

Sesión 06: Punteros

Programación para Sistemas

Ángel Herranz

2020-2021

Universidad Politécnica de Madrid

Recordatorio *arrays*

- Variable **global** o **local**, se necesita la longitud:

```
T a[N];  
T a[] = { e0, e1, ..., en-1 };
```

- Asignación prohibida:

```
a = b ❌
```

- Longitud: `sizeof(a) / sizeof(a[0])`
- **Argumento**, no se necesita la longitud (C la ignora)

```
void f(T a[]) { ... }
```

- Por convención se pasa la longitud como argumento:

```
void f(T a[], size_t n) { ... }
```

Recordatorio *strings*

- C no tiene *strings*
- Los *strings* en C son **arrays de char**

```
char s[] = "mundo";
```

- Por **convención**: los *strings* son **NULL-terminated**

```
char s[] = {'m', 'u', 'n', 'd', 'o', '\0'};
```

- **(sizeof(s) / sizeof(s[0])) == 6**

▶▶ La forma habitual de escribir el tipo es

```
[[char *s]] = [[char s[]]]
```

En el capítulo de hoy...

*p

&x

Direcciones de memoria

- C permite un **control absoluto** de la memoria
- Nueva **sintaxis**:

$$\begin{aligned} \langle expr \rangle & ::= \dots \\ & | \text{'\&'} \langle expr \rangle \\ & | \dots \end{aligned}$$

- Su **semántica**:

$$[[\&e]] = \text{«dirección de memoria de la expresión } e\text{»}$$

- Usaremos el *conversion specifier* **%p** de printf para mostrar **direcciones de memoria**

¿Dónde está la variable?

```
int x = 42;  
printf("El contenido de x es %i\n", x);  
printf("La dirección de memoria de x es %p\n", &x);
```

¿Donde está la variable?

```
int x = 42;  
printf("El contenido de x es %i\n", x);  
printf("La dirección de memoria de x es %p\n", &x);
```

El contenido de x es 42

La dirección de memoria de x es 0x7ffc4e20391c

dir.c: exploremos la memoria i ⌚ 10'

```
#include <stdio.h>
int global1;
int global2;

void f (int arg) {
    int local;
    printf("f(%i): &arg: %p\n",
           arg, &arg);
    printf("f(%i): &local: %p\n",
           arg, &local);
    if (arg) f(!arg);
}
```

```
int main() {
    int local;
    printf("main: &local: %p\n", &local);
    printf("main: &global1: %p\n",
           &global1);
    printf("main: &global2: %p\n",
           &global2);
    printf("main: &f: %p\n", &f);
    printf("main: &main: %p\n", &main);
    f(1);
    return 0;
}
```

dir.c: exploremos la memoria ii

- ¿Puedes ver dónde están las variables globales?
- ¿Puedes ver lo que ocupan?
- ¿Puedes ver cómo se distribuyen las variables y argumentos en el *stack*?
- ¿Has observado que las funciones son variables globales?
- 🏠 Añade más variables locales y argumentos

Punteros: variables con direcciones de memoria

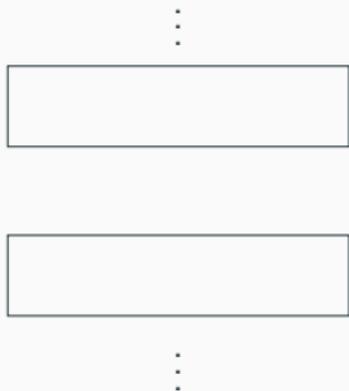
- Sintaxis para declarar punteros:

$$T *p;$$

- p es una variable que **contiene una dirección de memoria**,
- en la que hay **un elemento de tipo T**
- **accesible usando la expresión**

$$*p$$
$$\langle expr \rangle ::= \dots$$
$$| \quad * \langle expr \rangle$$
$$| \quad \dots$$
$$[[*e]] = \text{«contenido de la dirección de memoria } [[e]] \text{»}$$

Encajando las piezas i



Encajando las piezas i

```
int x;
```



Encajando las piezas i

⋮
x:

-243291612

 0x7fff15fb17d0

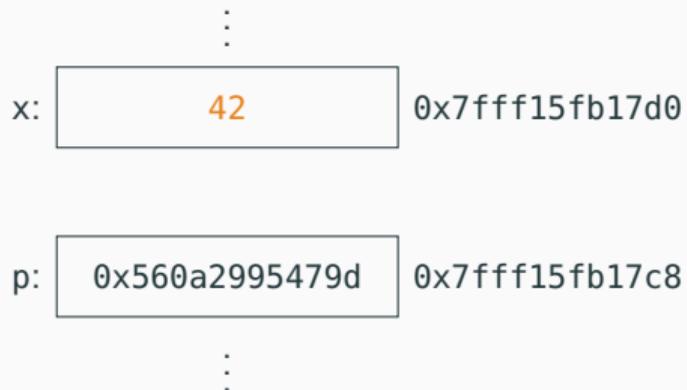
p:

0x560a2995479d

 0x7fff15fb17c8
⋮

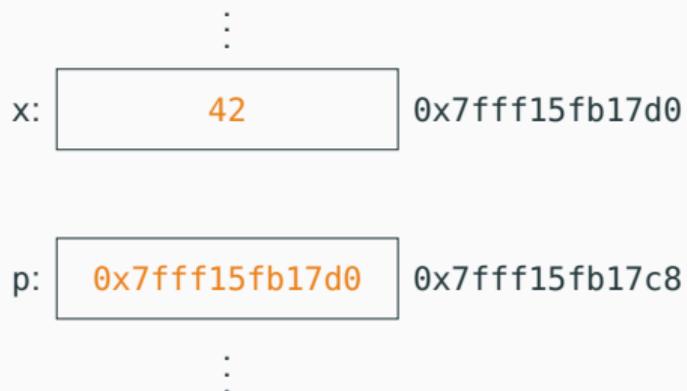
```
int x;  
int *p;
```

Encajando las piezas i



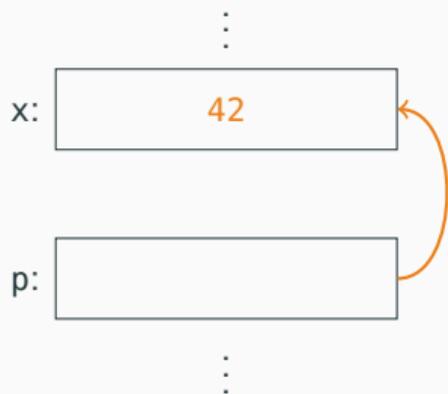
```
int x;  
int *p;  
x = 42;
```

Encajando las piezas i



```
int x;  
int *p;  
x = 42;  
p = &x;
```

Encajando las piezas i



```
int x;
```

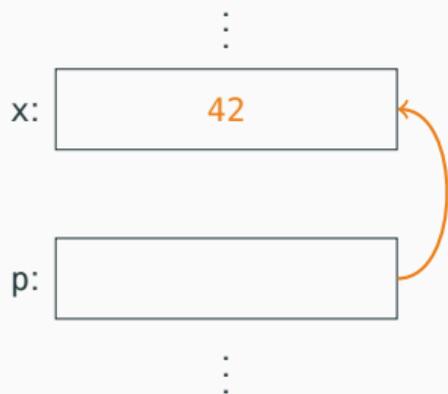
```
int *p;
```

```
x = 42;
```

```
p = &x;
```

Representación habitual

Encajando las piezas i



```
int x;
```

```
int *p;
```

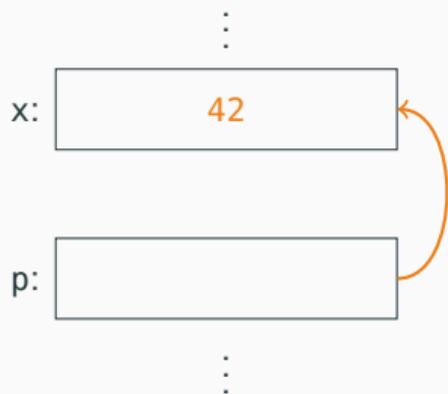
```
x = 42;
```

```
p = &x;
```

Representación habitual

```
printf("%i\n", *p)
```

Encajando las piezas i



```
int x;
```

```
int *p;
```

```
x = 42;
```

```
p = &x;
```

Representación habitual

```
printf("%i\n", *p)
```

42

Encajando las piezas ii

 ¿Qué hacen estas dos líneas después del código anterior?

```
*p = 27;  
printf("%i\n", x);
```

Encajando las piezas ii

 ¿Qué hacen estas dos líneas después del código anterior?

```
*p = 27;  
printf("%i\n", x);
```

 Entender estas últimas transparencias es **muy importante**

Función que intercambie dos enteros

```
int x = 42, y = 27;  
printf("Antes de intercambiar: (%i, %i)\n", x, y);  
intercambiar(x,y);  
printf("Despues de intercambiar: (%i, %i)\n", x, y);
```

Lo esperado:

Antes de intercambiar: (42, 27)

Despues de intercambiar: (27, 42)

intercambiar: primer intento ⌚ 5

```
void intercambiar(int x, int y) {  
    int aux = x;  
    x = y;  
    y = aux;  
}
```

intercambiar: primer intento ⌚ 5

```
void intercambiar(int x, int y) {  
    int aux = x;  
    x = y;  
    y = aux;  
}
```

 ¿Qué ocurre? (¡dibujémoslo en cajas!)

intercambiar: primer intento ⌚ 5

```
void intercambiar(int x, int y) {  
    int aux = x;  
    x = y;  
    y = aux;  
}
```

 ¿Qué ocurre? (¡dibujémoslo en cajas!)

 **Paso por valor:** el contenido de las variables se **copia** en los argumentos

 ¿Y si pasamos los **punteros como argumento**? ⌚ 5'+

Además, en el capítulo de hoy...

`*p`

`a[0]`

Estrecha relación entre punteros y arrays i

```
int *p;  
int a[] = ...;
```

	⋮	
p:	0x560a2995479d	0x7fff15fb17c8
a:	2	0x7fff15fb17d0
a[1]:	3	0x7fff15fb17d4
a[2]:	5	0x7fff15fb17d8
a[3]:	7	0x7fff15fb17dc
	⋮	

Estrecha relación entre punteros y arrays i

```
int *p;  
int a[] = ...;  
p = a;
```

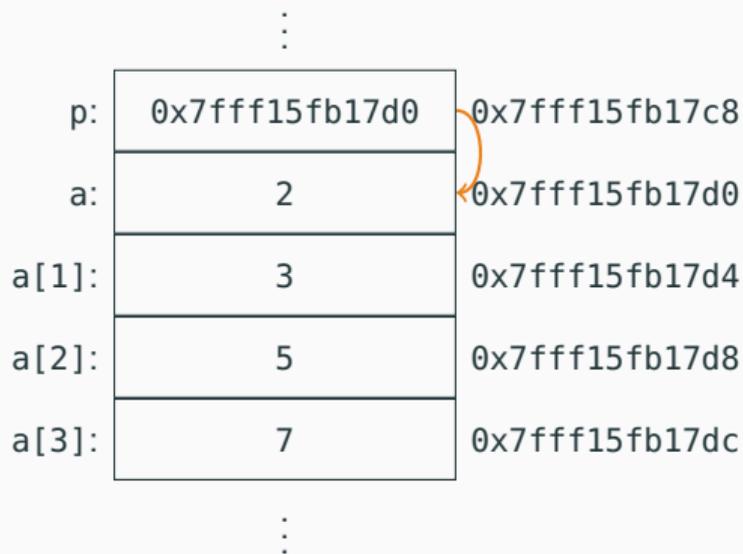
⋮

p:	0x7fff15fb17d0	0x7fff15fb17c8
a:	2	0x7fff15fb17d0
a[1]:	3	0x7fff15fb17d4
a[2]:	5	0x7fff15fb17d8
a[3]:	7	0x7fff15fb17dc

⋮

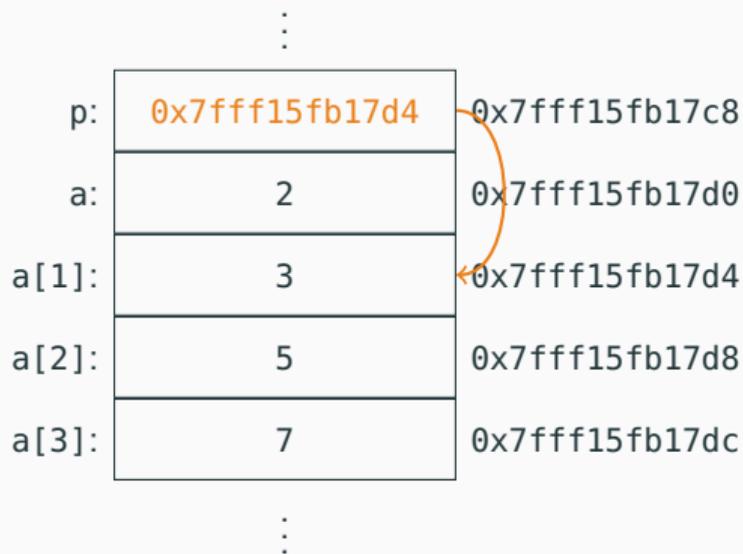
Estrecha relación entre punteros y arrays i

```
int *p;  
int a[] = ...;  
p = a;  
assert(*p == a[0]);
```



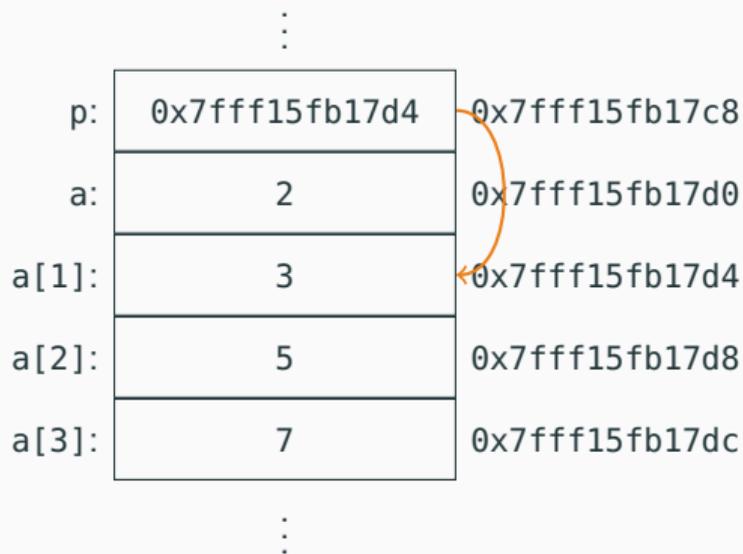
Estrecha relación entre punteros y arrays i

```
int *p;  
int a[] = ...;  
p = a;  
assert(*p == a[0]);  
p = p + 1;
```



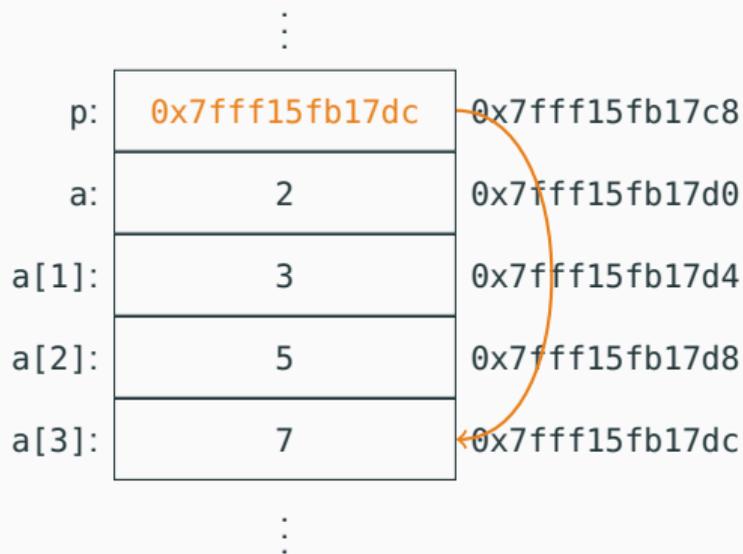
Estrecha relación entre punteros y arrays i

```
int *p;  
int a[] = ...;  
p = a;  
assert(*p == a[0]);  
p = p + 1;  
assert(*p == a[1]);
```



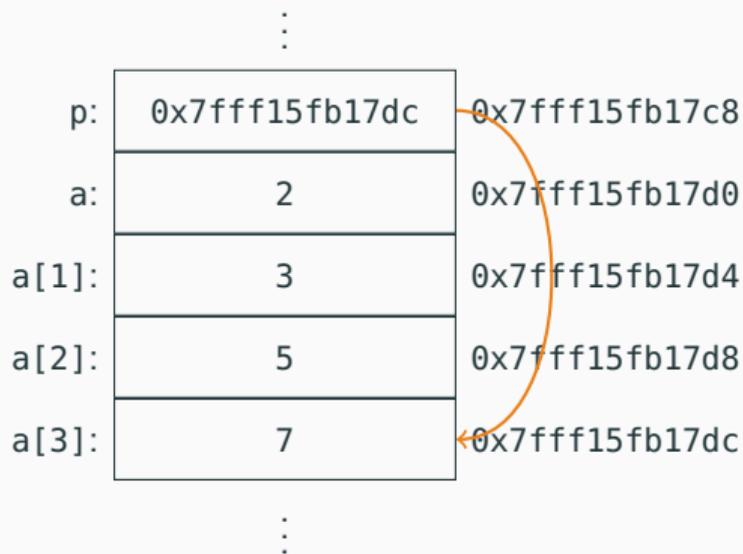
Estrecha relación entre punteros y arrays i

```
int *p;  
int a[] = ...;  
p = a;  
assert(*p == a[0]);  
p = p + 1;  
assert(*p == a[1]);  
p = p + 2;
```



Estrecha relación entre punteros y arrays i

```
int *p;  
int a[] = ...;  
p = a;  
assert(*p == a[0]);  
p = p + 1;  
assert(*p == a[1]);  
p = p + 2;  
assert(*p == a[3]);
```



Aritmética de punteros

```
int *p;
```

```
long long int *q;
```



Aritmética de punteros

```
int *p;
```

```
long long int *q;
```

```
p = (int *)q;
```



Aritmética de punteros

```
int *p;  
long long int *q;  
p = (int *)q;  
p++;
```



Aritmética de punteros

```
int *p;  
long long int *q;  
p = (int *)q;  
p++;  
q++;
```



Aritmética de punteros

```
int *p;
```

```
long long int *q;
```

```
p = (int *)q;
```

```
p++;
```

```
q++;
```

 Imprimir los valores de los punteros ⌚ 5'

```
⋮  
p: 0x7fff15fb17d4  
q: 0x7fff15fb17d8  
⋮
```

Aritmética de punteros

```
int *p;
```

```
long long int *q;
```

```
p = (int *)q;
```

```
p++;
```

```
q++;
```

 Imprimir los valores de los punteros ⌚ 5'

 ¿Ves la diferencia? ¿A qué se debe?

	⋮
p:	0x7fff15fb17d4
q:	0x7fff15fb17d8
	⋮

Estrecha relación entre punteros y arrays ii

- Asumiendo el siguiente contexto...

T a[] = ...;

T *p = a;

- Tenemos las siguientes verdades

Estrecha relación entre punteros y arrays ii

- Asumiendo el siguiente contexto...

T a[] = ...;

T *p = a;

- Tenemos las siguientes verdades

$$[[p]] = [a]$$

$$[[p]] = [&a[0]]$$

$$[[*p]] = [a[0]]$$

$$[[p+i]] = [&a[i]]$$

$$[[*(p+i)]] = [a[i]]$$

Alta densidad

La densidad de información en las transparencias anteriores es enorme pero...

La densidad de información en las transparencias anteriores es enorme pero...

es imposible programar en C si no las entiendes

La densidad de información en las transparencias anteriores es enorme pero...

es imposible programar en C si no las entiendes

 hoja de ejercicios