

Sesión 06: Punteros (incompletas)

Programación para Sistemas

Ángel Herranz

Otoño 2018

Universidad Politécnica de Madrid

En capítulos anteriores. . .

Sesión 0: Presentación

Sesión 1: Contacto C

Sesión 2: Ejecutando C

Sesión 3: Tipos básicos

Sesión 4: Módulos

Sesión 5: Arrays y Strings

En el capítulo de hoy...

$*p$

$\&x$

Direcciones de memoria

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

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

- Su **semántica**:

$\llbracket \&e \rrbracket = \text{«dirección de memoria de la expresión } e\text{»}$

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

¿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);
```

¿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);
```

El contenido de x es 42

La dirección de memoria de x es 0x7ffc4e20391c

dir.c: exploremos la memoria i

```
#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 donde 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

Variables de tipo puntero

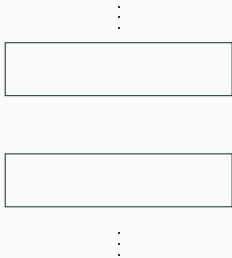
- Sintaxis:

$$T *p;$$

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

$$*p$$

Encajando las piezas i



Encajando las piezas i

`int x;`



Encajando las piezas i

```
int x;
```

```
int *p;
```

⋮

x: -243291612 0x7fff15fb17d0

p: 0x560a2995479d 0x7fff15fb17c8

⋮

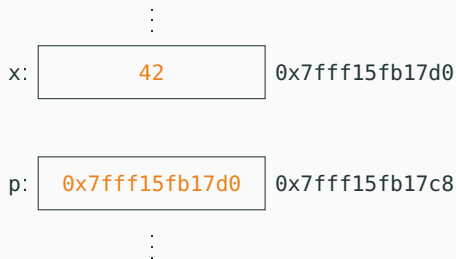
Encajando las piezas i

⋮
x: 42 0x7fff15fb17d0

p: 0x560a2995479d 0x7fff15fb17c8
⋮

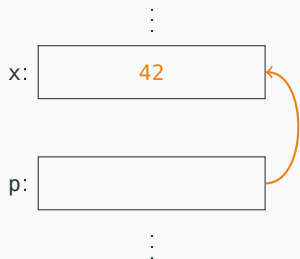
```
int x;  
int *p;  
x = 42;
```

Encajando las piezas i



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

Encajando las piezas i



```
int x;
```

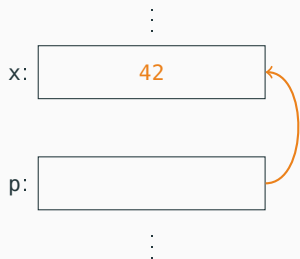
```
int *p;
```

```
x = 42;
```

```
p = &x;
```

Representación habitual

Encajando las piezas i



```
int x;
```

```
int *p;
```

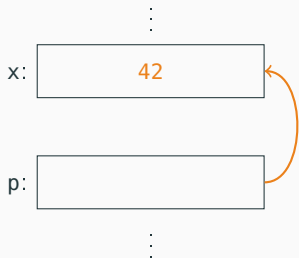
```
x = 42;
```

```
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

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

-  ¿Qué ocurre? (¡dibujémoslo en cajas!)

intercambiar: primer intento

```
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?

Además, en el capítulo de hoy. . .

A close-up photograph of two hands holding two interlocking puzzle pieces against a warm, golden-yellow background. The puzzle piece on the left is held by fingers from the left and has the text `*p` printed on it. The puzzle piece on the right is held by fingers from the right and has the text `a[0]` printed on it. The pieces are slightly offset, creating a gap between them.

`*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
	⋮	

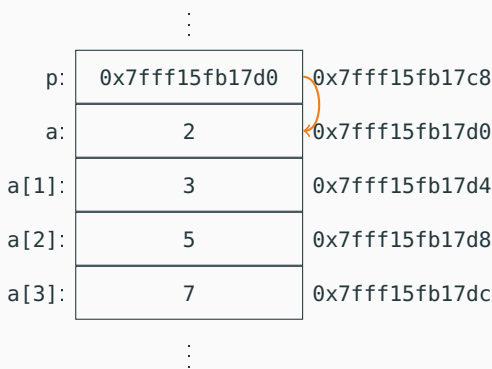
Estrecha relación entre punteros y arrays i

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

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

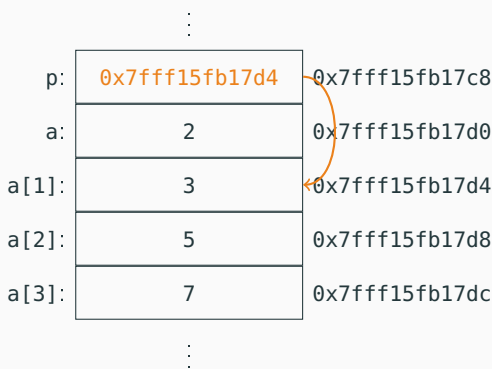
Estrecha relación entre punteros y arrays i

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



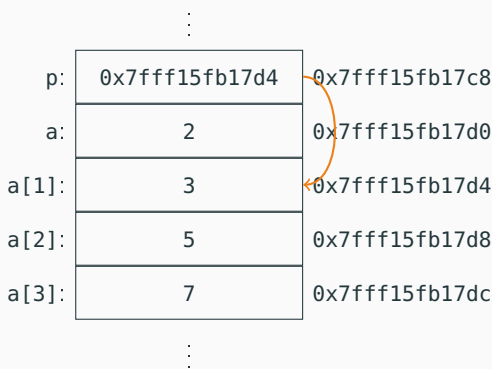
Estrecha relación entre punteros y arrays i

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



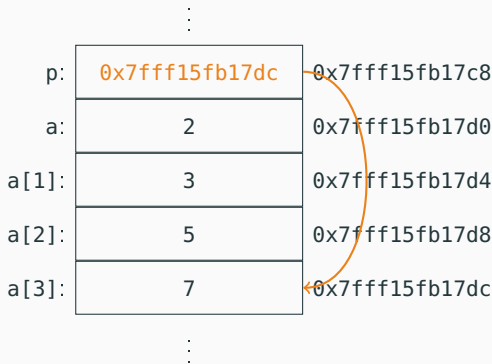
Estrecha relación entre punteros y arrays i

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



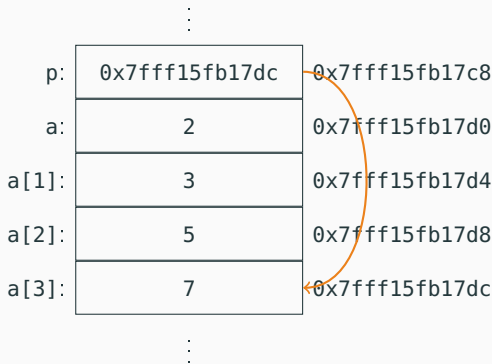
Estrecha relación entre punteros y arrays i

```
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

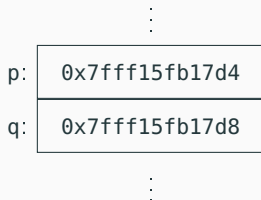
```
int *p;
```

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

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

```
p++;
```

```
q++;
```



💬 ¿Ves la diferencia?

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**

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

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

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

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

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

La densidad de información en las transparencias anteriores es enorme

Alta densidad

La densidad de información en las transparencias anteriores es enorme

Es imposible programar en C si no entiendes las transparencias anteriores

RPN (*Reverse Polish Notation*) i

The following algorithm evaluates postfix expressions using a stack, with the expression processed from left to right:

```
for each token in the postfix expression:
    if token is an operator:
        operand_2 = pop from the stack
        operand_1 = pop from the stack
        result = evaluate token with operand_1 and operand_2
        push result back onto the stack
    else if token is an operand:
        push token onto the stack
result = pop from the stack
```

Reverse Polish notation (Wikipedia)

- Los *tokens* serán floats, operadores ('+', '-', '*', '/') y el caracter de *fin de expresión* ('=')
- Leemos la expresión de la entrada estándar con scanf:

```
float operando;  
char operador[2];  
scanf("%f", &operando);  
scanf("%c", operador);
```

man 3 scanf

- Utilizaremos un array para implementar la pila de operandos
- Asumiremos que la pila no puede crecer en más de 1000 elementos


Así deberá comportarse nuestro programa:

```
$ ./rpn
```

```
15 7 1 1 + - / 3 * 2 1 1 + + - =
```

```
5
```

```
$ |
```

-  Modifica tu programa `rpn` para acceder a la pila de operandos utilizando punteros