

## 08-transpas-cadenas.pdf



**Anónimo**



**Programación Para Sistemas**



**2º Grado en Ingeniería Informática**



**Escuela Técnica Superior de Ingenieros Informáticos  
Universidad Politécnica de Madrid**

# Sesión 08: *Structs* y cadenas enlazadas

Programación para Sistemas

---

Ángel Herranz

2021-2022

Universidad Politécnica de Madrid

# Recordatorio

- ¡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**

# En el capítulo de hoy...

- *Structs*
- Cadenas enlazadas

# *Structs*

---

# struct i

*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*

# 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 {  
    int x;  
    int y;  
};
```

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

```
struct punto a, b;
```

# struct iv

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

```
struct rectangulo {  
    struct punto so;  
    struct punto ne;  
};
```

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

# Punteros a *struct*

---

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

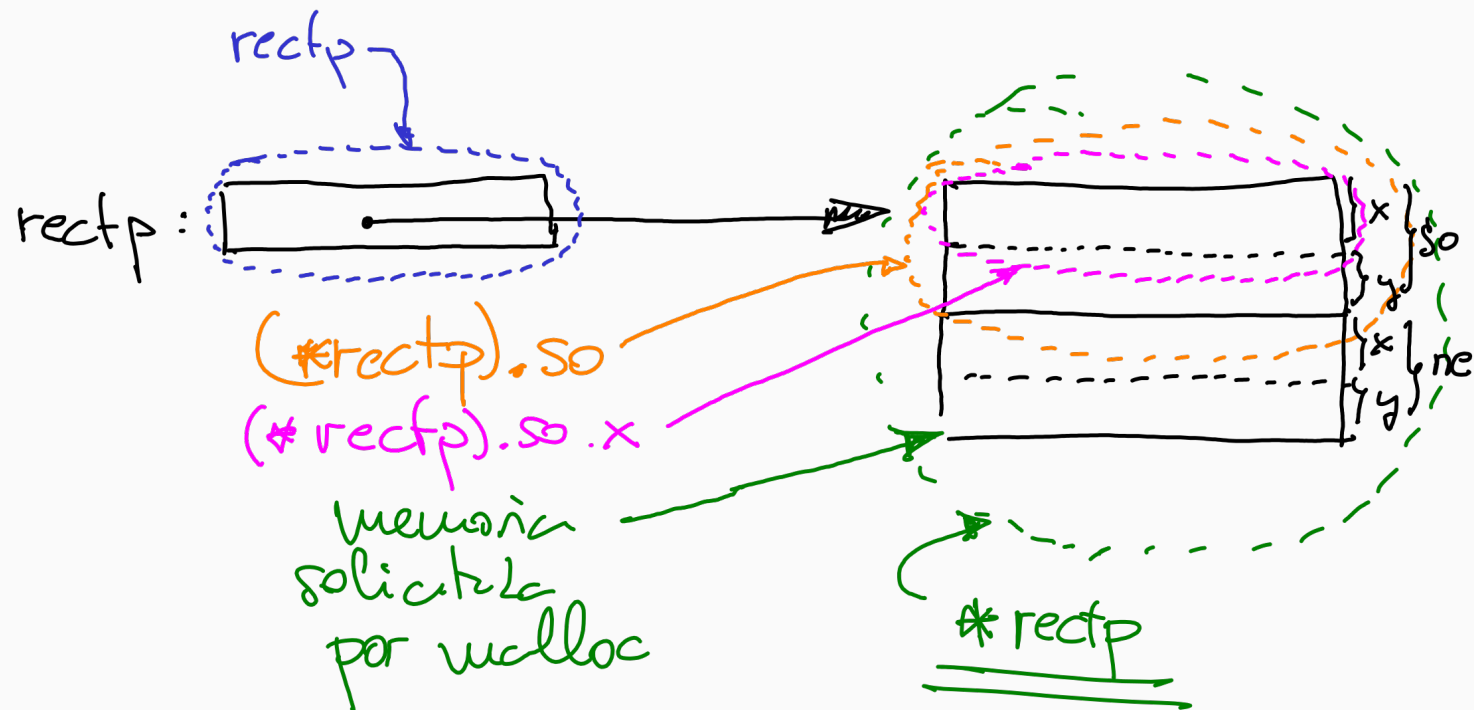
---

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

```
struct rectangulo {  
    struct punto so;  
    struct punto ne;  
};
```

# Solución

- Deberías haber dibujado algo parecido a esto:



# La *flecha*: ->

¿Qué significa (\*rectp).so?

# La *flecha*: ->

¿Qué significa (\*rectp).so?  
¿Por qué no \*rectp.so?

## La flecha: ->

¿Qué significa (\*rectp).so?

¿Por qué no \*rectp.so?

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

\*(rectp.so)



## La flecha: ->

¿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`

## La flecha: ->

¿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'

# 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 a, b;
```

```
struct rectangulo r, s;
```

# typedef de *structs*

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

```
struct punto a, b;  
struct rectangulo r, s;
```

- Usando **typedef** podríamos hacer esto

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

o 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

---

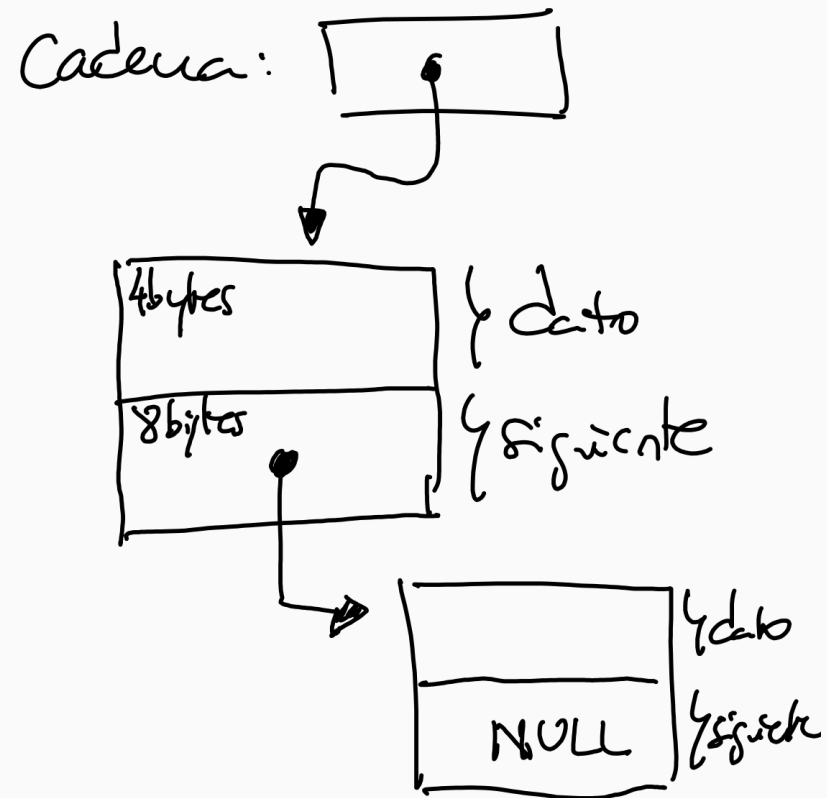
# Cadenas enlazadas: el tipo

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

# Cadenas enlazadas: el tipo

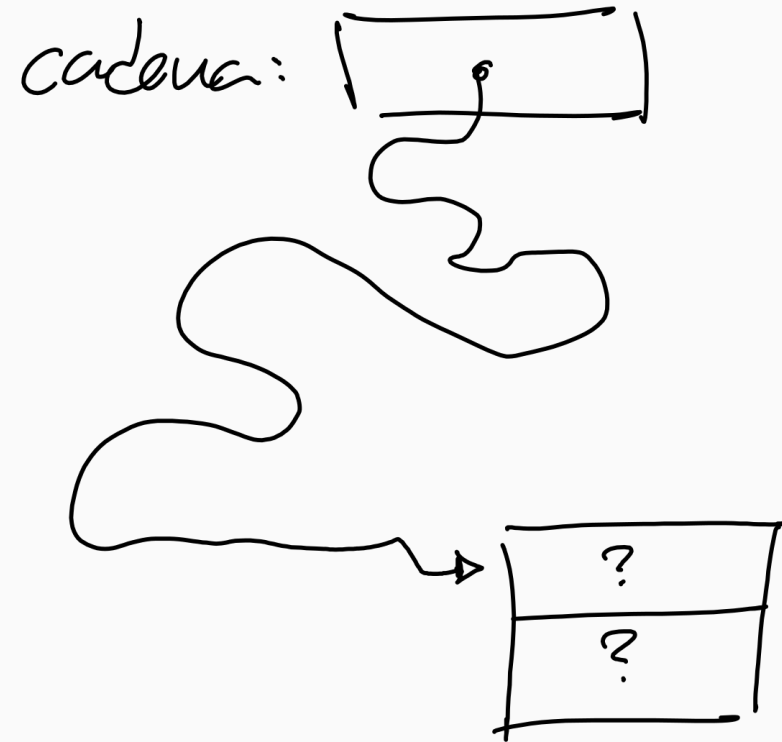
```
struct nodo {  
    int dato;  
    struct nodo *siguiente;  
};
```

```
struct nodo *cadena;
```

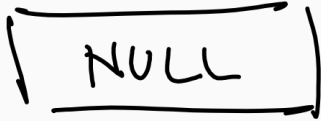


# Cadenas enlazadas: vacía

```
struct nodo *cadena;
```



# Cadenas enlazadas: vacía

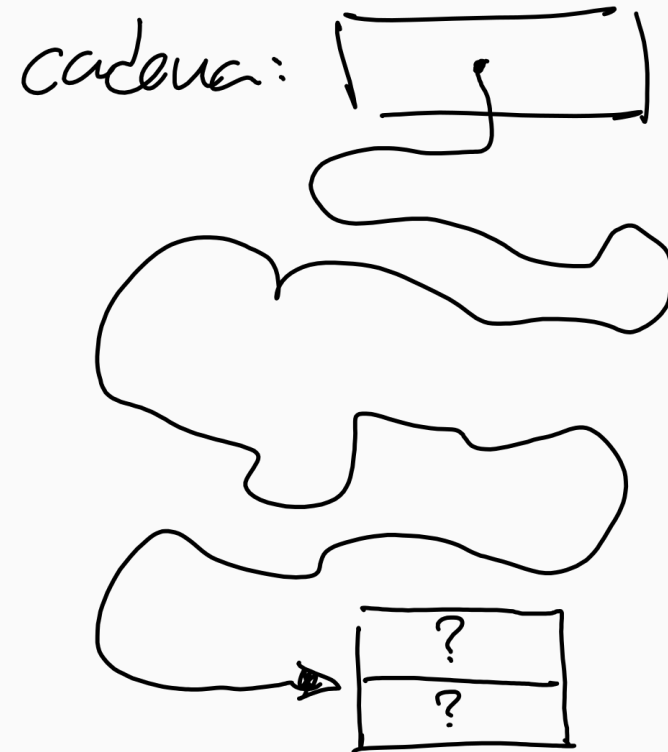
cadena: 

```
struct nodo *cadena;
```

```
cadena = NULL;
```

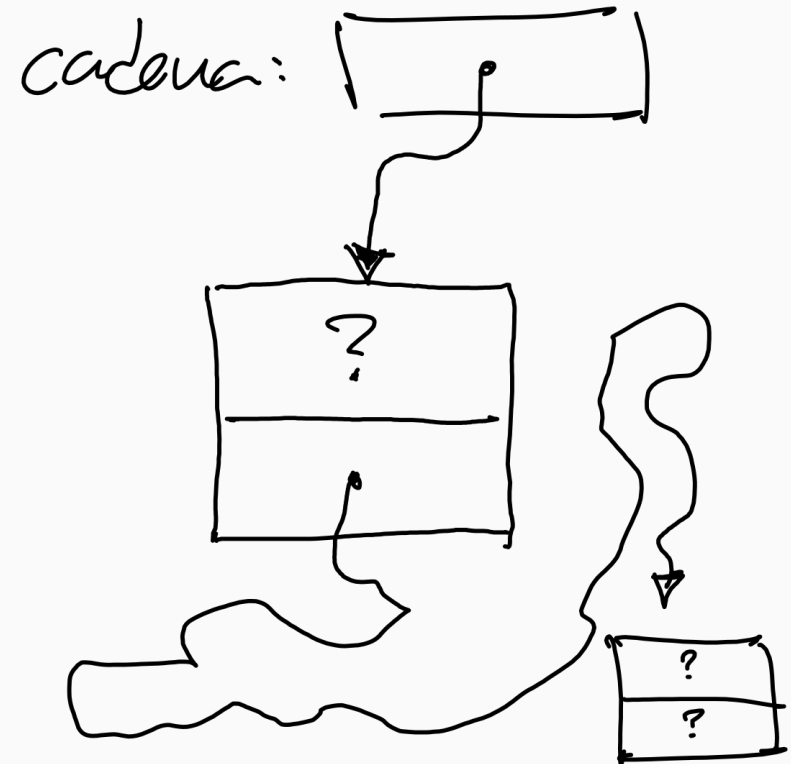
# Cadenas enlazadas: un elemento

```
struct nodo *cadena;
```



# Cadenas enlazadas: un elemento

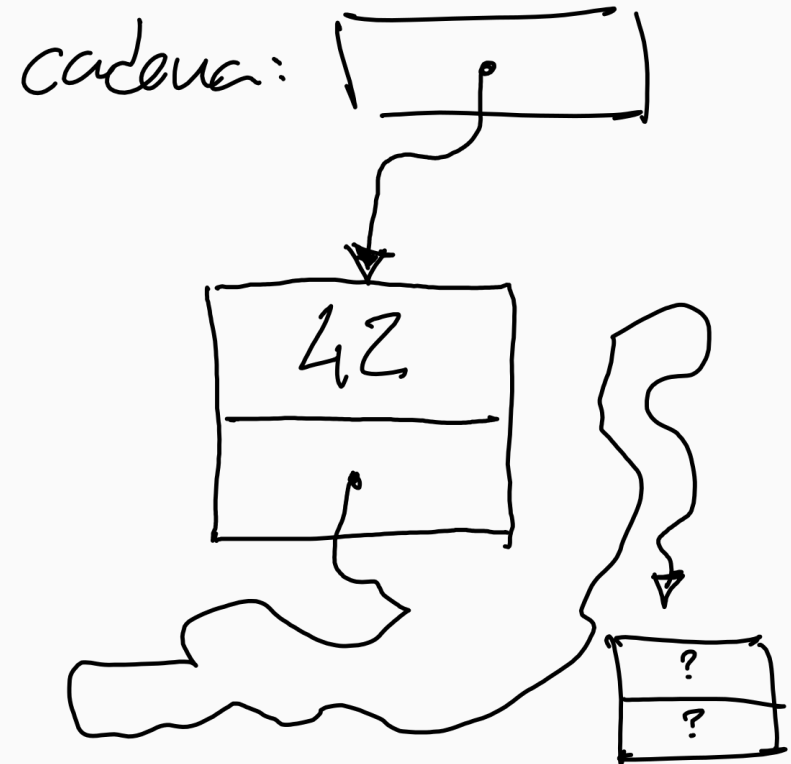
```
struct nodo *cadena;  
cadena =  
    (struct nodo *)  
    malloc(sizeof(struct nodo));
```





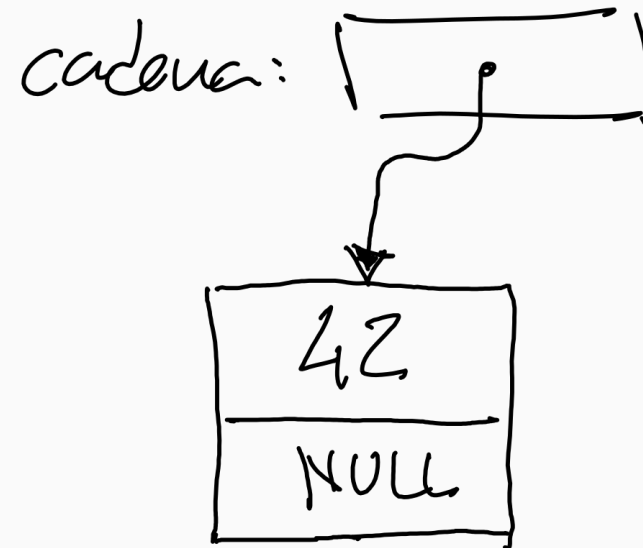
# Cadenas enlazadas: un elemento

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



# Cadenas enlazadas: un elemento

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



# Cadenas enlazadas: primero y último

- Expresión que representa el primero:

`cadena->dato`

- Recorrido hasta el último:

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

 Dibujar

# Cadenas enlazadas: añadir al principio

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

 Dibujar



# Cadenas enlazadas: añadir al final

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