


El código del programa



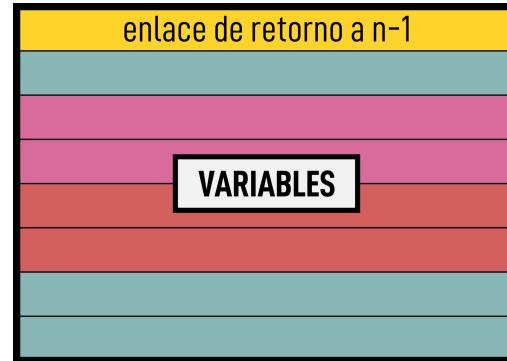
Próximamente
en

memoria dinámica



stack memoria automática

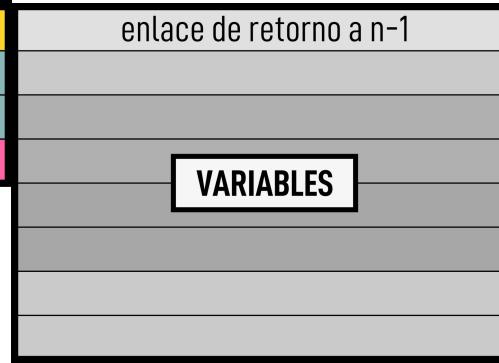
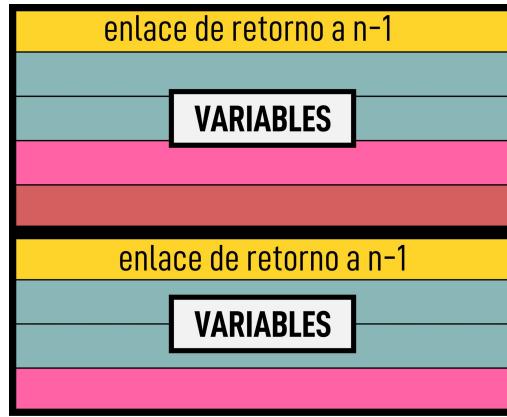
Sean 3 funciones

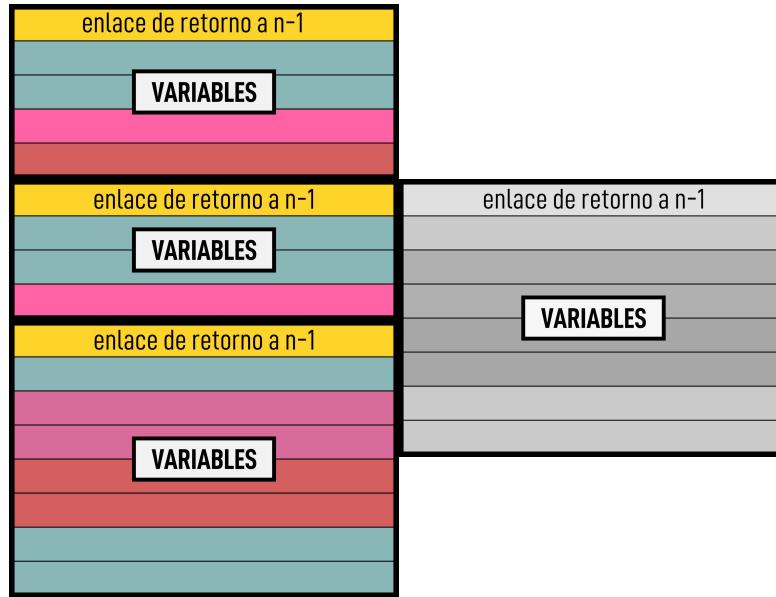


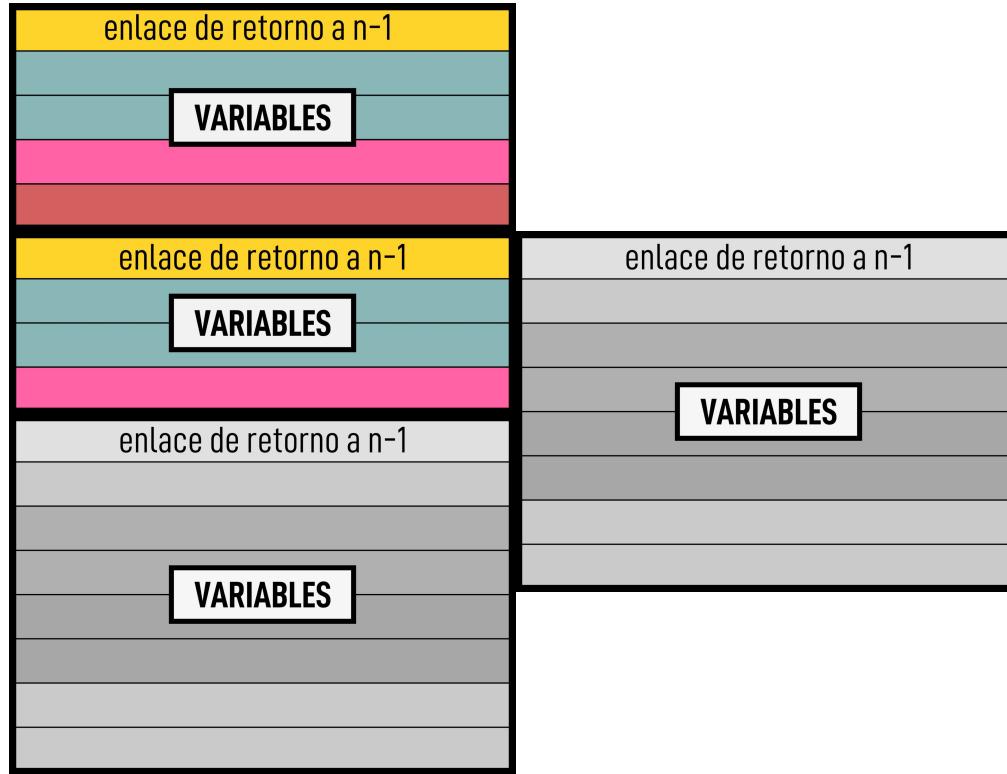


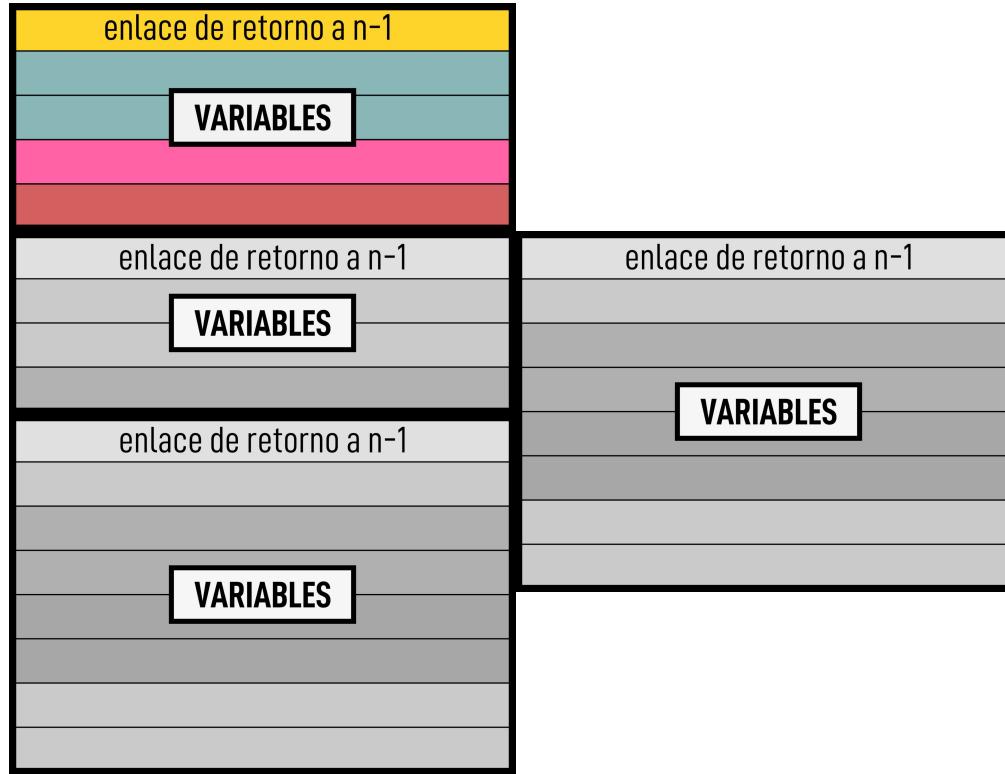


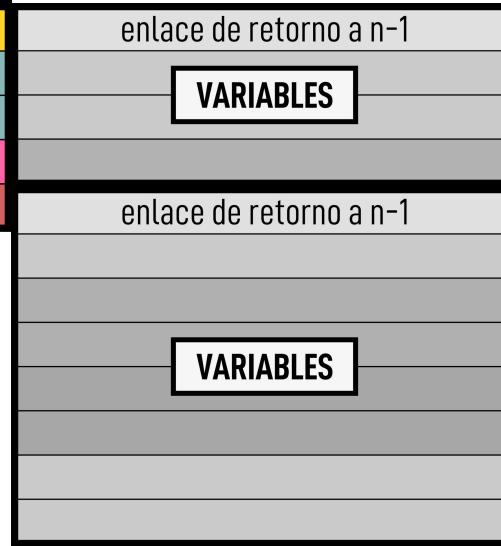
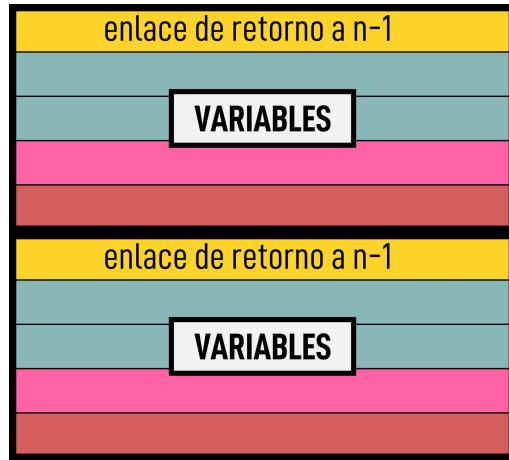




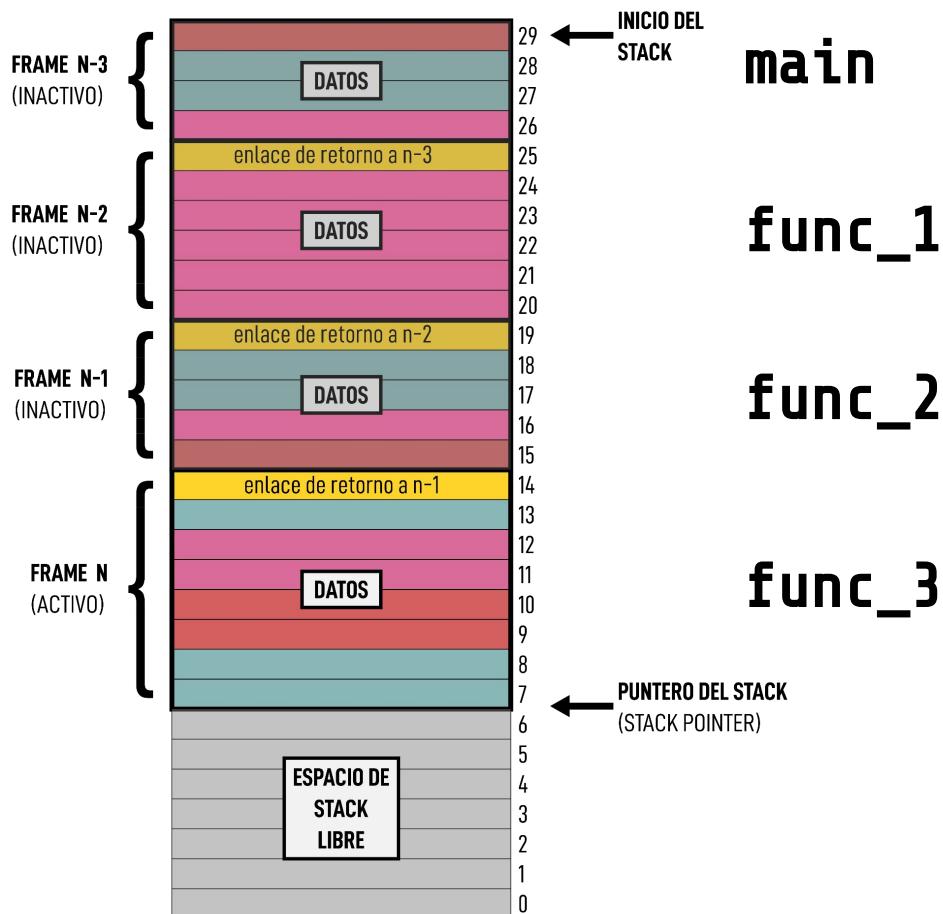












**El stack reutiliza
memoria con cada
llamada a función**

**La memoria tal vez
venga en 0 la
primera vez**

Pero con total seguridad

¡Después no!

**¡A la
terminal!**



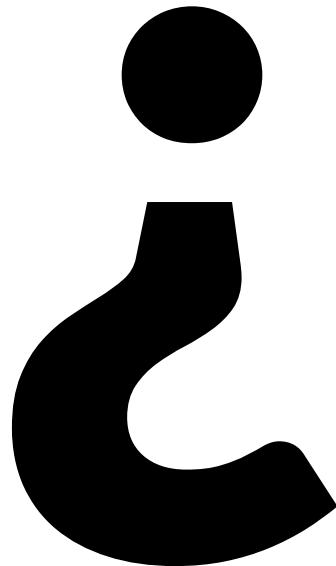
A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?



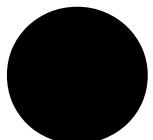
punteros



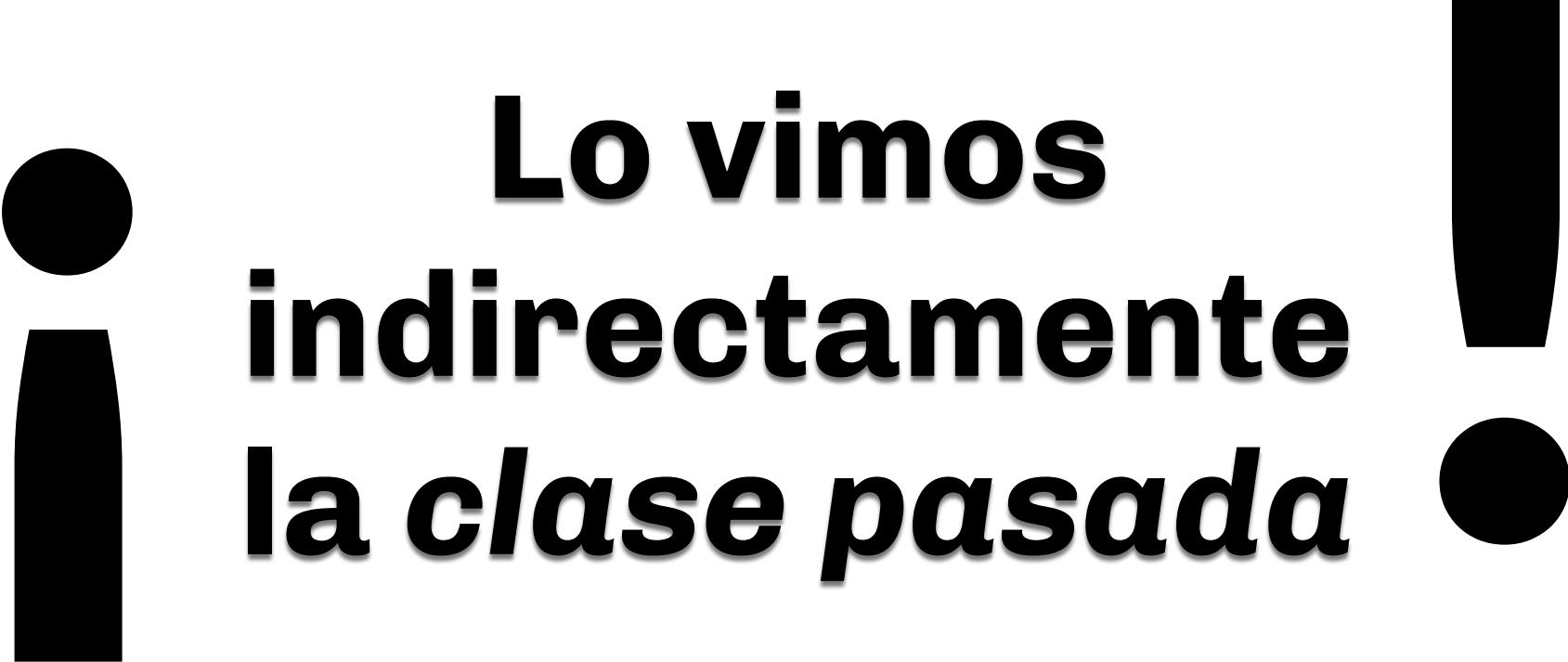


qué son





**Lo vimos
indirectamente
la clase pasada**



**Lo vimos
indirectamente
la clase pasada**

Con los arreglos

**O, por qué un arreglo
cambia fuera, pero
dentro de una función**

Para una función del tipo

```
void funcion(int tamanio, int arreglo[])
```

—
tipo *

Puntero a

Declaración de una variable de tipo puntero

a

tipo

*identificador;



```
int *ptr;
```



Y como le decimos a
donde apuntar

Una variable ‘puntero a’ int llamada ptr

Ojo que, un puntero como r-value **puede recibir**
números

```
ptr = 10;
```



¡Pero no debe! Porque el contenido de la variable

no es un
número

**ptr contiene
una dirección de
memoria**

Asignación con la dirección de otra variable

```
int valor = 10;  
int *ptr = &valor;
```

¡Los tipos de lo apuntado y el ‘apuntado’ deben coincidir!

&variable Operador “dirección de”

• Ah, como en !
i scanf •

La inicialización en ‘cero’

```
int *ptr = NULL;
```

Apuntando a ‘ningún’ lado. Está definido en stddef.h

**¿Pero que
podemos
hacer?**

```
int variable = 10;  
int* ptr = &variable;  
*ptr = 30;
```

—

*puntero Operador “contenido de”

Un ejemplo múltiple

```
int contador = 10;
```

```
int *puntero = &contador;  
*puntero = *puntero + 1;
```

Inspeccionando punteros

```
int numero = 10;  
int *puntero = &numero;  
  
printf("direccion de numero %p\n", &numero);  
printf("contenido de puntero %p\n", puntero);  
printf("direccion de puntero %p\n", &puntero);  
printf("apuntado por puntero %d\n", *puntero);
```

De paso vemos %p de printf

**¡A la
terminal!**



El tamaño de los punteros

Es siempre el mismo

`sizeof(int*) == sizeof(char*)`

Ojo con el *

```
int contador = 10;
```

```
int *puntero = &contador;
```

```
*puntero = *puntero * 2;  
(*puntero)++;
```



A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?



Efectos en él pasaje de argumentos

**Siempre y solo por
valor**

**¿Pero si pasamos
un puntero?**

**Y por referencia si
pasamos un puntero**

De-referenciación en funciones

```
void intercambio(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

**¡A la
terminal!**



Documentando pasaje por referencia

```
/**  
 * Este procedimiento toma dos punteros a enteros y intercambia los valores a los que apuntan.  
 *  
 * @param izquierdo - Puntero al primer entero.  
 * @param derecho - Puntero al segundo entero.  
 *  
 * @pre Los punteros a 'izquierdo' y 'derecho' deben apuntar a  
 *      direcciones de memoria válidas que contengan enteros.  
 *  
 * @post Despues de llamar a esta función, los valores apuntados  
 *       por 'izquierdo' y 'derecho' habrán sido intercambiados.  
 *  
 */  
void intercambio(int *izquierdo, int *derecho);
```

PUNTEROS



Otro uso...

Devolviendo tres resultados desde una función

```
int division_lenta(int dividendo, int divisor,  
                   int *cociente, int *resto);
```

Y se llamaría

```
int dividendo = 10;  
int divisor = 7;  
int cociente;  
int resto;  
int estado;  
estado = division_lenta(dividendo, divisor, &cociente, &resto);
```



¿Preguntas?



Arreglos II

Parecidos pero no iguales

`char[] ≠ char*`

Libertad de asignación

Un puntero puede reasignarse para apuntar a diferentes ubicaciones en la memoria.

Pero un arreglo está asociado con una ubicación específica en la memoria y no puede reasignarse.

¡Esto no funciona!

```
char cadena[40] = "Hola Mundo";
cadena = "chau mundo";
```

Tamaño

Un arreglo tiene un tamaño fijo que se determina en el momento de la declaración. Un puntero tiene un tamaño fijo asociado, el necesario para almacenar la dirección de memoria.

Degradación

Cuando pasas un arreglo como argumento a una función, se *degrada* a un puntero automáticamente.

Esto significa que la función recibirá una dirección de inicio del arreglo y no la longitud real del arreglo.

**¡A la
terminal!**



Razón por la cual hace
falta
capacidad
en el TP3

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?



Aritmética de punteros



¡Con cuidado!!

Que sí nos pasamos...
con suerte nos enteramos

*Hasta acá, lo
normal*

char cadena[7]	h	o	l	a	!	\0		
	0	1	2	3	4	5	6	7

```
char cadena[7]
```

h

o

l

a

!

\0

0

1

2

3

4

5

6

7

```
char *puntero = arreglo;
```

Pero...

Arreglos a la fuerza

```
char cadena[7] = "Hola!";
char* puntero = cadena; // lo 'degradamos'

printf("%s\n", puntero+2);
```

```
char cadena[7]
```

	h	o	l	a	!	\0		
0	1	2	3	4	5	6	7	

```
char *puntero = arreglo;
```

¿cadena+3 es igual a cadena[3]?

cadenas[3] es igual a cadenas[3] ?

cadena[3]

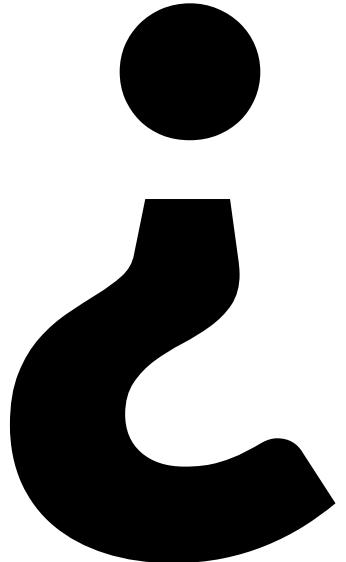


char cadena[7]	h	o	l	a	!	\0		
	0	1	2	3	4	5	6	7

char *puntero = arreglo;

**¡A la
terminal!**

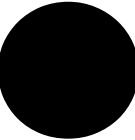




cadena[3]

==

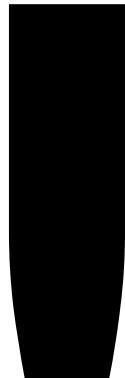
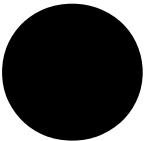
cadena + 3



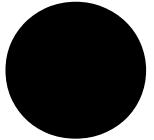
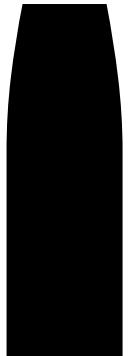
cadena[3] → **char**

≠

cadena + 3 → **char***



cadena[3]

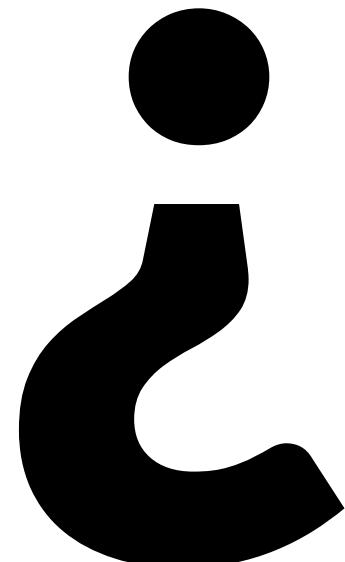
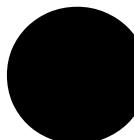


***(cadena + 3)**

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?





**Y esto para
qué**

Dada esta función para cadenas seguras

```
int copiar(int capacidad_origen, char origen[],  
          int capacidad_destino, char destino[],  
          int cantidad);
```

Podemos implementar

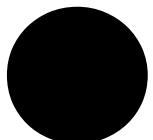
```
int copia_segmento(int capacidad_origen, char origen[],  
                   int capacidad_destino, char destino[],  
                   int desde, int cantidad)  
{  
    return copia(capacidad_origen - desde, origen + desde, \  
                capacidad_destino, destino, cantidad);  
}
```

Aunque incompleto, lo grueso ya está resuelto



¿Preguntas?





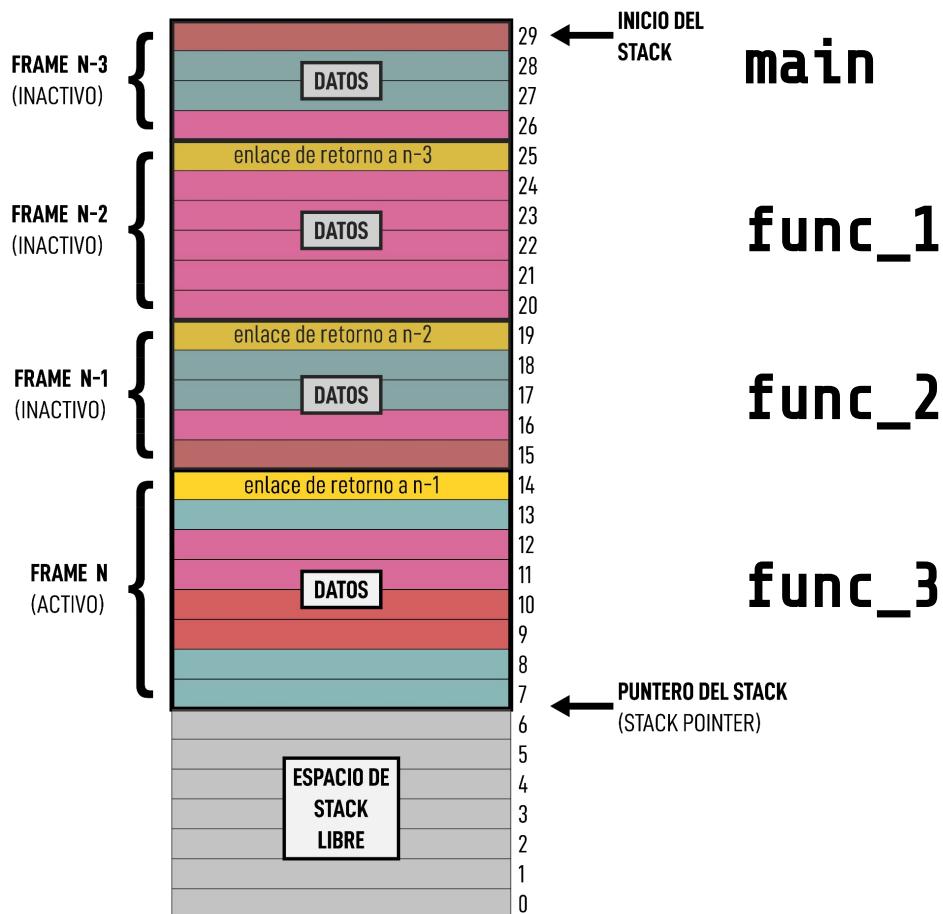
**Que pasa cuando
devolvemos un
puntero de una
variable automática**

¿Qué veríamos afuera?

```
char* puntero_a_cadena()
{
    char cadena[] = "hola mundo";
    return cadena;
}
```

**¡A la
terminal!**





**No nos
pertenece
y cualquier función lo modificará**

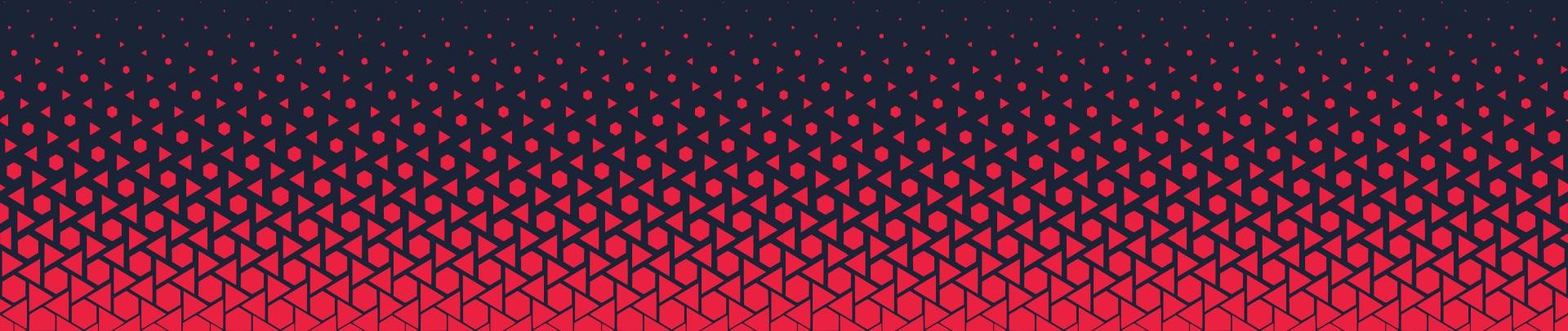
**Y lo pisa cualquier
llamada a función,
incluyendo printf**

A yellow cube with two large white question marks on its faces, resembling a power-up item from the Super Mario video game series.

¿Preguntas?



Hasta la próxima



unrn.edu.ar

UNRN

Universidad Nacional
de Río Negro



| unrnionegro