# Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)
*Universitat Politècnica de València*

fSO

# Práctica 2
# Programación en C (II)

## Contenido

# 1 Objetivos

Los objetivos generales de esta práctica son conocer nuevos aspectos del lenguaje C y practicarlos utilizando las herramientas UNIX:

- Profundizando en la estructura de funciones en C y en el manejo de variables en sus diversos ámbitos.
- Practicar los tipos de datos derivados de C como: cadenas, punteros y estructuras.
- Manejar el paso de parámetros en main(), que permite utilizar argumentos desde la línea de órdenes.

Para ello, el alumno debe realizar y/o modificar una serie de programas en C, comprobando resultados de compilación y ejecución.

## 1.1 Entorno necesario

El entorno de trabajo es el mismo descrito en la sesión anterior: Linux y el compilador gcc (GNU Compiler). Como entorno de edición se puede utilizar un editor de texto, se recomienda usar el editor **kate**.
Esta práctica también se puede realizar en Mac OS X, utilizando gcc y XCode[1].
**Los archivos con código fuente para realizar la práctica se deben descargar del Poliformat**.

# 2 Funciones en un programa C

En la sesión anterior hemos creado programas en C destacando la función principal denominada `main()` . Además de `main()`, en C se distinguen otros dos tipos de funciones: las definidas por el propio programador y las ya definidas en las librerías de C. Por ejemplo `printf()` está definida en la librería `stdio.h`. La figura-1 muestra el programa "`circulo.c`" con el que trabajará añadiéndole código y funciones.

```
#include <stdio.h>

#define PI 3.1416
main() {
   float area, radio;
   radio = 10;
   area = PI * (radio * radio);
   printf("Area del circulo de radio %f es %f\n", radio, area);
}
```

Figura-1: Código del fichero "*circulo.c*"

Recuerde que para compilar y generar archivos con código ejecutable debe utilizar la orden:

```
$ gcc -o circulo circulo.c
```

Compruebe que se ha generado el ejecutable con la orden `ls –la`. Para ejecutar el programa sólo tiene que invocar el ejecutable desde la línea de órdenes:
**Nota:** Se requiere el prefijo ./ porque en Linux el directorio de trabajo no está incluido en la variable PATH.

```
$  ./circulo
```

[1] NOTA: Aunque existe una versión del gcc para Windows, su funcionamiento difiere a lo descrito en esta memoria y por lo tanto no se recomienda su uso.

## 2.1 Ejercicio 1: definiendo y declarando funciones

Esta sección muestra cómo trabajar con funciones en un programa C.
Copie el contenido de `circulo.c` en un nuevo archivo `circulo2.c`. Al final del archivo `circulo2.c` y fuera de la función añada la función `areaC()`, con el código para el cálculo del área. La función areaC() tendrá el radio del circulo como parámetro de entrada y retornará el cálculo del área. Por lo que debe reemplaza la línea:

```
area = PI * (radio * radio);
```

por:

```
area = areaC(radio);
```

Compile y compruebe los mensajes que aparecen:

```
$ gcc circulo2.c –o circulo2
```

## 2.2 Cuestiones: errores o warnings

¿Qué indican los mensajes errores o warnings (avisos)?, ¿Cómo interpreta de dichos mensajes?

Par evitar los problemas de compilación observados puede proceder de dos modos: i) Cambiar el orden de las funciones, de manera que el código de `areaC` preceda al de la función `main()` como muestra la figura-2. Realice dicho cambio y observe que se genera el ejecutable de forma correcta. ii) Dejar la implementación de `areaC()` después de main() pero declarar dicha función antes del main(), mediante la línea:

```
float areaC(float radio);
```

Resaltar que en la declaración de la función sólo se indica el tipo de valor que devuelve y el de sus argumentos. Para evitar errores, las funciones en C deben estar definidas o declararlas de forma previa a su invocación.

```
#include <stdio.h>

#define PI 3.1416
float areaC (float radio) {
   return (PI * (radio * radio));
}
main() {
   float area, radio;
   radio = 10;
   area = areaC(radio);
   printf("Area del circulo de radio %f es %f\n", radio, area);
}
```

Figura-2: Código del fichero "*circulo2.c*"

## 2.3 Ámbito de las variables

Podemos observar en el código de la figura-2 (`circulo2.c`) la aparición de la variable `radio` en dos lugares diferentes: función `areaC` y función `main()`. Ello es posible debido a cómo trabaja C el ámbito de las variables:

- **Variables Globales**: las variables se declaran fuera de cualquier función y se pueden acceder desde cualquier función implementada dentro del archivo
- **Variables Local**: se definen dentro de una función y solo son accesibles dentro de dicha función. Estas variables no son persistentes y pierden su valor una vez que la función finaliza.
- **Variables Estáticas**: son variables locales pero que persiste su valor, por lo que conservan su valor final para la próxima llamada a función.

¿Qué ocurre cuando en un programa aparecen variable globales y locales definidas con el mismo nombre?, En dicha situación, la asignación de las variables locales tiene prioridad sobre la asignación de las globales del mismo nombre. Para comprobarlo compile el código `variables.c` de la figura-3 y analice su resultado ¿Qué valor piensa que se mostrará en pantalla para la variable x?

```
#include <stdio.h>
int x=2;
void main(){
    int x=3;
    m();
    printf("%d", x);
}

void m(){
    x = 4;
}
```

Figura-3: Código del fichero "*variables.c*"

## 2.4 Ejercicio 2: trabajando con variables

El código fuente de `variablesVarias.c` mostrado en la figura-4 trata de revisar situaciones que pueden surgir con el uso de las variables en sus diversos ámbitos.

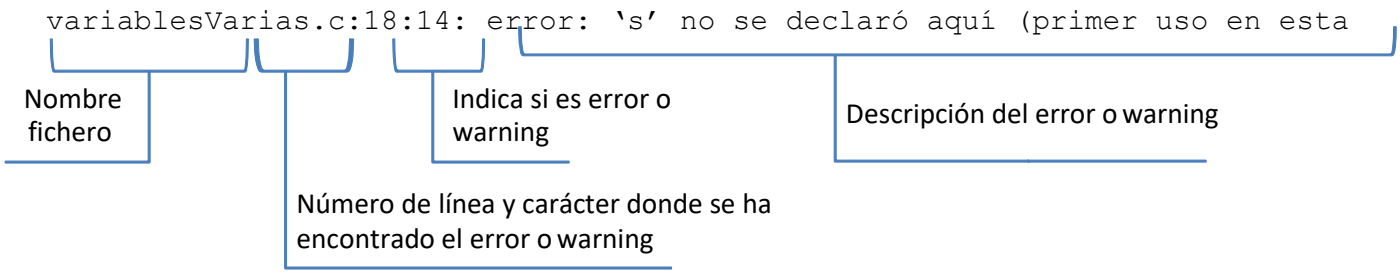
```
#include <stdio.h>
int a = 0; /* variable global */

// Función incrementa en 1 el valor de la variable global a
void inc_a(void) {
    int a;
    a++;
}

// Función retorna el valor anterior de v y guarda el nuevo valor v
int valor_anterior(int v) {
    int temp;

 // Declare aquí la variable s como estática
      temp = s;
      s = v;
      return b;
}

main()
{
      int b = 2; /* variable local */
      inc_a();
      valor_anterior(b);
      printf("a= %d, b= %d\n", a, b);
      a++;
      b++;
      inc_a();
      b = valor_anterior(b);
      printf("a= %d, b= %d\n", a, b);
}
```

Figura-4: Código a corregir proporcionado en el fichero "*variablesVarias.c*"

Compile `variablesVarias.c`

```
$ gcc -o variablesVarias variablesVarias.c
```

Al compilar aparecerán en el terminal una serie de errores. Los errores en los programas en C impiden la generación del código ejecutable y suelen tener el siguiente formato:




El programa contiene errores de programación que deben ser corregidos:

- La función `inc_a` trabaja con la variable global a, por lo que no debe estar definida como local.
- Según muestra el mensaje de error del compilador, hay que declarar la variable s, en la función `valor_anterior`. El valora anterior de v , estará en s al comenzar la función y finalmente en temp, por lo tanto la función debe devolver `temp`, en vez de b.

Al ejecutar `variablesVarias`, con las correcciones adecuadas, debe mostrar en consola:

```
a= 1, b= 2
a= 3, b= 2
```

### 2.5 Cuestiones

a) Justifique el cambio producido en el valor de la variable a y porqué se incrementa hasta tener un valor 3.
b) Asimismo, razone porqué se mantiene el valor de la variable b.

## 3 Parámetros por línea de órdenes

En UNIX es usual ejecutar una orden con diferentes parámetros leídos desde el terminal. Ejemplo: gcc dispone de multitud de opciones y parámetros que se configuraran por línea de comandos. Un programa en C, puede trabajar con parámetros pasados desde la línea de comandos. Para ello, la función `main()debe` estar definida con dos argumento **`argc`** y **`argv`** como:

```
int main(int argc, char *argv[])
```

donde:

a) `argc` contendrá el número de argumentos pasados por la línea de comandos, que siempre será mayor que cero, ya que el nombre de la orden es el primer argumento.
b) `argv` es un vector de cadenas con los argumentos. El primer elemento de este vector (`argv[0]`) será siempre el nombre de la orden.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
     // A completar…
}
```

Figura-5: Contenido inicial del fichero "*arguments.c*"

### 3.1 Ejercicio 3: programa argumentos

Partiendo de la base del archivo `arguments.c`, implemente un programa denominado `arguments.c`, que muestre por pantalla el número de argumentos "argc" y el contenido de cada una de ellos "argv[i]". Implemente un bucle que a partir del valor `argc,` invoque la función `printf` para mostrar cada argumento introducido desde la línea de órdenes. El resultado de su ejecución con distintos argumentos debe ser:

```
$ ./arguments
Numero de argumentos = 1
  Argumento 0 es ./arguments
$  ./arguments uno dos tres
Numero de argumentos = 4
  Argumento 0 es ./arguments
  Argumento 1 es uno
  Argumento 2 es dos
  Argumento 3 es tres
```

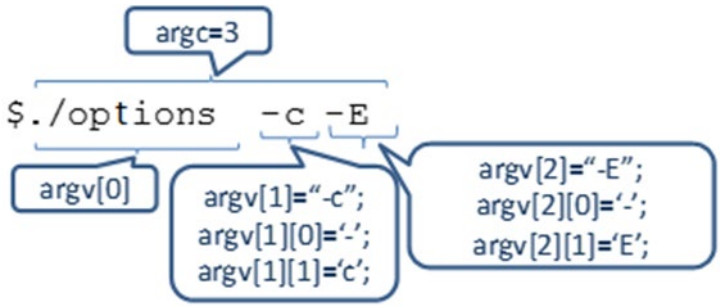
### 3.2 Ejercicio 4: programa opciones

Implemente un programa denominado `options.c` que, de forma similar al gcc, acepte las siguientes opciones y si es el caso muestre la ruta:

```
-c   Mostrará "Compilar"
-E   Mostrará "Preprocesar"
-i   Mostrará "Incluir + ruta"
```

A continuación, se muestra el resultado a conseguir en la ejecución de dicho programa con cada argumento:

```
$ ./options -c
   Argumento 1 es Compilar
$  ./options -c -E -i/includes
   Argumento 1 es Compilar
   Argumento 2 es Preprocesar
   Argumento 3 es Incluir /includes
```




## 4 Punteros y estructuras

Un puntero es una variable que contiene la dirección de otro objeto. Esto nos permite acceder y modificar elementos de cadenas y estructuras de forma simple. En esta parte de la práctica debe completar pequeños programas que trabajan con punteros, cadenas y estructuras.

## 4.1 Ejercicio 5: programa mayúsculas

Complete el programa `mayusculas.c` de la figura 6, para que lea un texto desde consola, lo convierta a mayúsculas y lo muestre por pantalla. Para ello:

a) Defina las variables `cadena` y `cadena2` como vectores de cadena con un tamaño `TAM_CADENA`.
b) Asigne a la variable cadena el texto leído desde consola. Para leer tiras de caracteres que contengan blancos se puede utilizar
   `scanf("%[^\n]s", str).`
c) Complete el bucle de conversión a mayúsculas. Para ello haga uso de dos punteros a cadenas, donde `p1` apunta a `cadena` y `p2` apunta a `cadena2.` Se debe ir copiando el elemento apuntado por `p1` a `p2` restándole 32 para convertirlo a mayúscula, sólo si es un carácter en minúscula. Al final del bucle debe añadir el carácter cero de fin de cadena en la cadena2.
d) Muestre por pantalla el string `cadena2`, que contendrá el texto convertido a mayúscula.

```
#include <stdio.h>
#define TAM_CADENA 200
main() {
    // Puntero a caracter para copiar las cadenas
    char *p1, *p2;

    // A) Definir las variables cadena y cadena2

    // B) Leer cadena de consola

    // C) Convertir a mayúsculas

    p1 = cadena;
    p2 = cadena2;
    while (*p1 != '\0') {
        //  Copiar p1 a p2 restando 32
    }

    // Acordarse de poner el cero final en cadena2
    // D) Sacar por consola la cadena2.
}
```

Figura-6: Contenido inicial del fichero “mayusculas.c”

## 4.2 Ejercicio 6: programa sumafilas

Complete el programa `sumafilas.c` de la figura 7, para que imprima en el terminal el resultado de sumar los datos de cada fila y el resultado de sumar los datos de todas las filas . Cada fila está definida como una estructura de dos elementos: un vector de datos y suma.
Para ello debe:

a) Definir una variable denominada `filas` que sea un vector de estructuras `FILA` de tamaño `NUM_FILAS.`
b) Implementar la función `suma_fila.` Esta función requiere como argumento el puntero de la fila a sumar y debe sumar el vector de datos de dicha fila y almacenar el resultado en su elemento `suma`.
c) Completar el bucle para sumar todas las filas, invocando a `suma_fila` con el argumento adecuado de la fila, completar el `printf` y actualizar la variable `suma_total`.

```
#include <stdio.h>

#define TAM_FILA 100
#define NUM_FILAS 10
struct FILA {
    float datos[TAM_FILA];
    float suma;
};
// A)Defina variable filas como un vector de FILA de tamaño NUM_FILAS

void suma_fila(struct FILA *pf) {
// B) Implemente suma_fila
}

// Inicie las filas con el valor i*j
void inicia_filas() {
    int i, j;
    for (i = 0; i < NUM_FILAS; i++) {
        for (j = 0; j < TAM_FILA; j++) {
            filas[i].datos[j] = (float)i*j;
        }
    }
}
main() {
    int i;
    float suma_total;

    inicia_filas();
    // C) Complete bucle
    suma_total = 0;
    for (i = 0; i < NUM_FILAS; i++) {
        // Llamar a suma_fila
        printf("La suma de la fila %u es %f\n", i, /* COMPLETAR */);
        // sumar la fila a suma_total
    }
    printf("La suma final es %f\n", suma_total);
}
```

Figura-7: Contenido inicial del fichero “sumafilas.c”

El resultado final de la ejecución del programa debe ser:

```
$  ./sumafilas
La suma de la fila 0 es 0.000000
La suma de la fila 1 es 4950.000000
La suma de la fila 2 es 9900.000000
La suma de la fila 3 es 14850.000000
La suma de la fila 4 es 19800.000000
La suma de la fila 5 es 24750.000000
La suma de la fila 6 es 29700.000000
La suma de la fila 7 es 34650.000000
La suma de la fila 8 es 39600.000000
La suma de la fila 9 es 44550.000000
La suma final es 222750.000000
```

# 5 Paso de parámetros por referencia

Para terminar esta sesión, vamos a introducir el paso de parámetros por referencia en una función. En las funciones `areaC()`, `circulos2.c` (figura 2) o variablesVarias.c (figura 4) se utiliza el paso de parámetros por valor, ya que se le pasa a la función el valor de la variable. El paso de parámetros por referencia implica proporcionar a la función la dirección a la variable, el puntero a la variable. Por ejemplo, la función `suma_fila` (figura 7) tiene como argumento un puntero a una estructura (`struct FILA *pf`). Resumiendo:

- **Por valor** significa que la función recibe sólo una copia del valor que tiene la variable, y por tanto no la puede modificar.
- **Por referencia** significa que se pasa la posición de memoria donde esta guardada la variable, por lo que la función puede saber cuánto vale, pero además puede modificar el valor de la variable.

El ejemplo de la figura-8, utiliza paso de parámetros por referencia para actualizar el valor de la variable "`c`".

**Nota:** La variable `c es pasada a F con "&c",` en realidad se pasa la dirección de "`c`" no su valor.

```
#include <stdio.h>

char F(char *c){
        c[0] = 'f';
        return (*c);
}
main () {
        char c;
        c = 'a';
        printf("%c\n",c);
        printf("%c\n", F(&c));
        printf("%c\n", c);
}
```

Figura-8: Código del fichero "*parametroSalida.c*"

## 5.1 Ejercicio 7: programa sumafilas2

Modifique la versión del programa "`sumafilas.c`" desarrollada en el apartado 4.2 de manera que esta nueva versión "`sumafilas2.c`" tenga las siguientes características:

a) La variable `filas` sea local y definida en la función `main`.
b) Implemente una función `inicia_fila` que requiera el puntero de la fila a inicializar. Esta función será llamada desde `main()` antes de invocar la función `suma_fila`.

El resultado de ejecución debe coincidir con el de "`sumafilas.c`" del apartado 4.2.