# Algoritmos y Estructuras de Datos II

TALLER - 30 de marzo 2023

# Laboratorio 2: Ordenación

- Revisión 2023: Marco Rocchietti

## Objetivos

1. Introducir las cadenas en C
2. Repaso de **`typedef`**
3. Implementar el algoritmo de ordenación por inserción (*insertion-sort*)
4. Implementar el algoritmo de ordenación rápida (*quick-sort*)
5. Comparar desempeño de los algoritmos *selection-sort*, *insertion-sort* y *quick-sort* en distintos ejemplos
6. Lectura y comprensión del código entregado por la cátedra
7. Trabajar con implementaciones opacas de funciones leyendo su documentación
8. Abstraer la noción de orden
9. Usar procedimientos en C
10. Uso de funciones locales en módulos en C

## Ejercicio 0: Cadenas

Las cadenas en C se pueden pensar como arreglos de caracteres (más adelante veremos que también son punteros, pero por ahora… usamos arreglos). Los caracteres son valores del tipo `char` (que representa exactamente un caracter de *1 byte*), entonces para guardar un *string* en C se puede usar el siguiente arreglo:

```
char cadena[5];
```

En este ejemplo el arreglo `cadena` tiene capacidad para guardar un *string* de hasta 4 (cuatro) caracteres de longitud. Esto es así porque toda cadena en C **debe terminar con el caracter `'\0'`** por lo cual ya tenemos un lugar ocupado. Esta convención permite saber dónde termina la cadena independientemente de la capacidad del arreglo. Entonces, se puede almacenar en **`cadena`** una palabra con longitud de entre uno y cuatro caracteres, pero incluso también se puede guardar una palabra vacía (en ese caso **`cadena[0] = '\0'`**). Si queremos armar el *string* con la palabra *“hola”* podemos hacer:

```
char cadena[5]={'h', 'o', 'l', 'a', '\0'};
printf("cadena: %s\n", cadena);
```

Es muy importante no olvidarse de poner el **`'\0'`** final ya que de lo contrario **`printf()`** va a recorrer a **`cadena[]`** por fuera de su rango hasta que aparezca un **`'\0'`** y al acceder a memoria inválida se producirá eventualmente una **violación de segmento** (*segmentation fault*). Notar que no hay ningún problema en hacer

```
char cadena[10]={'h', 'o', 'l', 'a', '\0'};
printf("cadena: %s\n", cadena);
```

ya que simplemente estamos usando cinco de los diez elementos del arreglo. Como el **'\0'** se encuentra en **cadena[4]**, la función **printf()** en el ejemplo anterior va a mostrar los caracteres que hay hasta esa posición (sin incluirla).

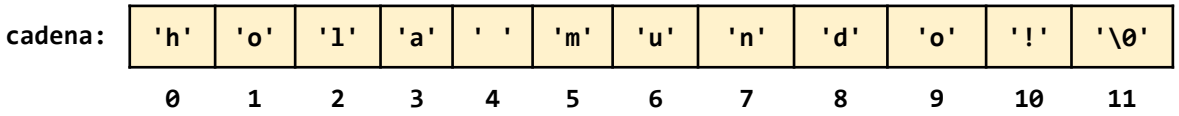
Otra forma más cómoda de armar el *string* con la palabra *"hola"* es hacer algo como:

```
char cadena[10]="hola";
printf("cadena: %s\n", cadena);
```

En este caso el caracter **'\0'** se agrega implícitamente en el arreglo **cadena.** Para no tener que contar la cantidad de caracteres que necesitamos se puede definir una cadena directamente haciendo:

```
char cadena[]="hola mundo!";
printf("cadena: %s\n", cadena);
```

el contenido del *array* es el siguiente:




Cabe destacar que todas estas maneras de armar *strings* solo son válidas al momento de inicialización del arreglo. Si hacemos

```
char cadena[]="hola mundo!";
cadena = "chau mundo!";
printf("cadena: %s\n", cadena);
```

El resultado va a ser un error de compilación. En este ejercicio vamos a trabajar con **cadenas guardadas en arreglos** que tienen un **tamaño fijo**. Vamos a definir un tipo para estas cadenas usando **typedef**. A modo de repaso, lo que hace **typedef** es dar un nombre nuevo a un tipo que ya existe en C. Entonces por ejemplo si queremos definir el tipo **natural** para los números positivos:

```
typedef unsigned int natural;
```

de esta manera cuando declaremos una variable del tipo **natural** la variable va a ser equivalente a una variable del tipo **unsigned int**. La sintaxis entonces es:

```
typedef <tipo-existente> <nuevo-tipo>;
```

Para definir el tipo nuevo **fixstring** hay que usar una sintaxis más extraña:

```
#define FIXSTRING_MAX 100

typedef char fixtring [FIXSTRING_MAX];
```

Lo que se hace aquí es definir el tipo **fixstring** como el tipo de los arreglos de elementos de tipo **char** que tienen una capacidad de **FIXSTRING_MAX** elementos. O sea que si declaramos

```
fixstring s;
```

es lo mismo que declarar

```
char s[FIXSTRING_MAX];
```

Dentro de la carpeta **ej0** van a encontrar los archivos **fixstring.h**, **fixstring.c** y **main.c**. Deben implementar en **fixstring.c** las funciones:

```
unsigned int fstring_length(fixstring s);

bool fstring_eq(fixstring s1, fixstring s2);

bool fstring_less_eq(fixstring s1, fixstring s2);
```

La función **fstring_length()** devuelve la longitud de la cadena guardada en el parámetro **s**, la función **fstring_eq()** indica si las cadenas **s1** y **s2** son iguales (contienen la misma cadena), mientras que **fstring_less_eq()** indica si la cadena guardada en **s1** es menor o igual que la guardada en **s2** en el sentido del orden alfabético. **No se permite usar librerías de C como string.h ni strings.h**. Una vez implementadas pueden probarlas compilando junto con **main.c**.

# Ejercicio 1: Insertion Sort

Dentro de la carpeta **ej1** se encuentran los siguientes archivos

| *Archivo* | *Descripción* |
|---|---|
| **array_helpers.h** | Prototipos y descripciones de la funciones auxiliares para manipular arreglos. |
| **array_helpers.c** | Implementaciones de las funciones de la librería `array_helpers` |
| **sort_helpers.h** | Prototipos y descripciones de las funciones `goes_before()`, `swap()` y `array_is_sorted()` |
| **sort_helpers.o** | Archivo binario con las Implementaciones las funciones declaradas en `sort_helpers.h` (código compilado para la arquitectura **x86-64**) |
| **sort.h** | Prototipo de la función `insertion_sort()` y su descripción |
| **sort.c** | Contiene una implementación incompleta de `insertion_sort()`, falta implementar `insert()` |
| **main.c** | Programa principal que carga un *array* de números, luego lo ordena con la función `insertion_sort()` y finalmente comprueba que el arreglo sea permutación ordenada del que se cargó inicialmente. |

*Si se trabaja en una computadora con arquitectura distinta a* **x86-64***, entonces seleccionar y renombrar uno de los siguientes archivos,* `sort_helpers.o_32` *o* `sort_helpers.o_macos` *según la arquitectura de su máquina.*

## **Parte A:** Ordenación por Inserción

Se debe realizar una implementación del algoritmo de ordenación por inserción (alias *insertion-sort*). Para ello es necesario completar la implementación del "procedimiento" `insert()` en el módulo **`sort.c`**. Como guía se puede examinar el resto del archivo **`sort.c`** y la definición del algoritmo de ordenación por inserción vista en el teórico. El algoritmo debe ordenar con respecto a la relación `goes_before()` declarada en **`sort_helpers.h`** cuya implementación está oculta puesto que viene ya compilada en **`sort_helpers.o`**.

## **Parte B:** Chequeo de Invariante

Se debe modificar el "procedimiento" `insertion_sort()` agregando la verificación de cumplimiento de la invariante del ciclo `for` que se vio en el teórico. Por simplicidad sólo se debe verificar la siguiente parte de la Invariante:

- el segmento inicial **`a[0,i)`** del arreglo está ordenado.

Para ello usar las funciones `assert()` y `array_is_sorted()`.

## Compilación

Una vez implementados los incisos *a)* y *b)*, se puede compilar ejecutando:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c array_helpers.c sort.c main.c
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -no-pie array_helpers.o sort.o sort_helpers.o main.o -o sorter
```

la opción **`-no-pie`** tiene que ver con que se están "linkeando" los objetos `array_helpers.o`, `sort.o` y `main.o` compilados en nuestra computadora con el objeto precompilado `sort_helpers.o`, cuya compilación fue realizada en una computadora distinta. En consecuencia esta opción puede ser necesaria para lograr compatibilidad entre los archivos binarios durante el "linkeo" y así poder generar el ejecutable. El programa puede ejecutarse de la siguiente manera:

```
$ ./sorter ../input/example-unsorted.in
```

Si el programa funciona bien en ese ejemplo (es decir, si no reporta error), probar con otros archivos de la carpeta **`../input`**, sin olvidar realizar una prueba con el archivo **`../input/empty.in`**

Analizar los resultados del programa y responder: ***¿Qué relación implementa la función*** `goes_before()` ***?¿Cuál es el criterio que usa?***

# Ejercicio 2: Quick Sort I

En este ejercicio se realizará una implementación *top-down* del algoritmo de ordenación rápida vista en el teórico. En la carpeta **`ej2`** se encuentran los siguientes archivos:

| ***Archivo*** | ***Descripción*** |
|---|---|
| **`array_helpers.h`** | Es el mismo que en el ejercicio anterior. |
| **`array_helpers.c`** | Es el mismo que en el ejercicio anterior. |
| **`sort_helpers.h`** | Contiene además la declaración y descripción de `partition()` |
| **`sort_helpers.o`** | Contiene implementaciones ilegibles de esas funciones (código compilado para la arquitectura **`x86-64`**) |
| **`sort.h`** | Contiene descripción de la función `quick_sort()` |
| **`sort.c`** | Contiene una implementación muy incompleta de `quick_sort()`, además falta implementar `quick_sort_rec()` |
| **`main.c`** | Contiene el programa principal que carga un arreglo de números, luego lo ordena con la función `quick_sort()` y finalmente comprueba que el arreglo sea una permutación ordenada del que se cargó inicialmente. |

*Si se trabaja en una computadora con arquitectura distinta a* **`x86-64`***, entonces seleccionar y renombrar uno de los siguientes archivos,* `sort_helpers.o_32` ***o*** `sort_helpers.o_macos` *según la arquitectura de su máquina.*

## **Parte A:** Implmentación de `quick_sort_rec()`

Implementar el "procedimiento" `quick_sort_rec()` en el archivo **`sort.c`**. Tener en cuenta que **no es necesario** implementar la función `partition()` puesto que la misma ya está implementada (aunque no puede leerse su código por estar compilada en `sort_helpers.o`). Para saber cómo utilizarla, examinar su descripción en **`sort_helpers.h`**.

A modo de guía se puede revisar la presentación del algoritmo de ordenación rápida realizada en la clase del teórico.

## **Parte B:** Función `main()`

Se debe abrir el archivo **`main.c`** y completar la función `main()` con una llamada al "procedimiento" `quick_sort()`. Para entender cómo utilizar este "procedimiento", examinar el archivo **`sort.h`**.

## Compilación

Una vez completadas las partes A y B, compilar el código con **`gcc`** siguiendo el mismo método del ejercicio 1.

# Ejercicio 3: Quick Sort II

En la carpeta **`ej3`** se encuentran los siguientes archivos

| ***Archivo*** | ***Descripción*** |
|---|---|
| **`sort_helpers.h`** | Contiene descripciones de las funciones `goes_before()`, `swap()` y `array_is_sorted()` |
| **`sort_helpers.o`** | Contiene implementaciones ilegibles de todo lo descripto en `sort_helpers.h` (código compilado para la arquitectura **`x86-64`**). Notar que la función `partition()` no está más aquí. |
| **`sort.h`** | Contiene descripción de la función `quick_sort()` |
| **`sort.c`** | contiene una implementación incompleta de `quick_sort()`, falta implementar `quick_sort_rec()` y `partition()`. |

*Si se trabaja en una computadora con arquitectura distinta a* **`x86-64`***, entonces seleccionar y renombrar uno de los siguientes archivos,* `sort_helpers.o_32` ***o*** `sort_helpers.o_macos` *según la arquitectura de su máquina.*

Copiar los archivos **`array_helpers.h`**, **`array_helpers.c`** y **`main.c`** del *ejercicio 2*. Luego copiar el "procedimiento" `quick_sort_rec()` (también del *ejercicio 2*) en el archivo **`sort.c`** y **definir** allí la función `partition()` usando como guía la presentación que se dio del algoritmo de ordenación rápida en la clase del teórico.

## Compilación

Una vez completada la definición de `partition()`, compilar el código con **`gcc`** siguiendo el mismo método del ejercicio 1.

# Ejercicio 4: Versus

Realizar una comparación de todos los algoritmos de ordenación implementados en este laboratorio. En la carpeta **`ej4`** se encuentran los siguientes archivos:

| ***Archivo*** | ***Descripción*** |
|---|---|
| **`sort_helpers.h`** | Se agregan nuevas declaraciones de funciones para manejo de contadores |
| **`sort_helpers.o`** | Contiene implementaciones ilegibles de todo lo descripto en `sort_helpers.h` (código compilado para la arquitectura **`x86-64`**) |
| **`sort.h`** | Contiene las declaraciones y descripciones de las implementaciones de los métodos de ordenación *selection-sort*, *insertion-sort* y *quick-sort* |
| **`sort.c`** | Contiene las definiciones incompletas de las funciones declaradas en `sort.h`. Deben completarse con el código de los ejercicios anteriores. |
| **`main.c`** | Contiene el programa principal que carga un arreglo de números, luego lo ordena usando alguno de los algoritmos de ordenación implementados y muestra:<br>• Tiempo de ejecución<br>• Número de comparaciones<br>• Intercambios realizados. |

*Si se trabaja en una computadora con arquitectura distinta a* **`x86-64`***, entonces seleccionar y renombrar uno de los siguientes archivos,* `sort_helpers.o_32` ***o*** `sort_helpers.o_macos` ***según la arquitectura de su máquina.***

Copiar los archivos **`array_helpers.h`** y **`array_helpers.c`** del ejercicio anterior y luego:

1. Abrir el archivo **`sort.c`** y copiar el código de cada uno de los algoritmos de ordenación resueltos en los ejercicios anteriores.
2. Abrir el archivo **`main.c`** y completar la función `main()` siguiendo los pasos indicados en los comentarios.

## Compilación y Ejecución

Una vez completados los ítems 1 y 2, compilar el código con **`gcc`** siguiendo el mismo método del ejercicio 1.

Analizar los resultados de la ejecución del programa para distintos ejemplos y sacar conclusiones sobre el desempeño de cada algoritmo de ordenación.

# Ejercicio 5: Ordenación alfabética

En la carpeta **`ej5`** van a encontrar los archivos **`fixstring.h`**, **`fixstring.c`**. Deben copiar las implementaciones de **`fstring_length()`**, **`fstring_eq()`** y **`fstring_less_eq()`** realizadas en el *ejercicio 0* a **`fixstring.c`** y luego completar la implementación de las función

```
void fstring_swap(fixstring s1,  fixstring s2);
```

que debe intercambiar los contenidos de las cadenas **`s1`** y **`s2`**. Para implementar esta función pueden utilizar de manera auxiliar a **`fstring_set()`** (que ya viene implementada).

Se incluye un código muy parecido al de los ejercicios anteriores pero que es capaz de leer un arreglo de palabras, es decir un arreglo de elementos de tipo **`fixstring`**. Los archivos de entrada están en la carpeta **`ej5/input.`** La idea será ordenar las palabras usando el algoritmo de *quick sort*. Deben entonces adaptar el código para que:

**a)** Se ordene el arreglo de *strings* de entrada de manera alfabética

**b)** Se ordene el arreglo de *strings* de entrada según el largo de las cadenas (la cadenas más cortas al principio).

La forma de compilar el código es:

```
$ gcc -Wall -Werror -Wextra -pedantic -std=c99 -c array_helpers.c fixstring.c main.c sort.c sort_helpers.c
$ gcc -Wall -Wextra -std=c99 array_helpers.o fixstring.o main.o sort.o sort_helpers.o -o word_sorter
```

Y luego un ejemplo de ejecución:

```
$ ./word_sorter input/example-easywords.in
4
casa chau hola perro
```