

# Sesión 07: *Structs* y memoria dinámica

## Hoja de problemas


Programación para Sistemas


Ángel Herranz

aherranz@fi.upm.es

Universidad Politécnica de Madrid

Otoño 2018

 **Ejercicio 1.** Repasa las transparencias de clase. El trabajo con *structs* es muy natural, gran parte de su sintaxis es idéntica a la de otros lenguajes de programación y otra parte es muy parecida. Sin embargo, la memoria dinámica, especialmente cuando se viene de lenguajes con recolección automática de basura, se hace muy complicado, mucho más al combinarlo con la estrecha relación entre arrays y punteros de C. De nuevo, entender cada transparencia se hace

 **Ejercicio 2.** Utilizando la definición de **struct** rectángulo, tienes que escribir un programa que lea rectángulos de la entrada estándar, calcule su área y los imprima en orden, de mayor área a menor área.

La entrada estándar tendrá el siguiente formato:

- Un entero  $n$  que indica el número de rectángulos que habrá que leer. El dato  $n$  estará entre 1 y 1000.
- $n$  filas con cuatro enteros  $A_x$ ,  $A_y$ ,  $B_x$ ,  $B_y$  cada una que indican los puntos  $(A_x, A_y)$  y  $(B_x, B_y)$  que definen el rectángulo.


La salida estándar tiene que sacar  $n$  filas, cada una con un rectángulo, estando las filas ordenadas de menor a mayor área.

Veamos un ejemplo de entrada:

```
2
0 0 1 1
-1 -2 3 1
```

Y su correspondiente salida:

```
-1 -2 3 1
0 0 1 1
```

 **Ejercicio 3.** Consultar el manual man 3 malloc. En dicha página del manual encontrarás la documentación de varias operaciones funciones:

- **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);

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);
```

Feature Test Macro Requirements for glibc (see feature\_test\_macros(7)):

```
reallocarray():
    _GNU_SOURCE
```

## DESCRIPTION

The malloc() function allocates size bytes and returns a pointer to the allocated memory. The memory is not initialized. If size is 0, then malloc() returns either NULL, or a unique pointer value that can later be successfully passed to free().

...

- ☐ **Ejercicio 4.** La llamada a la función calloc( $n$ ,  $s$ ) de la biblioteca estándar devuelve un puntero a  $n$  objetos de tamaño  $s$  cada uno con toda la memoria inicializada a 0. En este ejercicio tendrás que implementar tú mismo la función calloc, puedes llamarla micalloc ;), llamando a la función malloc.

*Ejercicio 8-6 del K&R*

- » **Ejercicio 5.** Para poder probar tu implementación de ordenación de enteros propuesta en las transparencias, te sugerimos usar la potencia de Bash.

Las siguientes órdenes de bash generan un fichero `enteros.txt` con una entrada aleatoria apropiada para tu programa:

```
R=$RANDOM
echo $R > enteros.txt
for ((i = 0 ; i < $R ; i++)); do
    echo $RANDOM >> enteros.txt;
done
```

A mi me ha salido esto:

```
$ cat enteros.txt
5
29022
957
13682
10575
22773
$ |
```

Ahora, si has llamado a tu programa `ordenar_enteros`, puedes ejecutarlo así:

```
$ ./ordenar_enteros < enteros.txt
957
10575
13682
22773
29022
$ |
```

☐ **Ejercicio 6.** Modifica el programa que ordena ristas de enteros (versión final de las transparencias) para que la ordenación la realice una función `bubble_sort`. Este ejercicio está muy orientado a que puedas entender los siguientes puntos:

- Como los punteros y los arrays *son lo mismo* en C, se puede declarar un array como argumento de la función. Como es un puntero, lo que modificamos es aquello a lo que apunta, que resulta ser el contenido del array.
- En la sesión anterior ya vimos que no era posible conocer la longitud de un array cuando es argumento de una función. Por ello es necesario que la función de ordenación reciba como argumento la longitud del array.

☐ **Ejercicio 7.** Modifica el programa anterior para usar un algoritmo de ordenación más rápido que *bubble sort*: *quick sort* o *merge sort* por ejemplo.

🔊 **Ejercicio 8.** Siguiendo las indicaciones del ejercicio sobre pilas de la sesión anterior, tu trabajo es elaborar una implementación basada en cadenas enlazadas.

☐ **Ejercicio 9.** Lo cierto es que las pilas hacen un uso muy restringido de las cadenas enlazadas. Por ello, en este ejercicio, te proponemos elaborar un tipo abstracto de datos *listas*. Las funciones podrían ser algo como:

- crear\_vacia
- insertar\_por\_el\_principio
- insertar\_por\_el\_final
- insertar\_por\_posición
- longitud
- primero
- último
- iésimo
- borrar\_primer
- borrar\_último
- borrar\_por\_posición