Algoritmos y Estructuras de Datos
2º de Ingeniería Informática, Curso 2008/2009

SEMINARIO "C para programadores java"
(este seminario forma parte de las actividades del *proyecto piloto*)

Sesión 3

Contenidos:

1. Funciones
2. Gestión dinámica de memoria.
3. Funciones de E/S.

Ejercicios

---

1. Funciones

- **Estructura de definición de funciones**:

```
tipo_devuelto nombre_función ( parámetros ) { cuerpo }
```

Ej.:
```
int suma (int a, int b) {
    int r= a+b;
   return r;
}
```

- Una función no se puede declarar dentro de otra (no se pueden anidar), aunque sí se pueden definir y anidar bloques: { … { … } … { { } … } …}
- **Valor devuelto.**
  - Sólo puede haber 1 tipo devuelto (como en Módula o Pascal).
  - Si no devuelve nada se pone: **`void`**
  - Por defecto, si no se pone nada, se supone que devuelve un `int`.
  - Se puede devolver un `struct` o `union`, pero no un array. En su lugar, se puede devolver un puntero al array.
  - Acabar la ejecución del procedimiento: **`return;`**
  - Acabar y devolver un *valor*: **`return`** `expresion;`
- **Parámetros.**
  - Lista de pares: **`(`**`tipo1 nombre1, tipo2 nombre2, ...`**`)`**
  - Cada nombre debe llevar su tipo (aunque se repitan).
  - El paso es siempre **por valor**.
  - Simulación del paso **por referencia**: usar punteros.

```
void suma2 (int a, int *b) {
   int r;
   r= a + *b;
   *b= r;
}
```

  - **OJO: no devolver variable local por referencia:** la variable local está en la pila, y desaparece al acabar la ejecución de la función. Ejemplo:

```
void suma3 (int a, int b) {
   int r;
   r= a + *b;
   return &r;
}
```

  - **Paso de arrays:**

```
float media (int array[], int tamanno) ...
```

    - no se especifica el tamaño del array a recibir;
    - el paso de parámetros array es por referencia: se accede al array original a través de la variabl-parámetro
    - alternativa: usar punteros (`int * array` en vez de `int array[]`)

- **Ejemplo de función recursiva:**

```
int factorial(int n) {
   if (n=0) return 1;
   return n*factorial(n-1);
}
```

- **Variables locales.**
  - **Se crean en la pila para cada llamada y se eliminan al acabar la llamada. Insistimos: no devolver por referencia.**
  - Se llaman variables `auto`; auto es la opción por defecto, no hace falta indicarla.
  - Deben ir siempre al principio del cuerpo de la función, justo después de las llaves, `{`. Nunca entre líneas de código. En C++ sí se puede.
  - **Variables locales `static`**:
    - **primero**: olvidar significado de *variable static* en java[1] (variable de clase compartido por todos los objetos de esa clase);
    - en C, son variables:
      - locales en cuanto a visibilidad
      - globales en cuanto a tiempo de vida:
        - existen haya o no haya en marcha una ejecución de la función;
        - conservan sus valores entre distintas llamadas
    - Inicializar en declaración; se ejecutará sólo en la primera llamada.
    - OJO: no usar junto con recursividad: hay una sóla instancia de la variable para todas las llamadas recursivas a la función; ejemplo: pensar función factorial, implementada con variable static ¿?
    - Ejemplo de uso:

```
int ticket(void) {
   static int turno=1;
   return turno++;
}
```

  - **Variables locales `register`:** *sugiere* al compilador que la variable sea almacenada en un registro de la CPU (a ser posible).

```
float media (int array[], int tam) {
   static float acum;
   register int i;
   for (i= 0, acum= 0; i<tam; i++)
      acum+= array[i];
   return acum/tamanno;
}
```

- **Declaración de una función.** En algunos casos puede ser necesario *declarar* la sintaxis de la función antes de *definirla* (p. ej. si hay doble recursividad).

```
tipo_devuelto nombre_función ( parámetros );
```

  - En la lista de parámetros se pueden poner sólo los tipos (omitir los nombres).

```
int suma (int, int);
float media (int [], int);
```

[1] ¿sabéis lo que son las variables *static* en java?

- Parámetros y valor devuelto por **la función `main`**.
  - **Valor devuelto.** Si existe, debe ser un `int`. Es el valor devuelto por el programa al sistema operativo, como código de error.
  - **Parámetros.** Indican los argumentos escritos por el usuario en la línea de comandos.
    - Primero: de tipo `int`. Número de argumentos introducidos.
    - Segundo: array de cadenas `(char *)`. Indica el contenido de esos argumentos.

```
main(int num_arg, char *str_arg[]){
   int i;
   printf("Hay %d argumentos.\n", num_arg);
   for(i=0; i<num_arg; i++)
      printf("Argumento %d: %s\n", i, str_arg[i]);
}
```

- **Paso de funciones como parámetros.**
  - Es posible definir **punteros a funciones**, pasarlas como parámetros de una función y aplicar los operadores de dirección e indirección (llamar a la función apuntada).
  - Declaración de parámetro de tipo puntero a función:

```
valor_devuelto (*nombre_parametro) (tipo1, tipo2, ...)
```

  - Ejemplo.

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

int suma (int a, int b)
{return a+b;}

int mult (int a, int b)
{return a*b;}

int opera (int a, int b, int (*op)(int,int))
{
   return(*op)(a, b);
}

main(int argc, char *argv[]) {
   int a, b;
   if (argc<3) return -1;  /* ERROR: faltan operandos */
   a= atoi(argv[1]);
   b= atoi(argv[2]);
   printf("Suma: %d\n",opera(a, b, &suma));
   printf("Prod: %d\n",opera(a, b, &mult));
}
```

2. Gestión dinámica de memoria

- Se puede **reservar memoria en tiempo de ejecución**. Las funciones para reserva de memoria están en: `stdlib.h`
- Toda la memoria reservada **debe ser liberada** antes de acabar. Es decir, no hay liberación automática de memoria, **NO HAY RECOLECTOR DE BASURA**.
- El manejo de memoria dinámica se hace usando punteros.

```
int *arrayDimanico;
float **matrizDinamica= NULL;
```

- **Reserva de memoria: `malloc`**

```
void * malloc (unsigned tamano)
```

  - Reserva `tamano` bytes de memoria y devuelve un puntero a la zona reservada.

```
arrayDinamico = (int *) malloc (100*sizeof(int));
```

  - Devuelve `NULL` si no se ha podido reservar esa cantidad.
  - No se inicializa la memoria.

- **Reserva e inicialización de memoria: `calloc`**

```
void * calloc (unsigned ndatos, unsigned size)
```

  - Reserva `ndatos*size` bytes de memoria y devuelve un puntero a la zona reservada. Se supone que `size` es el tamaño del tipo de datos a escribir y `ndatos` el número de datos.

```
arrayDinamico = (int *) calloc (100, sizeof(int));
```

  - Devuelve `NULL` si no se ha podido reservar esa cantidad.
  - Se inicializa la memoria reservada con 0.

- **Relocalizar de memoria: `realloc`**

```
void * realloc (void *ptr, unsigned tamano)
```

  - Redimensiona la cantidad de memoria reservada previamente a la nueva cantidad de bytes dada en `tamano`.
  - Si el nuevo tamaño es mayor, espacio nuevo no se inicializa.
  - Devuelve NULL si no se pudo satisfacer la demanda.

- **Liberar memoria: `free`**

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

  - Libera una zona de memoria reservada antes. No usar si `ptr == NULL`.
  - OJO: no hay recolector de basura; liberar la memoria reservadada cuando ya no se vaya a usar. Buena costumbre: “liberar donde se reserva”.
  - No hacer free sobre puntero no inicializado o ya liberado.
  - No basta con hacer `ptr=NULL`: eso haría que la memoria se pierda.

- **Ejemplo**.

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

char *memoria= NULL;

int main() {
  char c;
  long i, tam= 0;
  do {
    tam+= 1000000;
    memoria= (char *) malloc (tam);
    if (!memoria) {
      printf("Error. Imposible reservar más memoria.\n");
      return 0;
    }
    printf("Reservados %ld bytes...\n", tam);
    for (i= 0; i<tam; i++)
      memoria[i]= (char) i;
    printf("Pulse s para seguir saturando la máquina.\n");
    while ((c= getchar())=='\n');
    free(memoria);
  } while (c=='s' || c=='S');
  return 0;
}
```

- **Otro ejemplo:** páginas 310 y 311 del texto guía.

IMPORTANTE: No acceder a lo apuntado por un puntero:
- antes de que apunte a una dirección válida (variable estática, local, o de memoria dinámica reservada previamente)
- después de haber liberado la memoria a la que apuntaba
- después de haber asignado NULL al puntero

3. Funciones de E/S

- No forman parte del lenguaje, están en librería estándar: **`#include <stdio.h>`**

- **Escribir por pantalla**: **`printf`**

```
printf ( <cadena> );
printf ( <cadena con formato>, <expresión 1>, <expresión 2>, ...);

printf("¡Buenos días!");
printf("Un entero: %d y un real: %f\n", 27*12, 3.1416);
```

En lugar de

```
System.out.println("Un entero:" + 27*12 + "y un real:" +3.1416");
```

- Especificación de **formato** en `printf`:

```
% [flags] [width] [.prec] [h|l|L] caracter_tipo
```

- Caracteres de **tipo**:

d, i (decimal), o (octal), u (unsigned), x, X (hexadecimal), f (float con la forma [-]dddd.dddd), e, E, (float con la forma [-]d.dddd e[+/-]ddd), g, G (float "inteligente"), c (char), s (string), p (puntero).

- **`sprintf`**. Igual que `printf`, pero en lugar de escribir en pantalla escribe el resultado en una cadena (el primer parámetro).

```
char resultado[100];
sprintf(resultado, "Probando sprintf con el número %u.\n", -1);
printf(resultado);
```

- **Leer de teclado: `scanf`** (en lugar de `BufferedReader, InputStreamReader…`)
  Formato parecido a la función `printf`.

```
int scanf (<cadena con huecos>, &<var1>, &<var2>, ...);
```

```
int i, j, n;
float r;
char cadena[20];

n= scanf("%d", &i);          → Lee un entero y lo guarda en i
n= scanf("%d  %f", &j, &r);  → Lee un entero (en j) y un real (en r)
n= scanf("%s", cadena);      → Lee una cadena (hasta el primer espacio)
```

  El valor devuelto por `scanf` indica el **número de huecos leídos**.
  Los especificadores de formato son los mismos que para `printf`.

- **`sscanf`**. El equivalente a `scanf`, pero en lugar de leer de teclado lee los caracteres de una cadena (el primer parámetro).

```
char entrada_simulada[100]= " 26  88.3  92";
int a, b;
float g;
sscanf(entrada_simulada, "%d %f %d", &a, &g, &b);
```

- **`putchar(char c)`**. Escribe un carácter por pantalla. Equivale a:
  `printf("%c", c)`
- **`char getchar(void)`**. Lee un carácter de teclado. Equivale a:
  `(scanf("%c", &c), c)`

- **Ejemplo.**

```
#include <stdio.h>

main() {
   int i, n;
   float f;
   char cadena[30];

   printf("Introduce un entero, un float y una cadena:");
            /* Probar distintas posibilidades en la siguiente línea */
   n= scanf("%d %f %s", &i, &f, cadena);
            /* Adaptando también este printf, claro */
   printf("Huecos leídos: %d\nValores: %d, %f, %s \n",n,i,f,cadena);
}
```

- **Ojo:** normalmente la salida estándar es la pantalla y la entrada estándar el teclado. Pero se puede utilizar redirección de entrada y salida.

```
>> gcc ejer1.c -o ejer1.out   → Compilamos el programa de ejemplo
>> ./ejer1.out               → Ejecutamos. E/S: teclado/pantalla
>> ./ejer1.out > res.txt     → Ahora la salida va al fichero res.txt
>> ./ejer1.out < in.txt      → Aquí la entrada está en el fich. in.txt
>> ./ejer1.out >res.txt <in.txt
```

La entrada estándar, salida estándar y salida de error estándar también se pueden manejar como ficheros (con los nombres **`stdin`**, **`stdout`** y **`stderr`**) con las funciones que veremos a continuación.

Ejercicios

1. Haz un programa que reciba n+2 argumentos en la línea de comandos, con n<=5. Los dos primeros son nombres de ficheros de texto, los n restantes palabras. La salida será el número de veces que aparece cada una de esas palabra en el fichero de texto correspondiente al primer argumento. Esa información se escribirá en un fichero de texto cuyo nombre vendrá dado por el segundo argumento.

2. Escribe un procedimiento que reserve memoria para una matriz de reales de tamaño **a×b**, y otro que multiplique dos matrices de dimensiones compatibles.

3. Encuentra y corrige el error existente en el siguiente programa. Antes de ejecutarlo deduce el resultado que tendrá.

```
#include <stdio.h>

#define CUADRADO1 (A) A*A
#define CUADRADO2(A) (A)*(A)
#define CUADRADO3(A) ((A)= (A)*(A))

int main (void) {
  int i= 3, j= 4;
  printf("CUADRADO1(%d+1)= %d\n", i, CUADRADO1(i+1));
  printf("CUADRADO2(%d+1)= %d\n", i, CUADRADO2(i+1));
  CUADRADO3 (j);
  printf("CUADRADO3(%d)= %d\n", j, CUADRADO3(j));
  return 0;
}
```