

Sesión 10: Structs

Programación para Sistemas

Ángel Herranz

Curso 2024-2025

Recordatorio: malloc y free

- ¡Dame **más memoria** (para *arrays*)!

```
int *enteros = (int *) malloc(N * sizeof(int));
```

```
char *s = (char *) malloc(N * sizeof(char));
```

```
double *reales = (double *) malloc(N * sizeof(double));
```

- ¡Ya **no la necesito más**!

```
free(enteros);
```

```
free(s);
```

```
free(reales);
```

- **malloc** en C: es *como new* en Java
- **free** en C: **no existe** en Java porque **en Java es automático**

Recordatorio: malloc y free

- ¡Dame **más memoria** (para *arrays*)!

```
int *enteros = (int *) calloc(N, sizeof(int));  
char *s = (char *) calloc(N, sizeof(char));  
double *reales = (double *) calloc(N, sizeof(double));
```

- ¡Ya **no la necesito más**!

```
free(enteros);  
free(s);  
free(reales);
```

- **malloc** en C: es *como new* en Java
- **free** en C: **no existe** en Java porque **en Java es automático**

Recordatorio: *malloc*'s friends (man 3 malloc)

MALLOC(3)

Linux Programmer's Manual

MALLOC(3)

NAME

`malloc`, `free`, `calloc`, `realloc` - allocate and free dynamic memory

SYNOPSIS

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

```
void *malloc(size_t size);
```

```
void free(void *ptr);
```

```
void *calloc(size_t nmemb, size_t size);
```

```
void *realloc(void *ptr, size_t size);
```

```
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

DESCRIPTION

The `malloc()` function allocates `size` bytes and returns a
...

Recordatorio: aritmética de punteros i

```
int *p;
```

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

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

Recordatorio: aritmética de punteros i

```
int *p;
```

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

```
p = a;
```

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

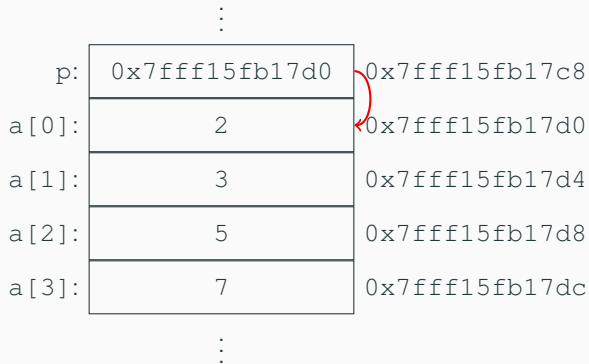
Recordatorio: aritmética de punteros i

```
int *p;
```

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

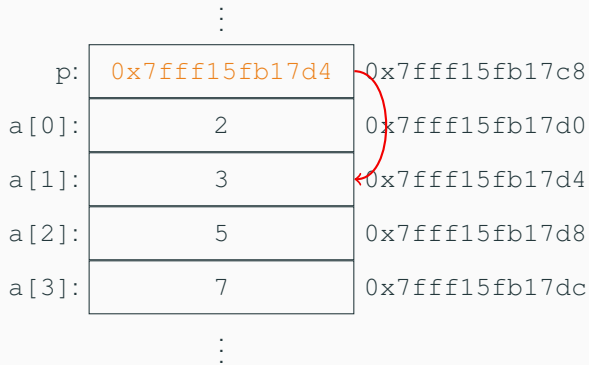
```
p = a;
```

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



Recordatorio: aritmética de punteros i

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



Recordatorio: aritmética de punteros i

```
int *p;
```

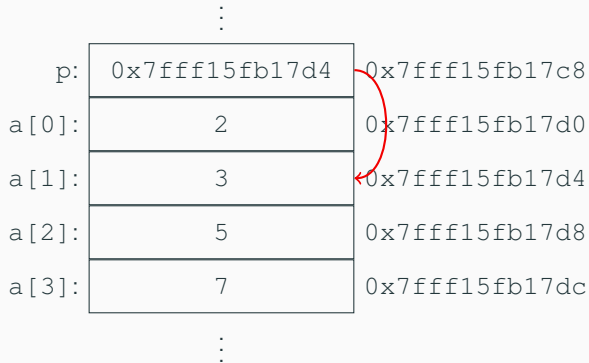
```
int a[] = ...;
```

```
p = a;
```

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

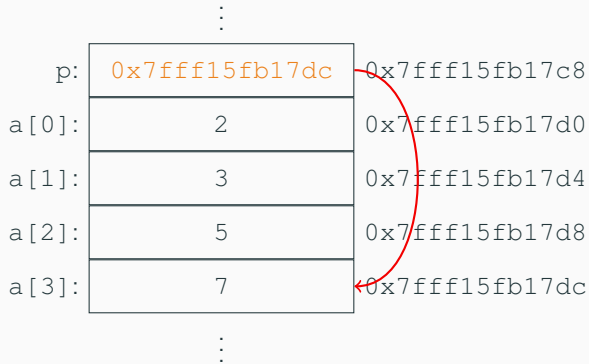
```
p = p + 1;
```

```
assert(*p == a[1]);
```



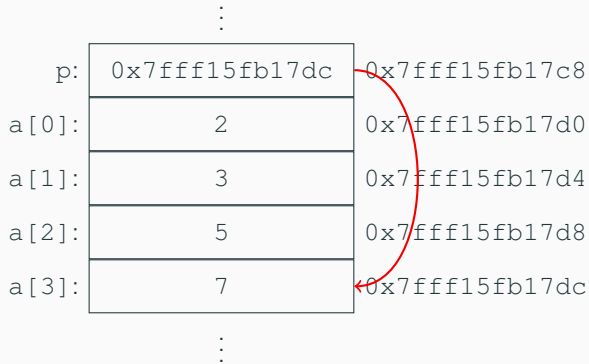
Recordatorio: aritmética de punteros i

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



Recordatorio: aritmética de punteros 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]);
```



Recordatorio: aritmética de punteros ii

```
int *p;
```

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

⋮

p:	0x560a2995479d
q:	0x7fff15fb17d0

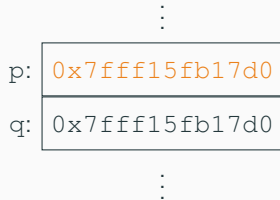
⋮

Recordatorio: aritmética de punteros ii

```
int *p;
```

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

```
p = (int *) q;    ← sólo didáctico!
```



Recordatorio: aritmética de punteros ii

```
int *p;  
long long int *q;  
p = (int *) q;      ← sólo didáctico!  
p++;
```

⋮

p:	0x7fff15fb17d4
q:	0x7fff15fb17d0

⋮

Recordatorio: aritmética de punteros ii

```
int *p;  
long long int *q;  
p = (int *) q;      ← sólo didáctico!  
p++;  
q++;
```

⋮

p:	0x7fff15fb17d4
q:	0x7fff15fb17d8

⋮

Recordatorio: relación punteros-array

- Asumiendo el siguiente contexto...

T `a[] = ...;`

T `*p = a;`

- Tenemos las siguientes verdades

Recordatorio: relación punteros-array

- Asumiendo el siguiente contexto...

$T \ a[] = \dots;$

$T \ *p = a;$

- Tenemos las siguientes verdades

$$p \equiv a$$

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

$$*p \equiv a[0]$$

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

$$*(p+i) \equiv a[i]$$

Memory leaks

Cuando se nos olvida liberar memoria

Memory leaks

Cuando se nos olvida liberar memoria

Segmentation fault

Cuando se nos olvida solicitar memoria

Memory leaks

Cuando se nos olvida liberar memoria

Segmentation fault

Cuando se nos olvida solicitar memoria
o la usamos más allá de la solicitada

Memory leaks

Cuando se nos olvida liberar memoria

Segmentation fault

Cuando se nos olvida solicitar memoria
o la usamos más allá de la solicitada

Comportamiento indefinido

Cuando liberamos memoria no solicitada

En el capítulo de hoy...

- *Structs*
- Cadenas enlazadas

Structs

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called “records” in some languages, notably Pascal.) [...]

Capítulo 6, K&R

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called “records” in some languages, notably Pascal.) [...]

Capítulo 6, K&R

Simplificando: *como* objetos sin métodos

struct ii

- Empezamos creando una variable para representar un punto en coordenadas cartesianas enteras

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

- El código anterior declara la variable `a`,
- como un registro (*struct*),
- con dos atributos (*members*) `x` e `y` de tipo entero,
- accesibles con la sintaxis `a.x` y `a.y`

Sintaxis popular

 Escribe un programa con dos structs a y b

```
struct {  
    int x;  
    int y;  
} a, b;
```

y **explora** la sintaxis de **struct**

 5'

- Ideas:

```
a.x = 1;  
printf("x == %i\n", a.x);  
sizeof(a)  
b = a;
```

struct iii

- Si observas con detalle verás que la frase

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

se puede considerar como **un nuevo tipo**

- Para escribir menos se puede declarar una **etiqueta (tag)** para el *struct* de esta forma:

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

- Ahora la **etiqueta punto_s** nos permite declarar variables así:

```
struct punto_s a, b; /* "struct punto_s" es un tipo */
```

struct iv

- Por supuesto, es posible declarar **structs de structs**, **arrays de structs** y **punteros de structs**

```
struct rectangulo_s {  
    struct punto_s so;  
    struct punto_s ne;  
};
```

```
struct rectangulo_s r; // r es un "struct rectangulo"  
struct punto_s h[6]; // h es un array de "struct punto"  
struct punto_s *p; // p es un puntero a "struct punto"
```

Punteros a struct

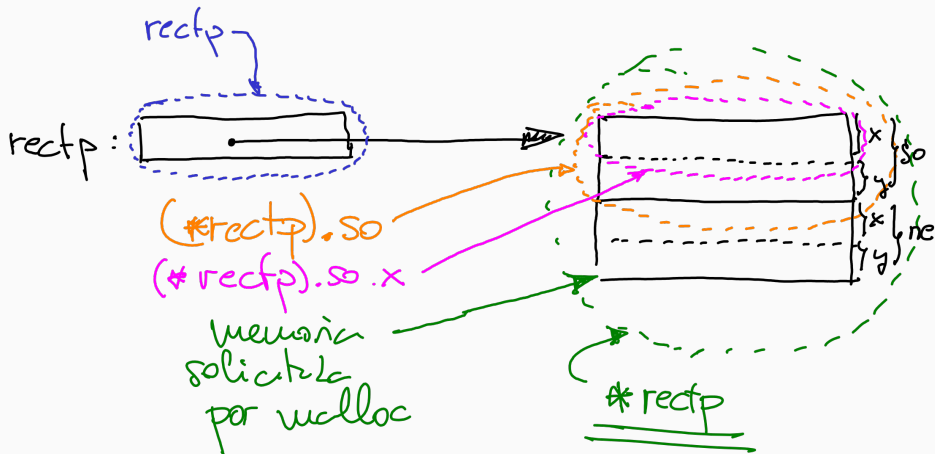
```
struct rectangulo_s *rectp;  
rectp = (struct rectangulo_s *)  
        malloc(sizeof(struct rectangulo_s));
```

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

```
struct rectangulo_s {  
    struct punto_s so;  
    struct punto_s ne;  
};
```

Solución

- Deberías haber dibujado algo parecido a esto:



¿Qué significa `(*rectp) .so`?

La flecha: →

¿Qué significa `(*rectp).so`?

¿Por qué no `*rectp.so`?

¿Qué significa `(*rectp).so`?

¿Por qué no `*rectp.so`?

C pone los paréntesis que faltan en `*rectp.so`
donde no queremos:

`*(rectp.so)`

¿Qué significa `(*rectp).so`?

¿Por qué no `*rectp.so`?

C pone los paréntesis que faltan en `*rectp.so`
donde no queremos:

`*(rectp.so)`

`(*rectp).so = rectp->so`

¿Qué significa `(*rectp).so`?

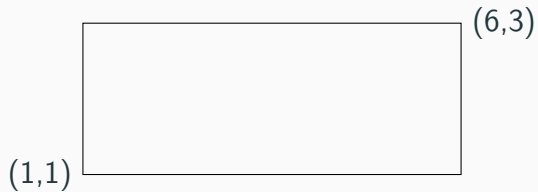
¿Por qué no `*rectp.so`?

C pone los paréntesis que faltan en `*rectp.so`
donde no queremos:

`*(rectp.so)`

`(*rectp).so = rectp->so`

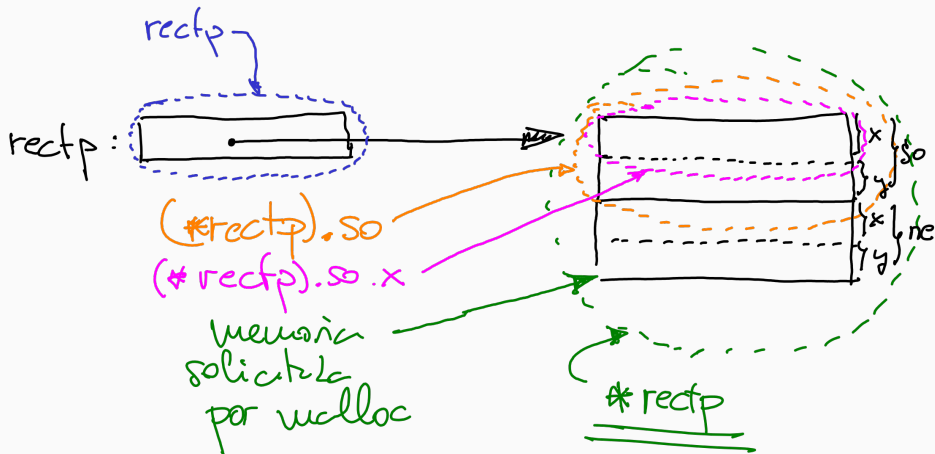
Almacena este rectángulo en rectp



 5'

Solución

- Deberías haber dibujado algo parecido a esto:



Punteros a structs como arrays

- Cuando queremos un array de *structs* reservamos *n* espacios del tamaño del *struct*:

```
size_t n = 1000000;  
struct rectangulo_s *rects;  
rects = calloc(n, sizeof(struct rectangulo_s));
```

- Ahora *rects* apunta al primer *struct* de un array con el que podríamos trabajar:

```
for (i = 0; i < n; i++) {  
    /* Ej. leer los rectángulos de stdin */  
    leer_rect(stdin, &rects[i]);  
}
```

typedef: definiendo nuevos tipos

- Podemos definir nuevos tipos con **typedef**
- Ejemplo

```
typedef long long unsigned int natural;
```

typedef: definiendo nuevos tipos

- Podemos definir nuevos tipos con **typedef**
- Ejemplo

```
typedef long long unsigned int natural;
```

- Funciona igual que la definición de una variable,
- pero define un nuevo tipo, **natural**, que es igual a `long long unsigned int`

 Por convención, algunos desarrolladores usan el sufijo `_t`

```
typedef long long unsigned int natural_t;
```

```
// Ejemplo de decl de variables de tipo natural_t:  
natural_t n, m;
```

typedef de structs

💬 ¿Alguna idea para evitar tener que declarar variables así?

```
struct punto_s a, b;
```

typedef de structs

💬 ¿Alguna idea para evitar tener que declarar variables así?

```
struct punto_s a, b;
```

- Usando **typedef** podríamos hacer esto

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

```
typedef struct punto_s punto_t;
```

incluso no sería necesario declarar la etiqueta:

```
typedef struct {int x; int y} punto_t;
```

- Podríamos declarar variables de una forma más **legible**:

```
punto_t a, b;
```

```
rectangulo_t r, s;
```

Punteros a struct: uso masivo en C

```
$ man fopen
```

```
FOPEN(3)
```

```
Linux Programmer's Manual
```

```
FOPEN(3)
```

```
NAME
```

```
fopen, fdopen, freopen - stream open functions
```

```
SYNOPSIS
```

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

```
FILE *fopen(const char *pathname, const char *mode);
```

```
...
```



Interpreta esas líneas de la página del manual:

FILE es internamente un tipo *struct*

Aunque los usamos como tipos abstractos

```
FILE *fd = fopen("/etc/password", O_RDONLY);  
char linea[2050];  
while (fgets(linea, 2049, fd)) {  
    /* hacer algo con linea */  
}
```

`fopens` y `fgets` forman parte del API de `FILE`

Cadenas enlazadas

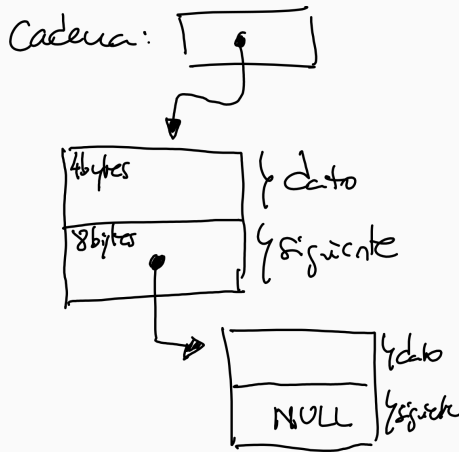
Cadenas enlazadas: el tipo

```
struct nodo_s {  
    int dato;  
    struct nodo_s *siguiente;  
};
```

Cadenas enlazadas: el tipo

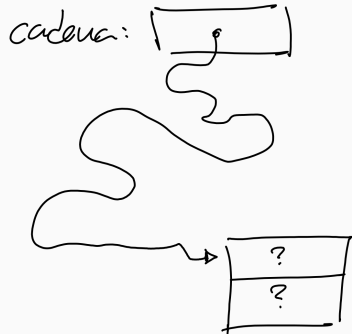
```
struct nodo_s {  
    int dato;  
    struct nodo_s *siguiente;  
};
```

```
struct nodo_s *cadena;
```

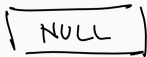


Cadenas enlazadas: vacía

```
struct nodo_s *cadena;
```



Cadenas enlazadas: vacía

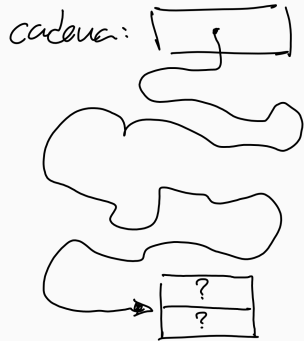
cadena: 

```
struct nodo_s *cadena;
```

```
cadena = NULL;
```

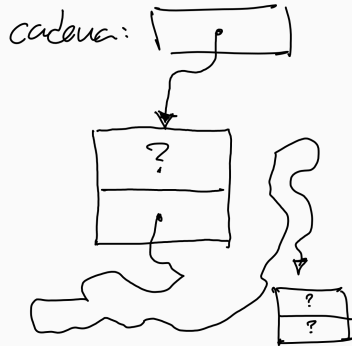
Cadenas enlazadas: un elemento

```
struct nodo_s *cadena;
```



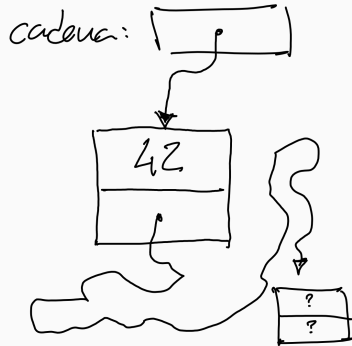
Cadenas enlazadas: un elemento

```
struct nodo_s *cadena;  
cadena =  
    (struct nodo_s *)  
    malloc(sizeof(struct nodo_s));
```



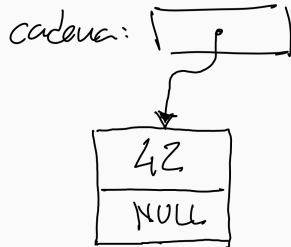
Cadenas enlazadas: un elemento

```
struct nodo_s *cadena;  
cadena =  
    (struct nodo_s *)  
    malloc(sizeof(struct nodo_s));  
cadena->dato = 42;
```



Cadenas enlazadas: un elemento

```
struct nodo_s *cadena;  
cadena =  
    (struct nodo_s *)  
    malloc(sizeof(struct nodo_s));  
cadena->dato = 42;  
cadena->siguiente = NULL;
```



Cadenas enlazadas: primero y último

- Expresión que representa el primero:

`cadena->dato`

- Recorrido hasta el último:

```
struct nodo_s *ultimo;  
ultimo = cadena;  
while (ultimo->siguiente != NULL) {  
    ultimo = ultimo->siguiente;  
}
```

 Dibujar

Cadenas enlazadas: añadir al principio

```
struct nodo_s *primero;  
primero = (struct nodo_s*)malloc(sizeof(struct nodo_s));  
primero->dato = nuevo;  
primero->siguiente = cadena;  
cadena = primero;
```

 Dibujar

 5'

Cadenas enlazadas: añadir al final

```
ultimo = cadena;
while (ultimo->siguiente != NULL) {
    ultimo = ultimo->siguiente;
}
ultimo->siguiente =
    (struct nodo_s*)malloc(sizeof(struct nodo_s));
ultimo = ultimo->siguiente;
ultimo->dato = nuevo;
ultimo->siguiente = NULL;
```

 Dibujar

Cadenas enlazadas: borrar el primero

```
cadena = cadena->siguiente;
```



Dibujar ¿Algún problema?

Cadenas enlazadas: borrar el primero

```
cadena = cadena->siguiente;
```



Dibujar ¿Algún problema?

¡Memory leak!
¿Solución?

Cadenas enlazadas: borrar el primero

```
primero = cadena;  
cadena = cadena->siguiente;  
free(primero);
```

 Dibujar ¿Algún problema?

¡Memory leak!
¿Solución?

Cadenas enlazadas: borrar el último

```
penultimo = cadena;  
while (penultimo->siguiente->siguiente != NULL) {  
    penultimo = penultimo->siguiente;  
}  
ultimo = penultimo->siguiente;  
penultimo->siguiente = NULL;  
free(ultimo);
```

 Dibujar