El lenguaje de Programación C

Fernando J. Pereda <ferdy@gentoo.org>

¿Por qué aprender C?

- Portable y muy extendido
- Estándar (C89, C99)
- El lenguaje de los sistemas
- Un lenguaje fácil (no, no es broma)

¿Por qué ignorar C?

-
-
-
-
- En la Universidad enseñan Java

Hola Mundo

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

int main(int argc, char *argv[])
{
  printf(“Hola Mundo!\n”);
  return EXIT_SUCCESS;
}
```

Variables y funciones

- Zonas de memoria
- Permiten guardar datos
  - Información
  - Código
- Tienen nombre y tipo

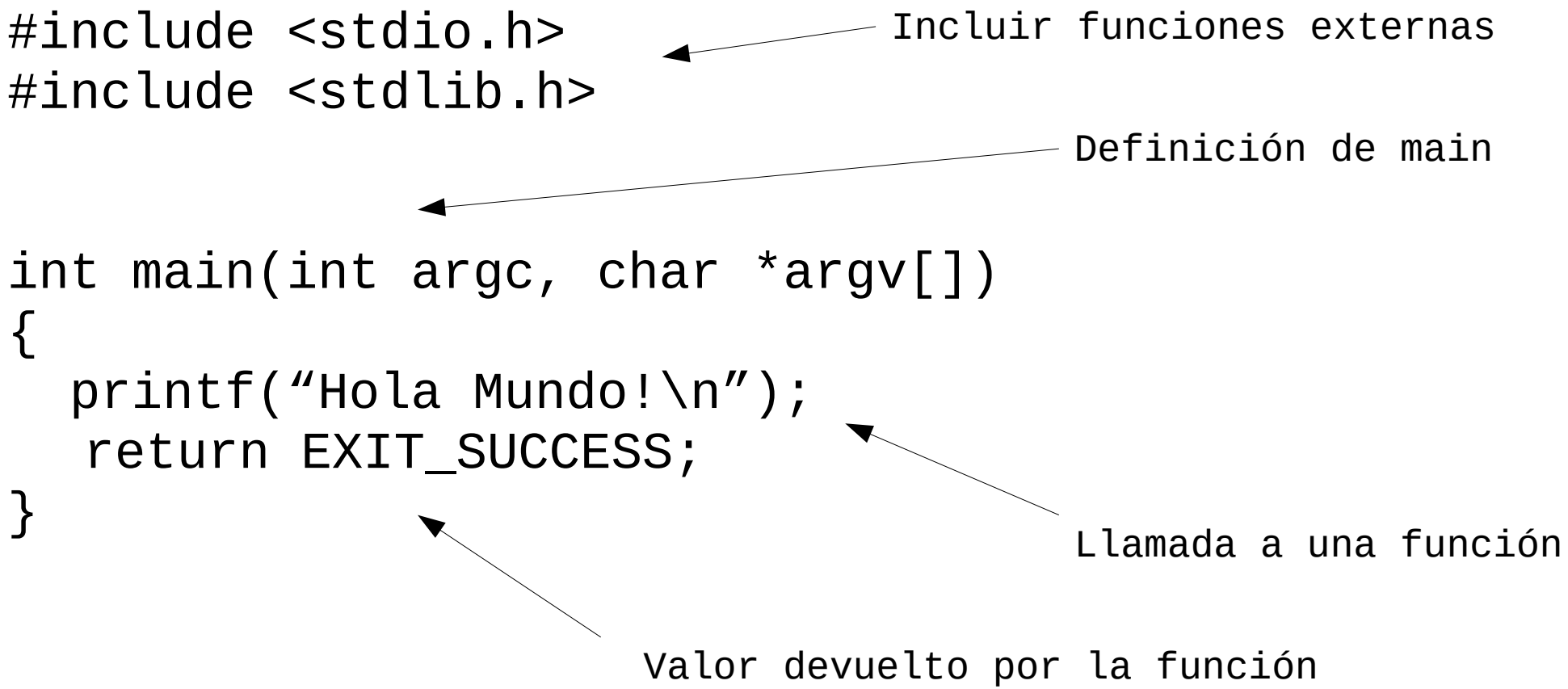
Tipos básicos y arrays

- Numéricos
  - ➔short, int, long, float, double
- Otros
  - ➔char, punteros
- Muchas variables juntas → array

Algunas declaraciones

```
int a;              /* Un entero sin inicializar */

int b = 3;          /* Un entero inicializado */

int a[34];          /* Un array de 34 enteros */

unsigned long n;    /* Un entero largo sin signo */

unsigned c;         /* Un entero sin signo */
```

Hola Mundo – Diseccionado



Definiciones y declaraciones

- Declarar antes de definir o usar
- Las declaraciones se agrupan en ficheros cabecera (algo.h)

```
/*
 * Yo soy un comentario
 * los compiladores me ignoran
 */
int mivariable; /* declaración */
char una_funcion(int a, long b); /* declaración */

int suma(int a, int b) /* definición Y declaración
{                       * de una función que devuelve
   return (a + b);      * un int y recibe dos int.
}                       */
```

Estructuras de control (I)

```
if (a == b) {
   hacer_algo_inteligente();
} else if (! algo) {
   hacer_otra_cosa(b);
} else {
   int i;
   for (i = 0; i < a; i++) {
      switch (f(i)) {
         case 1:
            /* algo especial */
            break;
         default:
            /* algo común */
            break;
      }
   }
}
```

printf

- Imprimir 'cosas'
  - ➔Similar a System.out.println

```
printf(“Hola %s\n“, “Mundo”);

int n = 45;
printf(“n = %d\n”, n);

char *cadena = “Blah”;
printf(“Una cadena: %s\n”, cadena);
```

Estructuras de datos (I)

- Clave para el éxito de C
- Unen variables de distinto tipo

```
struct punto {
   int x;
   int y;
};
```

```
struct persona {
   char nombre[20];
   char dni[10];
};

struct alumno {
   char nia[10];
   struct persona info_personal;
};
```

Estructuras de datos (II)

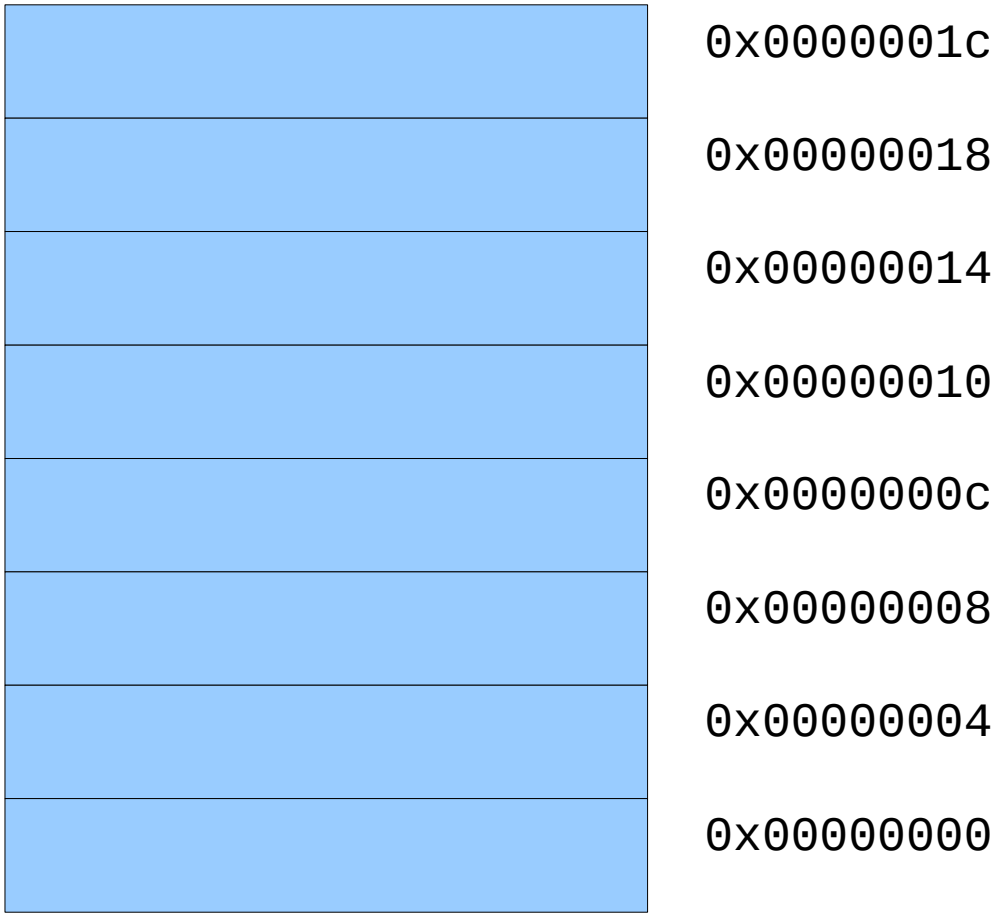
- Funciones que reciben y devuelven estructuras

```
struct punto crear_punto(int x, int y)
{
    struct punto p;
    p.x = x;
    p.y = y;
    return p;
}

int coordenada_x(struct punto p)
{
    return p.x;
}
```

Hemos aprendido

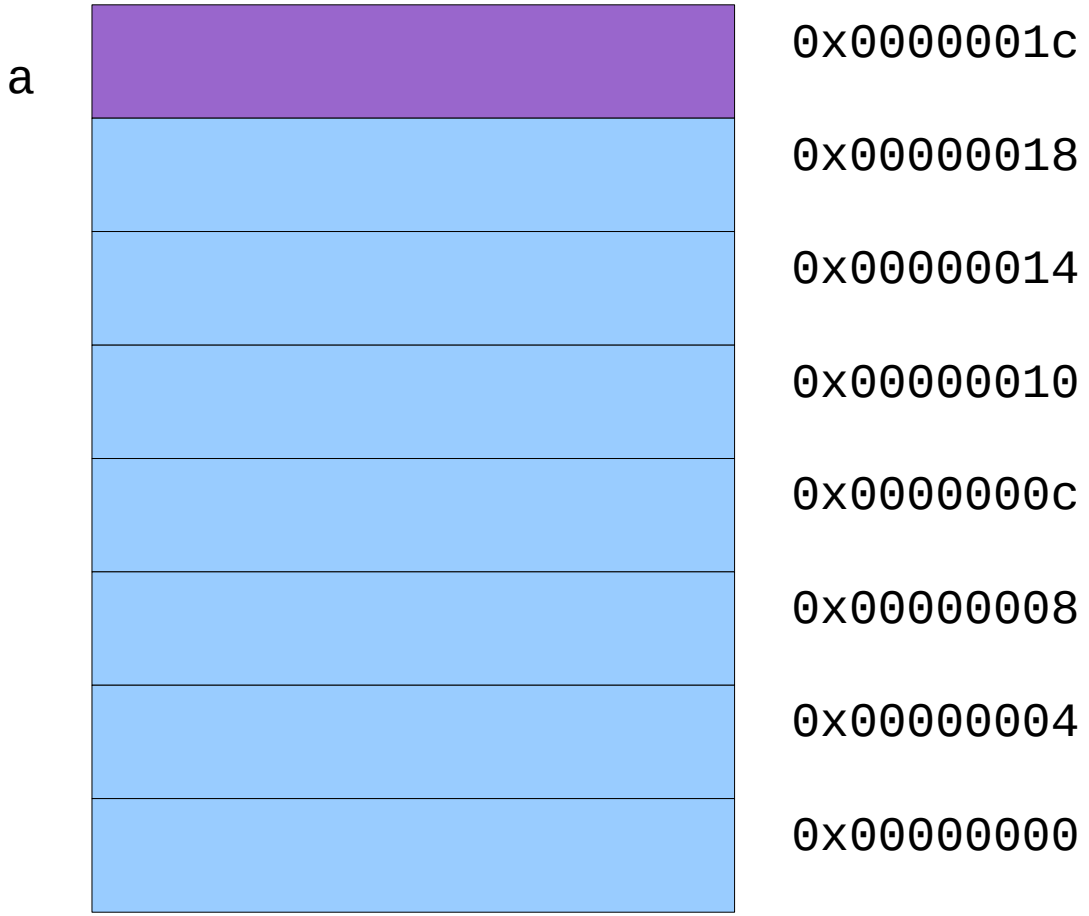
- Declarar variables y funciones
- Definir y llamar funciones
- Controlar el código que se ejecuta
- Crear estructuras de datos

¿Cómo funciona todo esto? (I)



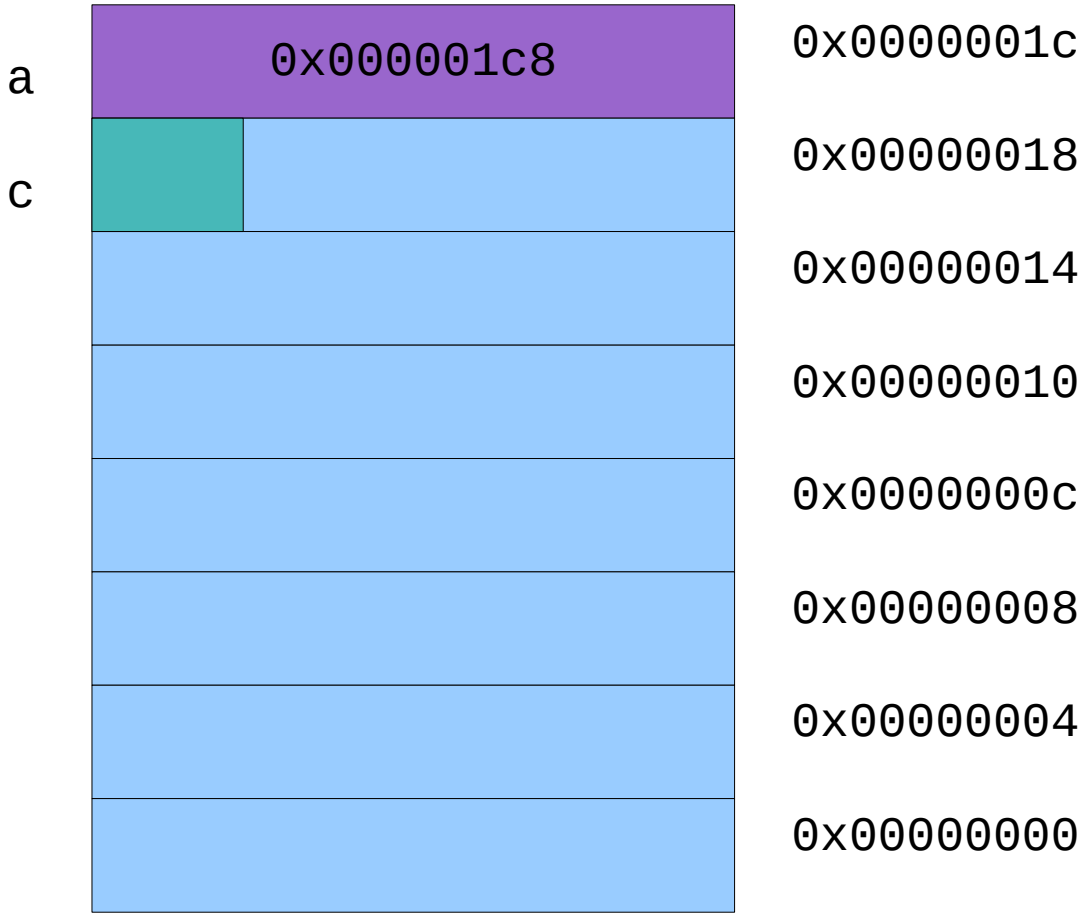
¿Cómo funciona todo esto? (II)

```
int a;
```



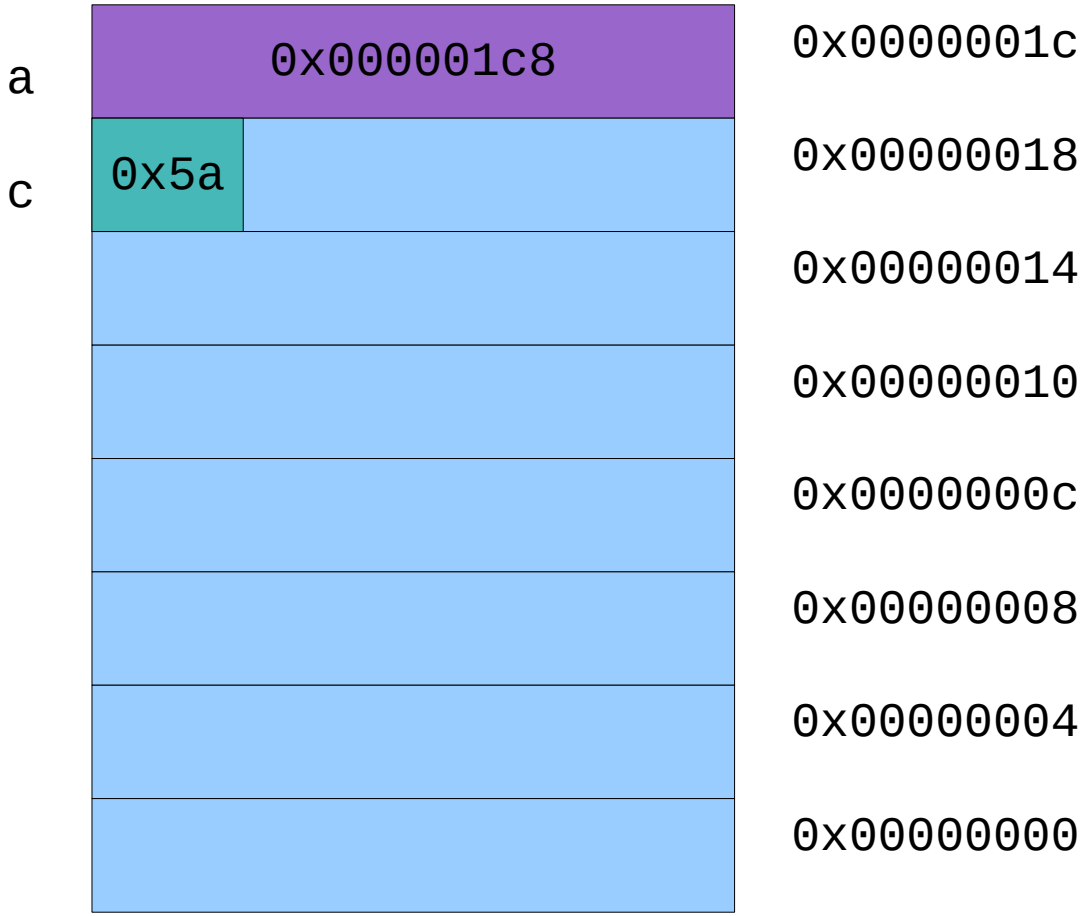
¿Cómo funciona todo esto? (III)

```
int a;
char c;

a = 456;
```



¿Cómo funciona todo esto? (IV)

```
int a;
char c;

a = 456;
c = 'Z';
```



¿Cómo funciona todo esto? (V)

- Paso de parámetros por valor (copia)
  - ¡Las estructuras y los arrays también!
- No existen 'referencias'
  - Rendimiento pobre
  - Poco flexible
    - Las funciones no pueden modificar estructuras ni variables
    - No se puede 'devolver dos valores'

Solución

- Que existan los punteros
- Un puntero es un tipo básico más
- Un puntero guarda la **dirección** de otra variable o función

```
int *p1;
char *p2;
struct mi_struct *p3;
```

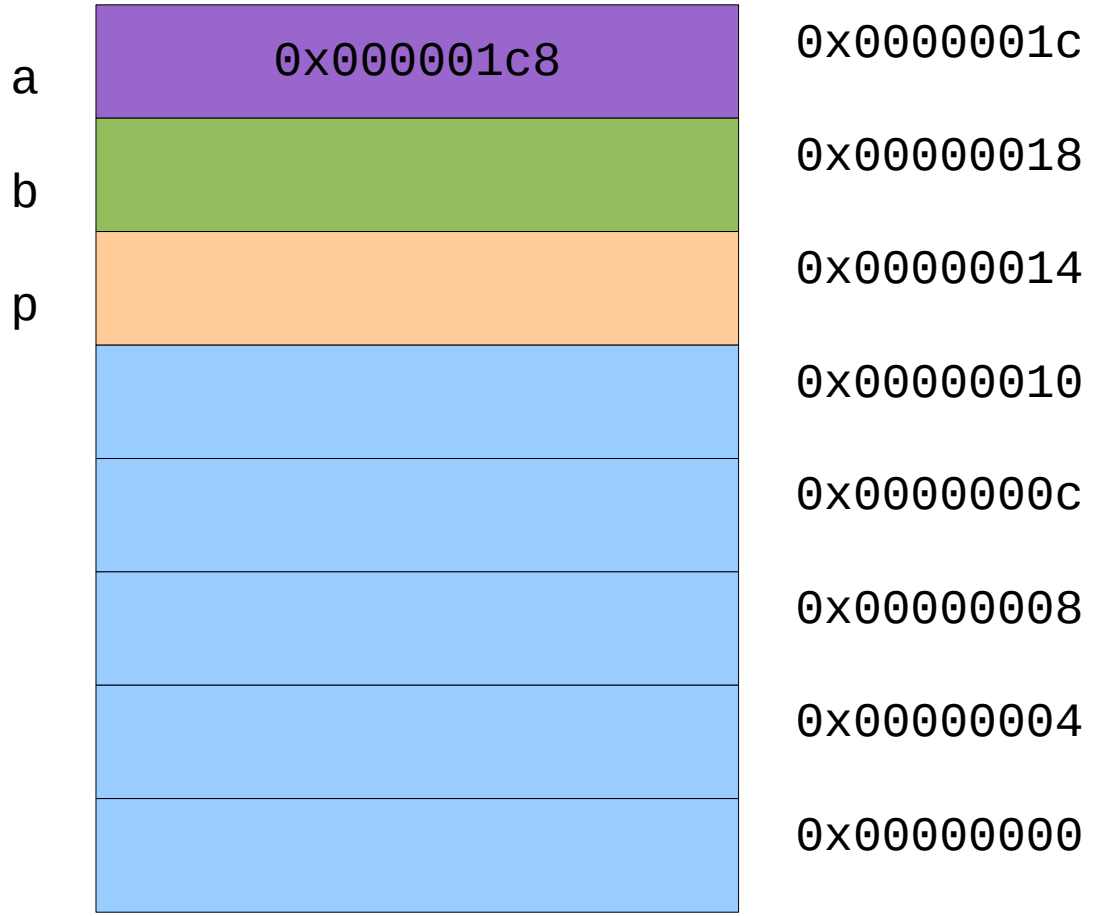
Nuevos operadores

- Obtener dirección: &
- Acceder a lo apuntado: *

```
int a;
int b;
int *p;

a = 34;
p = &a; /* p apunta a a */
b = *p; /* b vale 34 */
```

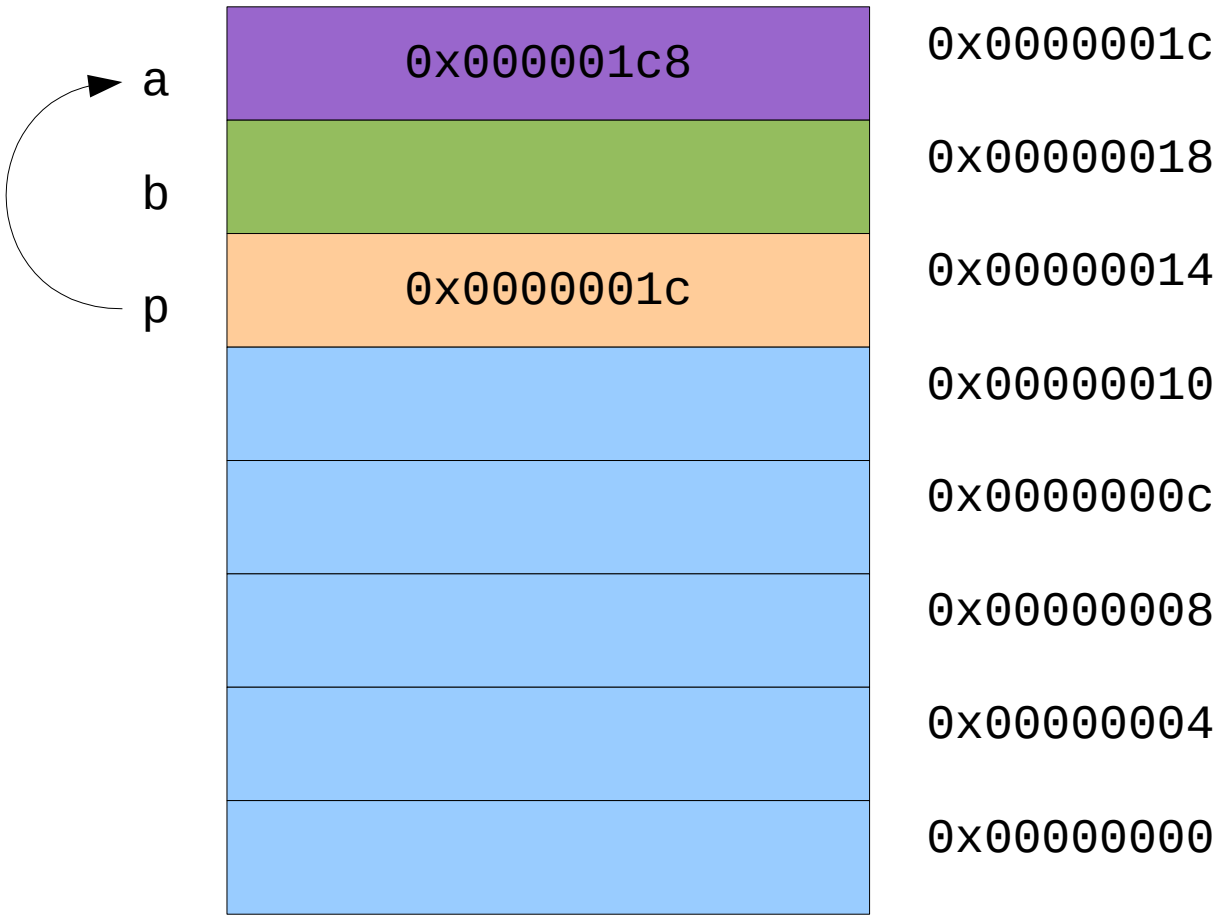
Punteros diseccionados (I)

```
int a;
int b;
int *p;

a = 456;
```



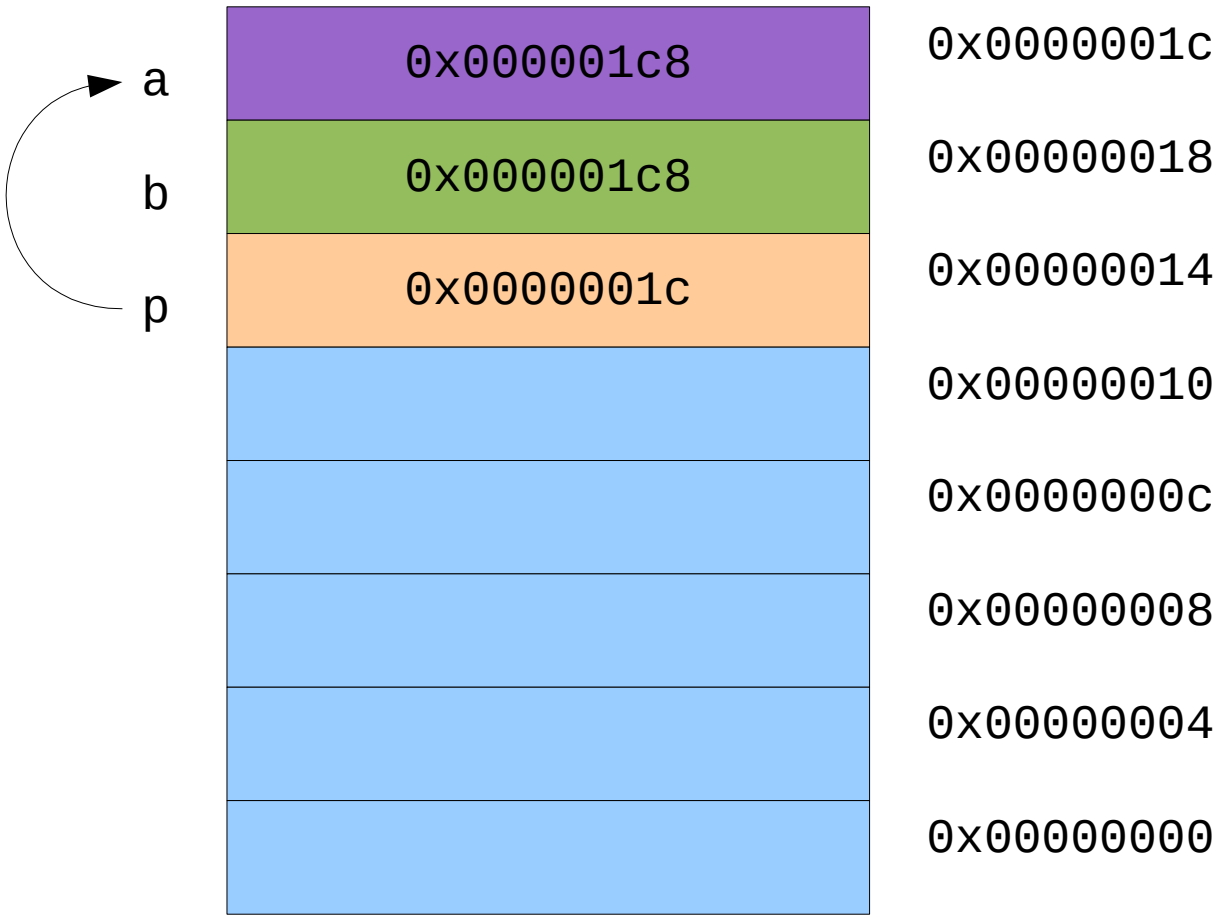
Punteros diseccionados (II)

```
int a;
int b;
int *p;

a = 456;
p = &a;
```



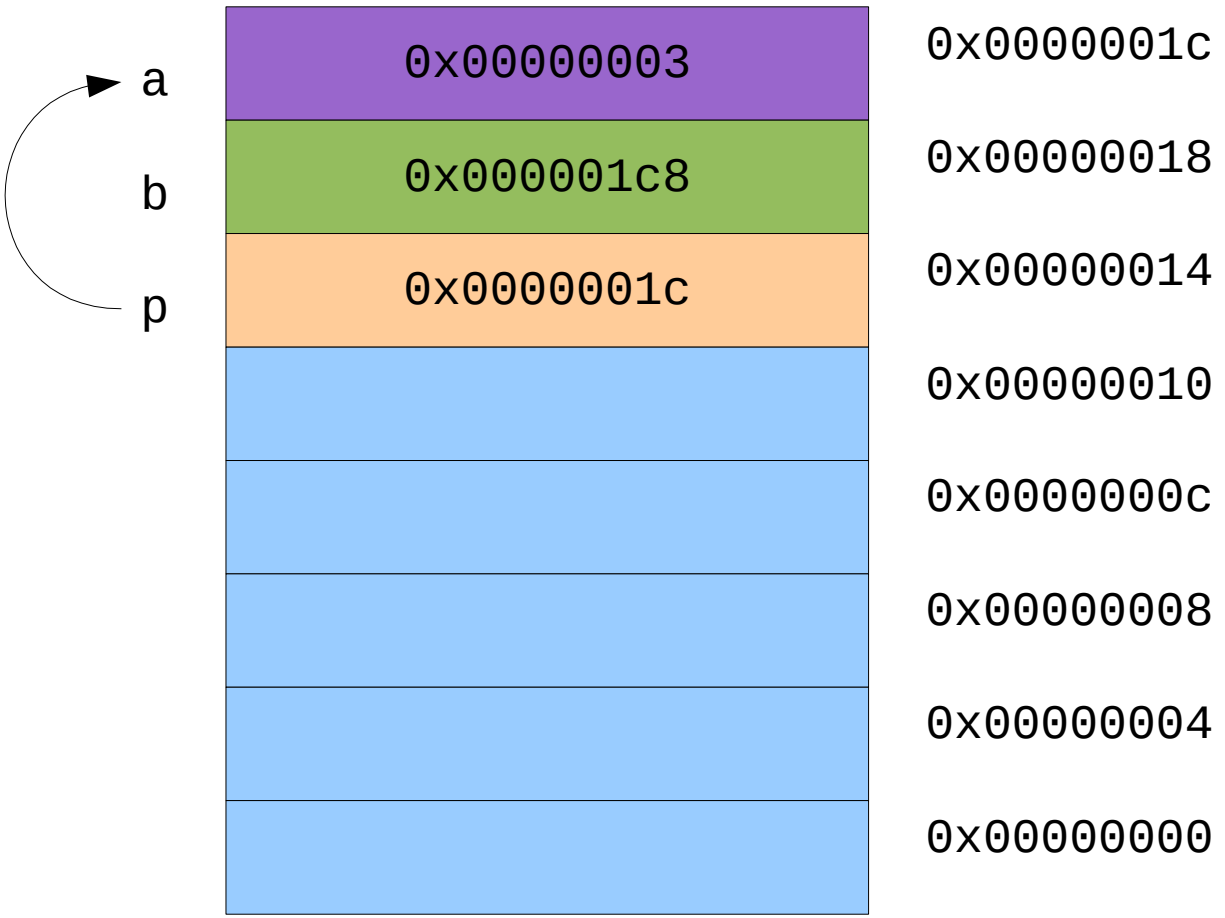
Punteros diseccionados (III)

```
int a;
int b;
int *p;

a = 456;
p = &a;
b = *p;
```



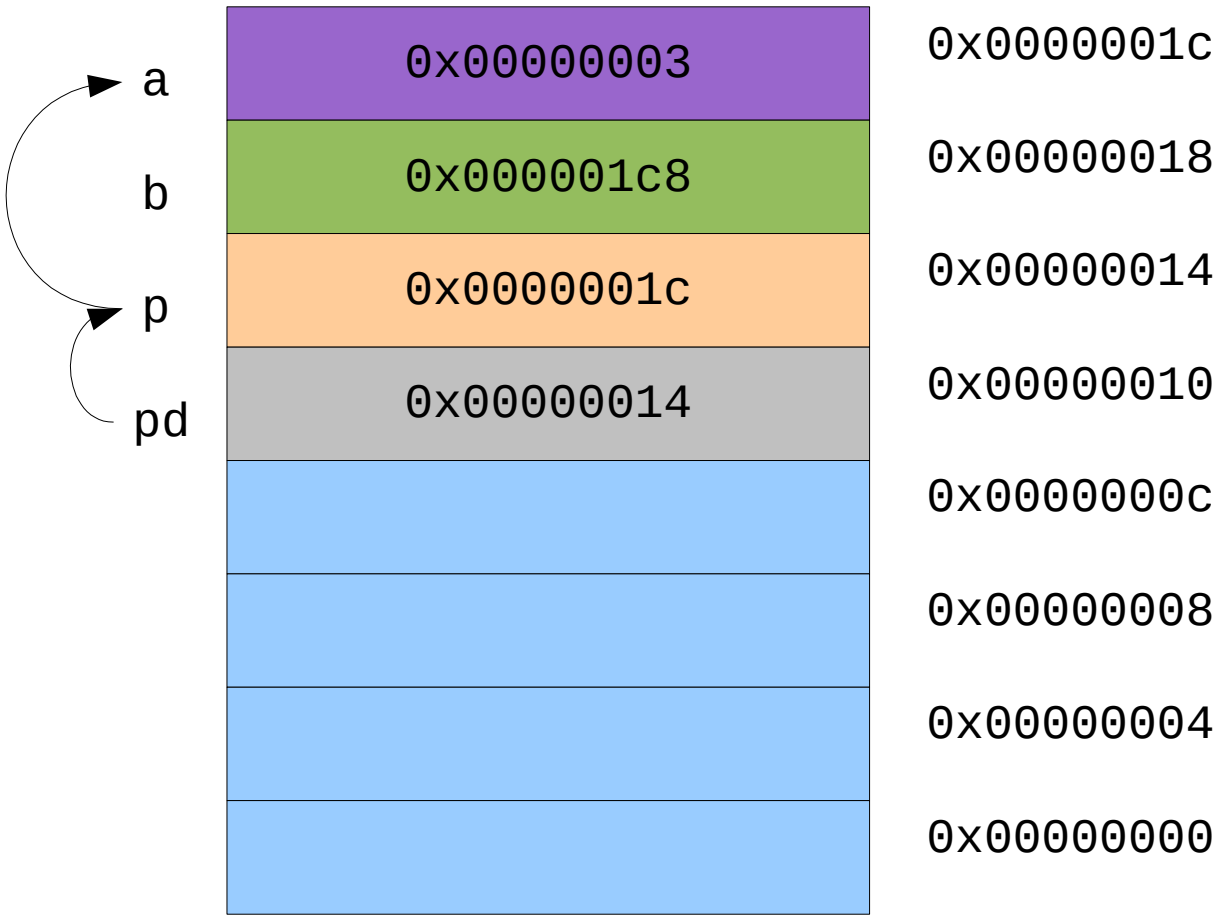
Punteros diseccionados (IV)

```
int a;
int b;
int *p;

a = 456;
p = &a;
b = *p;
*p = 3;
```



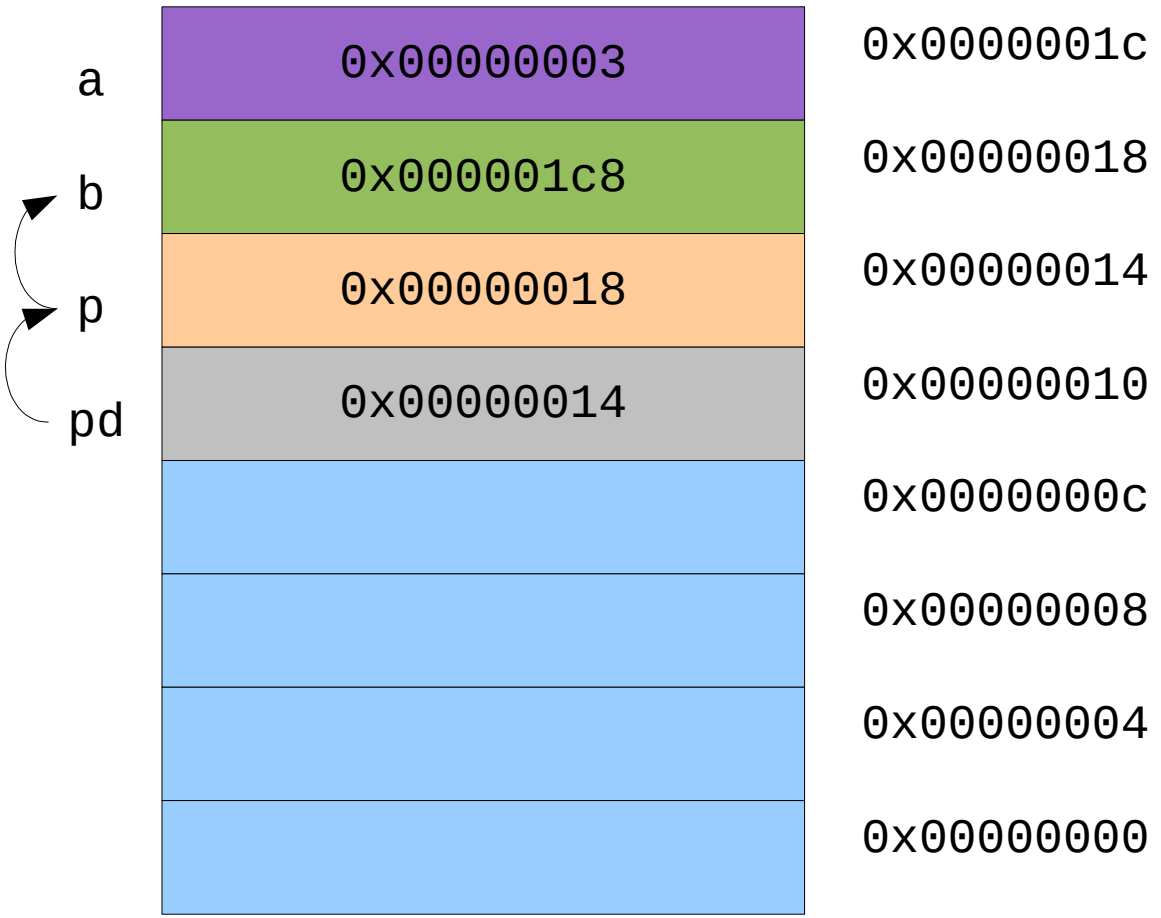
Punteros diseccionados (V)

```
int a;
int b;
int *p;

a = 456;
p = &a;
b = *p;
*p = 3;

int **pd;
pd = &p;
```



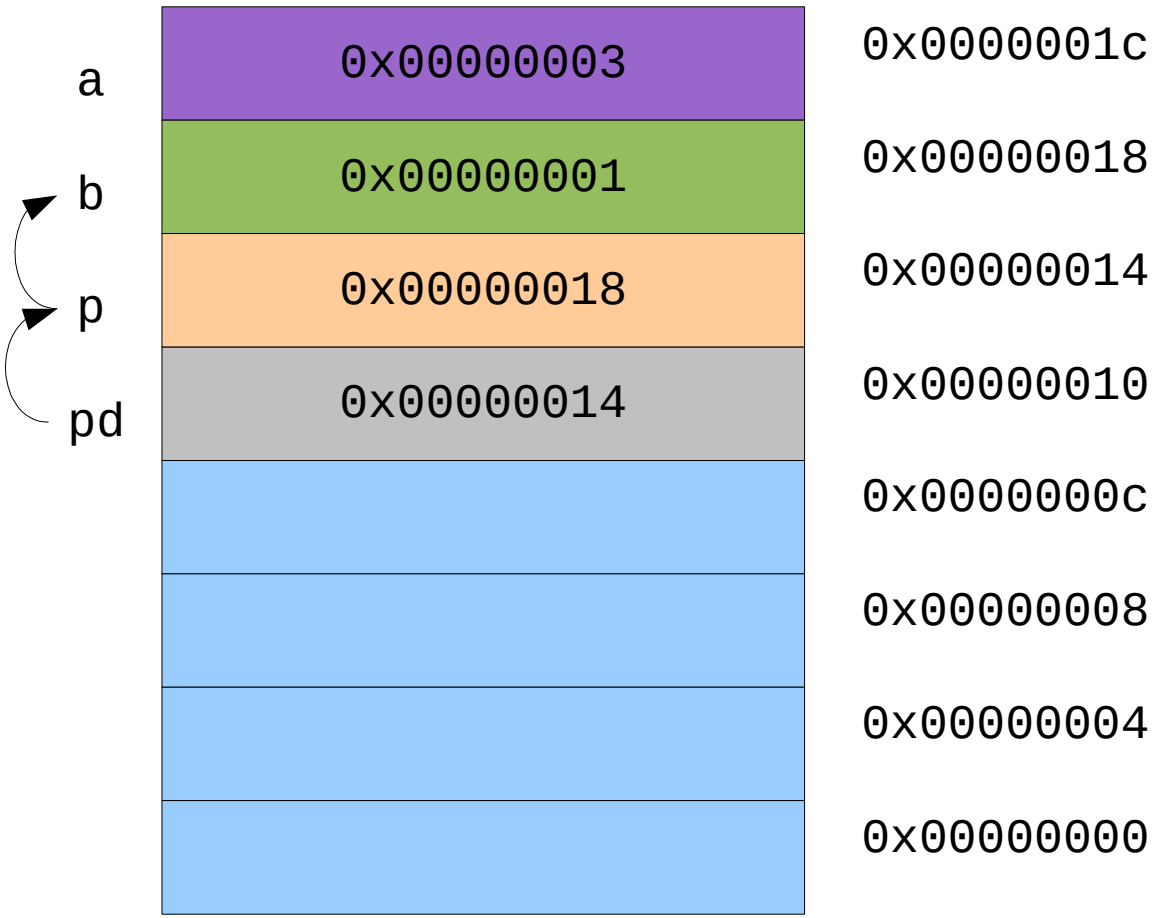
Punteros diseccionados (VI)

```
int a;
int b;
int *p;

a = 456;
p = &a;
b = *p;
*p = 3;

int **pd;
pd = &p;
*pd = &b;
```



Punteros diseccionados (VII)

```
int a;
int b;
int *p;

a = 456;
p = &a;
b = *p;
*p = 3;

int **pd;

pd = &p;
*pd = &b;
**pd = 1;
```



Tamaños

- char <= short <= int <= long
- float <= double
- T* == P*
- El tamaño de una estructura **no** es la suma de los tamaños de sus miembros.

Algunas consideraciones

- Los punteros son 'baratos'
- Son fuente de dolores de cabeza y frustraciones
- Dan miedo
  - ➔ ¡Pero en Java (casi) TODO es un puntero y a nadie le da miedo!

¿Dices que en Java qué? (I)

- Que en Java, un objeto, es un puntero.

```
public class Punteros {
   public static void main(String[] args) {
      String a = null;
      String b = new String(“Algo”);
      boolean iguales = a.equals(b);
   }
}
```

¿Dices que en Java qué? (II)

```
Exception in thread "main" java.lang.NullPointerException
        at Punteros.main(Punteros.java:5)
```

Segmentation Fault

- Equivalente al nullPointerException
- Se obtiene al acceder a una dirección no válida (puntero no válido)
- La única forma de evitarlo es ser capaz de entenderlo (y reproducirlo)

Segmentation Faults (I)

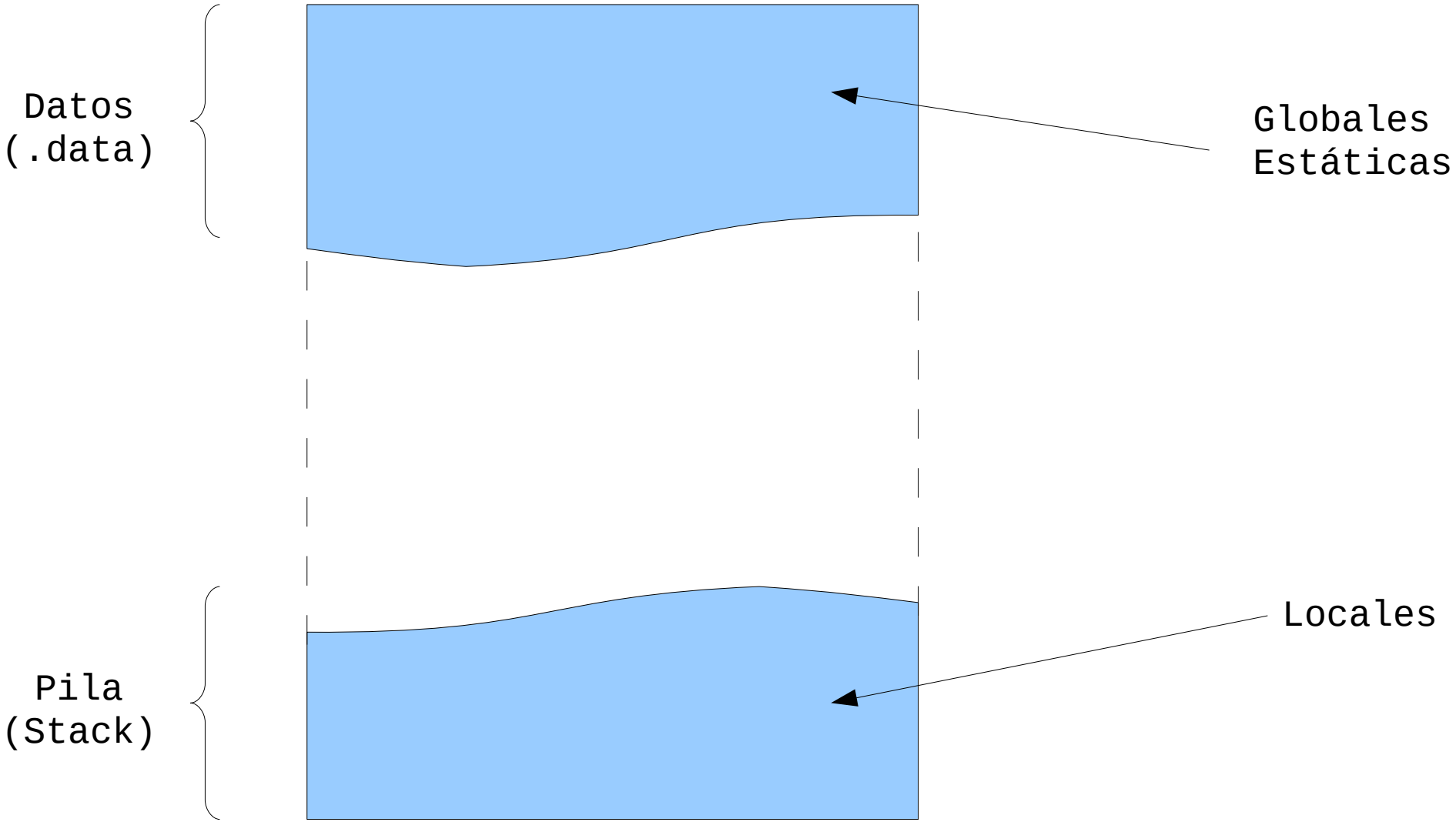
```
int main(int argc, char *argv[])
{
    int *p = 0;
    *p = 23;

    return 0;
}
```

```
int main(int argc, char *argv[])
{
    return *((int *)0);
}
```

¿Dónde viven las variables? (I)

- Globales y estáticas (static)
- Locales
- Registros (register)

¿Dónde viven las variables? (II)

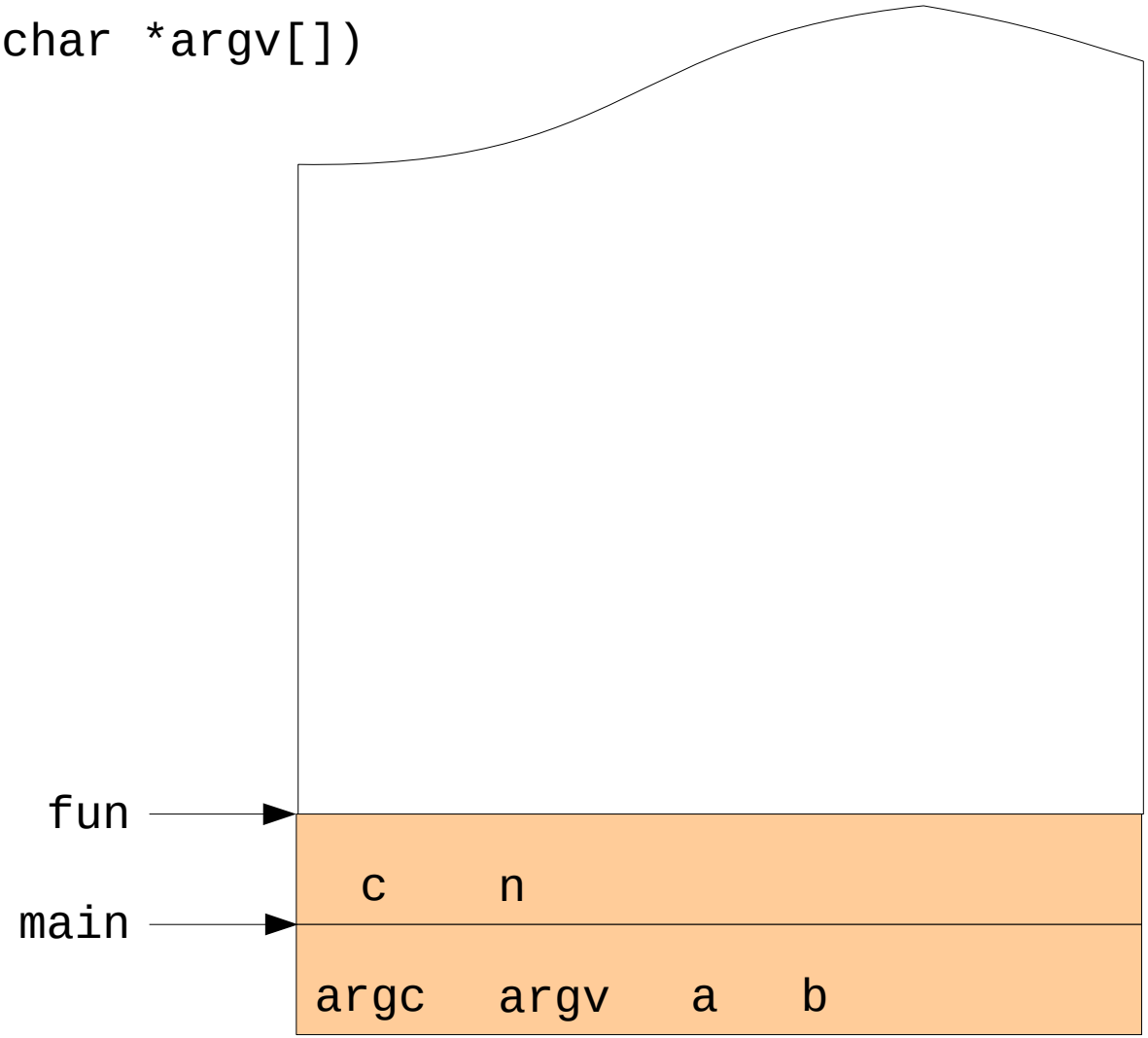


¿Dónde viven las variables? (III)

- Las variables register no 'tienen' dirección de memoria.
  - ➔No se pueden obtener punteros a ellas.

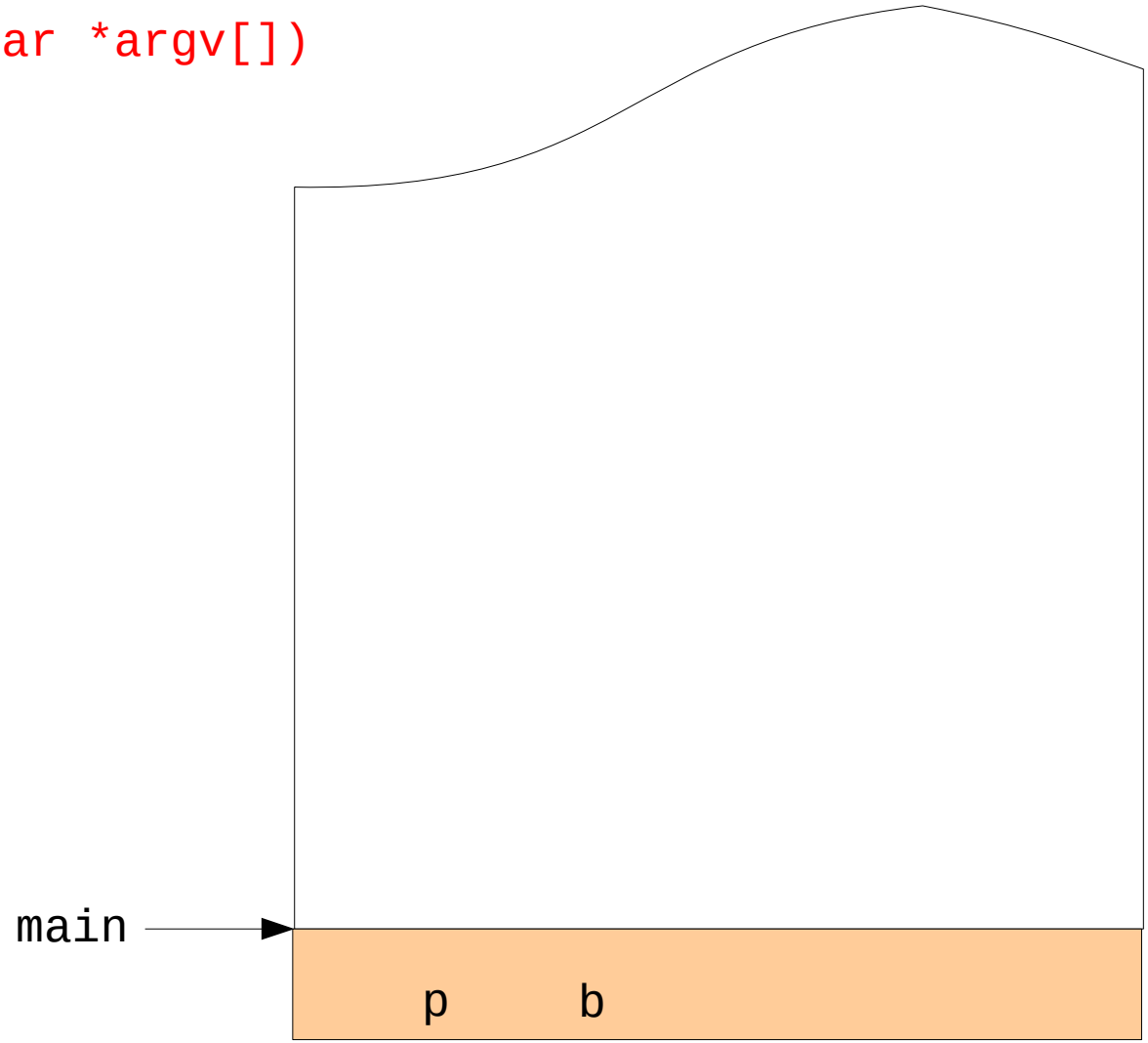
Punteros a variables locales (I)

```
int main(int argc, char *argv[])
{
    int a = 23;
    int b;
    b = fun(a);

    return b;
}

int fun(int n)
{
    int c = n;
    return 2 * c;
}
```



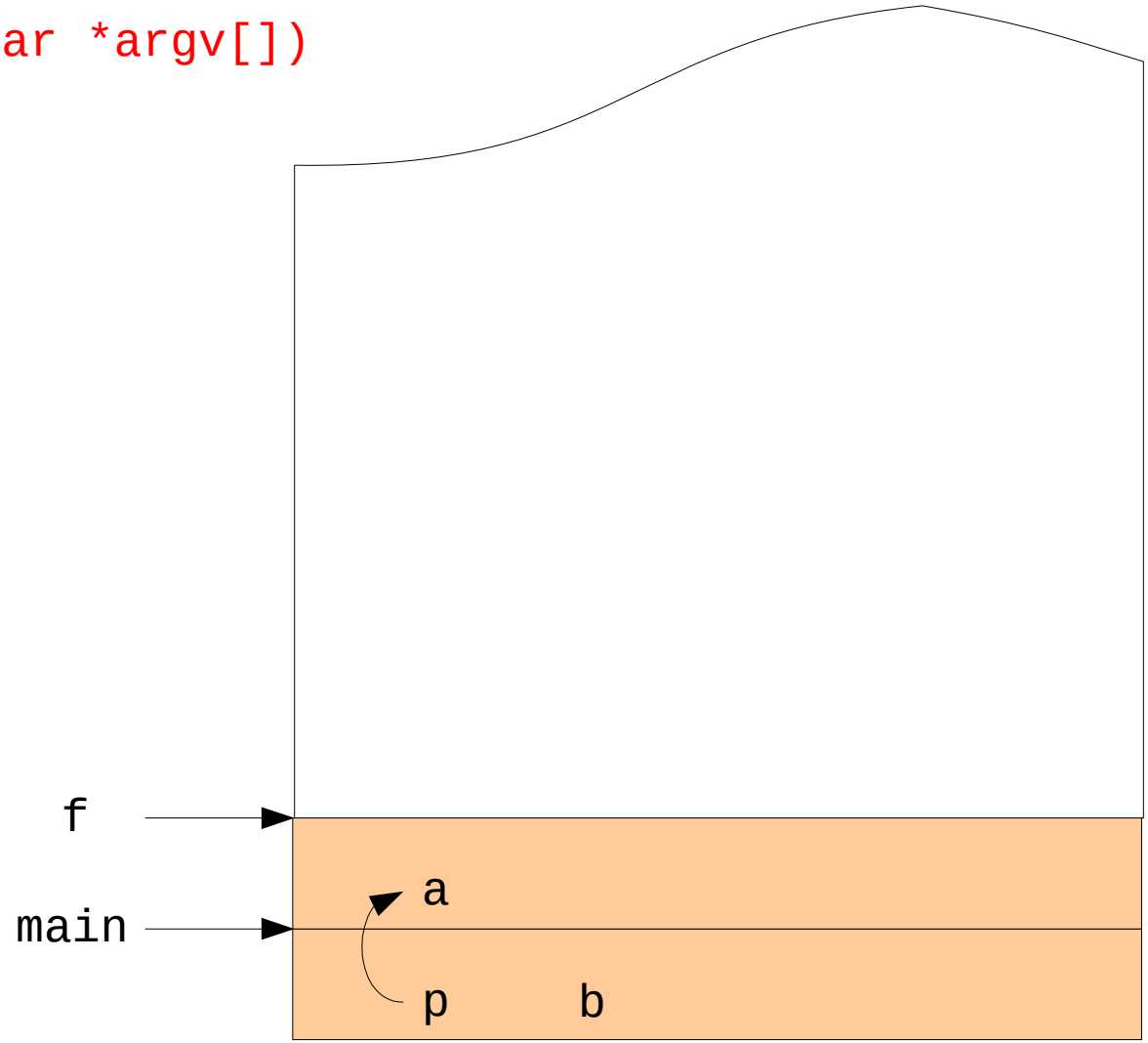
Punteros a variables locales (II)

```
int main(int argc, char *argv[])
{
    int *p;
    int b;
    p = f();
    b = *p;
    return b;
}

int *f(void)
{
    int a = 45;
    return &a;
}
```



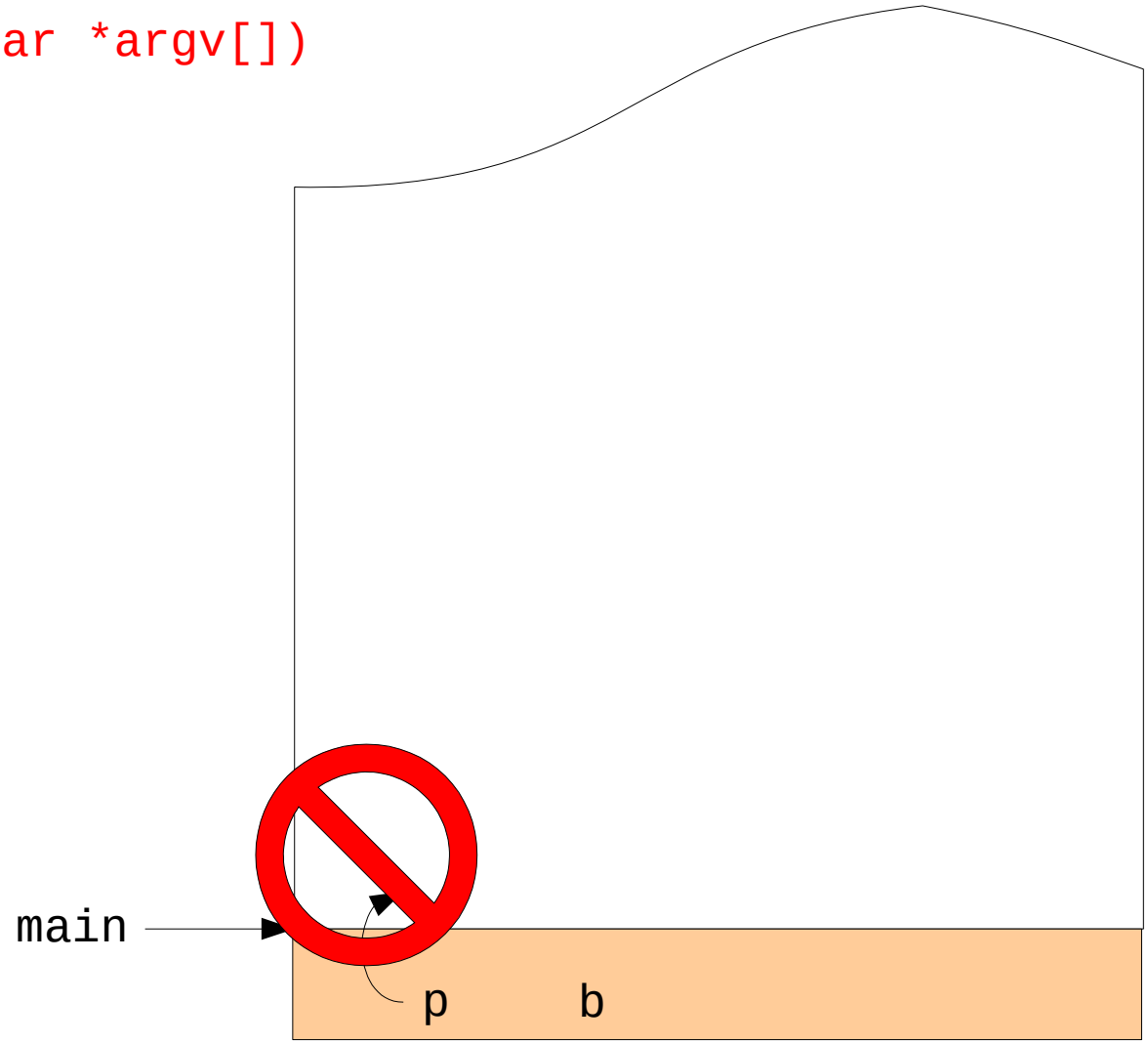
Punteros a variables locales (III)

```
int main(int argc, char *argv[])
{
    int *p;
    int b;
    p = f();
    b = *p;
    return b;
}

int *f(void)
{
    int a = 45;
    return &a;
}
```



Punteros a variables locales (IV)

```
int main(int argc, char *argv[])
{
    int *p;
    int b;
    p = f();
    b = *p;
    return b;
}

int *f(void)
{
    int a = 45;
    return &a;
}
```



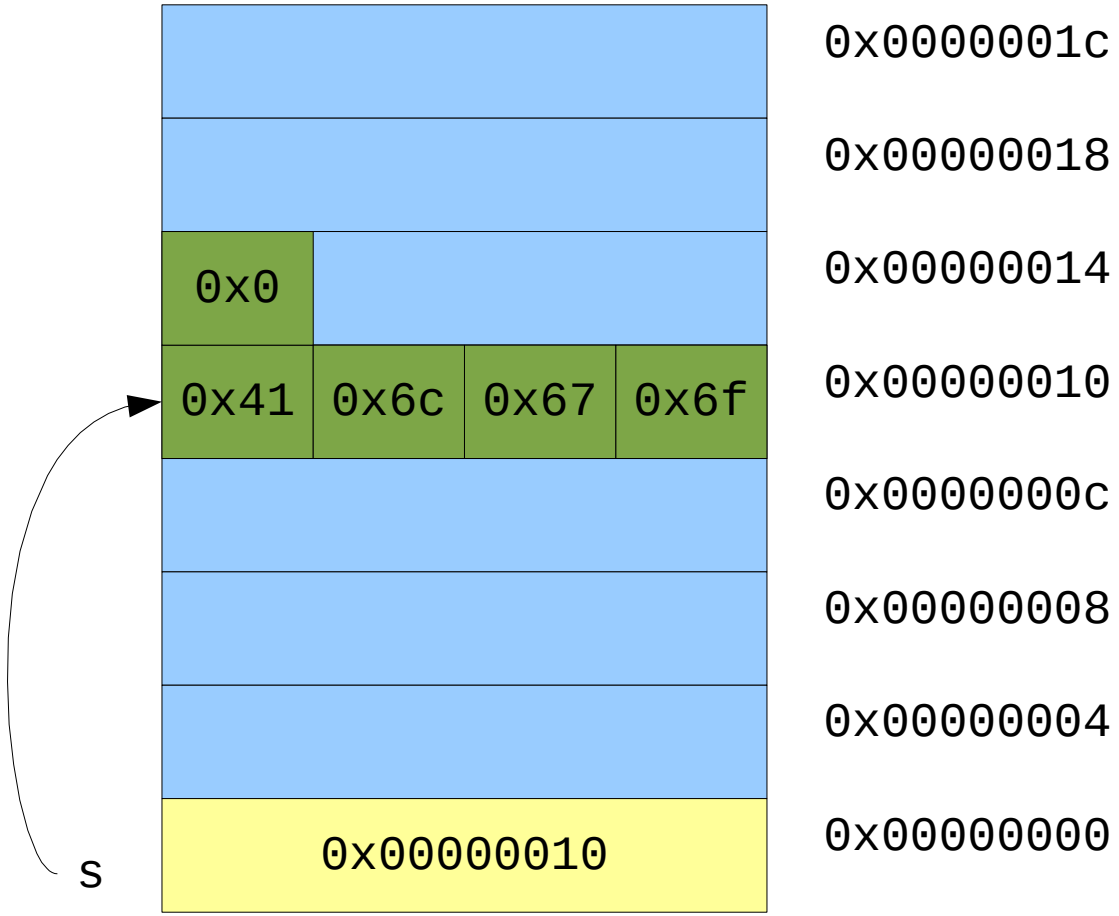
Cadenas de caracteres (I)

- No existen
- Pero se pueden 'simular'
- Puntero a una zona de memoria con caracteres acabada en el byte 0.

```
char *cadena;
```

Cadenas de carcacteres (II)

```
char *s = “Algo”;
```



Const y otras hierbas

- 'const' permite hacer las interfaces algo más seguras.
- No tiene por qué generar mejor código.

```
const int n = 3; /* El valor de n no cambia */
```

Declaraciones 'raras'

```
char *p;

const char *p;

char *const p;

const char *const p;
```

Punteros a funciones

```
int (*compar)(const void*, const void*);
```

```
char *(*fun)(int *, const struct *const punto);
```

Conversiones entre punteros

T* → void* → T*

T* → void* ⇏ P*

Memoria dinámica

- Esta memoria vive en la sección heap
  - ➔El heap se encuentra tras la zona .data
- Si no se libera → ¡ FUGA !

```
char *p = malloc(n_bytes);

/* ... código que usa p ... */

free(p);
```

Segmentation Fault más corto

- Programa que provoca un segfault y ocupa SOLO 5 bytes

Segmentation Fault más corto

- Programa que provoca un segfault y ocupa SOLO 5 bytes

```
main;
```

Bibliografía

- The C Programming Language – Brian Kernighan & Dennis Ritchie
- Mastering Algorithms with C – Kyle Loudon

¿ Preguntas ?

¿?

Muchas gracias por venir