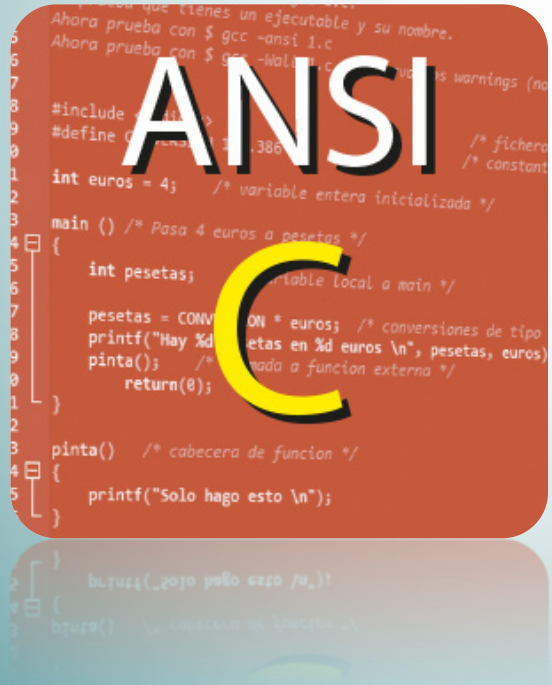
Programación Estructurada en ANSI C

Sesión 3



Rafael Menéndez de Llano Rozas

DEPARTAMENTO DE INFORMÁTICA Y ELECTRÓNICA

Índice

3. Funciones: definición

- La programación estructurada se basa en usar solamente sentencias secuenciales (de asignación o E/S), condicionales e iterativas.
- Además ese código debe ser **dividido** en subrutinas construidas con ese principio. En C (ANSI C) estas subrutinas se llaman funciones.
- Tres partes importantes:

  1. La definición. La función puede tomar una serie de argumentos y devolver un dato (por defecto el tipo es int, pero siempre está bien ponerlo):

```
tipo_ a_devolver     nombre_funcion   (declaración de argumentos, …)
{
     declaración de variables locales;

     Sentencias;

     return();          /* el valor devuelto opcionalmente */
}
```

3. Funciones: uso

2. La declaración. Siempre es conveniente declarar las funciones que se van a usar (llamar) en otra función. Si la definición se hace después además es obligatorio. Recomendación: USA SIEMPRE PROTOTIPOS.

La declaración de la función se hace con prototipos de función que son casi la primera línea de la función:

```
tipo_funcion    nombre_funcion    (tipos de argumentos);
```

3. El uso. Una vez declarada, se puede llamar asignando el valor que devuelve (a través del **return**) a una variable del mismo tipo (salvo tipo void):

```
variable = nombre_funcion (argumentos, …);
```

3. Funciones: uso

```
#include <stdio.h>       /* definicion de cabecera */
int main ()
{
      void asteriscos(void);     /* declaracion */

      asteriscos ();      /* llamada */
      printf("Aqui pongo un nombre\n");
      asteriscos ();      /* llamada */
      return(0);
}

void asteriscos (void)
{                         /* cuerpo */
      int cont;                                  // variable local

      for (cont = 1; cont < 40; cont ++)
            putchar('*');
      putchar('\n');
}
```

2. PROTOTIPO

3. LLAMADA

1. DEFINICIÓN

Se pueden cambiar de orden

3. Funciones: peculiaridades

- Se pueden crear **prototipos** de tres maneras:
  - Sólo lo que devuelve, ejemplo: float funcion ();
  - Con tipos de argumentos, ejemplo: float funcion (int , float);
  - Hasta con nombres, ejemplo: float función (int i, float f);
- Si no se devuelve o no se le pasa nada es **void**.
- El **orden** de las funciones en C no está definido pero se suele poner la función *main* al principio. Eso nos obliga a usar prototipos.
- Pueden ser llamadas desde cualquier lugar.
- La función *main* no se llama (pero nada lo impide).
- No se pueden **anidar**.
- Una función se puede llamar a sí misma: recursión.

Ejercicio 14: Funciones

- Bájate los programas 17.c, 18.c, 19.c y 20.c guárdalos en tu directorio.
  1. Ábrelos con un editor, examínalos y comprende las funciones y su uso.
  2. Abre el 17 y comenta el prototipo ¿qué pasa?
  3. Edita el programa, pasa la función asteriscos antes de main. ¿qué pasa?
  4. Comprueba realmente hasta que argumento puedes llegar, dependiendo del tamaño del dato long en 18.c
  5. Entiende la diferencia entre un programa iterativo y uno recursivo.
  6. Observa que el programa 20 no tiene fin.

3. Funciones: Argumentos y punteros

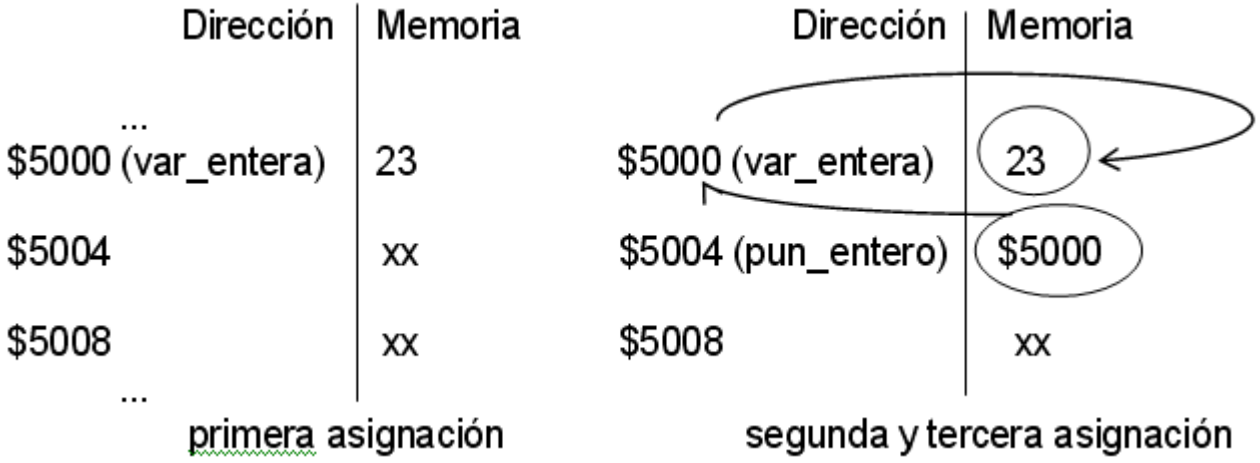
- Los argumentos a una función se pueden pasar de varias formas:
  - **Por valor**: En la función se copia el valor actual (copia parámetro actual al formal) del argumento y los cambios no tienen repercusión fuera de la función.
    - **Constante**: Con la palabra reservada `const` indicamos explícitamente que no se debe cambiar.
  - **Por referencia**: Se quiere que la función cambie el argumento con que se la ha llamado.
    - Para ello en C se utiliza un tipo de dato que contiene direcciones y se llama puntero.
    - Existen tantos tipos de puntero como tipos de variables.
    - Para declararlos se utiliza "*":

```
int *entero;
float *real;
int variable;
```

3. Punteros

- Para trabajar con punteros existen dos operadores unarios:
  - Dirección: &.
  - Indirección: *.
- Además los operadores asignación, suma y resta también se pueden usar (avanzarán o retrocederán posiciones en memoria de acuerdo al tipo de datos apuntado).

```
var_entera = 23;              /* almacenado en $5000 */
pun_entero = &var_entera;     /* almacenado en $5004 */
var_entera = *pun_entero;     /* equivalente a la primera */
```



3. Funciones: Paso por referencia

- Ejemplo de paso por referencia:

```
void intercambia(int *u, int *v)
{
   int temp;
   /* lo apuntado por u se pone en temp */
   temp = *u;
   /* lo apuntado por v se pone en lo apuntado por u */
   *u = *v;
   /* el valor de temp se pone en la dirección v */
   *v = temp;
}
```

- Dos cosas muy importantes a tener en cuenta con los punteros:

1. Siempre se usan inicializados:
   1. Con la dirección de una variable.
   2. Con un array.
   3. De forma dinámica (malloc y calloc) (equivalente a **new**).
2. Con suficiente espacio.

Ejercicio 15: Funciones

- Bájate los programas 21.c y guárdalo en tu directorio.
  1. Ábrelo con un editor, examínalo y comprende las funciones y su uso.
  2. Observa que el programa 21 no tiene *main*.
  3. Crea uno para que intercambia dos valores de enteros.
  4. Cambia la función para que uno de los argumentos sea constante, compila e indica lo que pasa.
     - Usando la palabra reservada const

3. Variables: ámbito (scope)

- Existen varios tipos de ámbito de variables:
  - **Globales**: Se declaran fuera de las funciones y se conocen después de declararlas.
    - Normalmente es una **mala** práctica de programación hacer uso intenso de variables globales, ya que siempre pueden ser cambiadas inadvertidamente desde cualquier función.
    - Una forma aceptable de hacerlo es declarándolas de tipo ***extern*** en las funciones, con lo cual se explicita que es global.

      ```
      extern int varible_global;   //por ejemplo una entero
      ```
  - **Locales**: Se declaran dentro de las funciones y sólo se conocen allí. Incluido el main. Pueden ocultar a las globales.
  - **Locales a un bloque**. Sólo se conocerá en el bloque ({ }) donde se han declarado.
- Además se pueden hacer constantes con *const*:

  ```
  const float pi = 3.1416;
  ```

3. Variables: Almacenamiento

- Al ser el C de medio nivel, el compilador puede intentar controlar el modo de almacenamiento de las variables:
  - Automáticas: "**auto**". Por defecto. No se suele poner.
  - Externas: "**extern**". Es una forma explícita de declarar que se va a usar una variable global. ¡ No se crea !
  - Estáticas: "**static**". Permanecen durante toda la ejecución del programa.
    - Cuando una variable es local, es almacenada en el *stack* y cuando esa función termina desaparece.
    - Con *static* se logra que una variable permanezca durante toda la ejecución del programa.
  - Registros: "**register**". Se "intentarán" localizar en algún registro de la CPU.

Ejercicio 16: Variables

- Bájate los programas 22.c y guárdalo en tu directorio.
  1. Ábrelos con un editor, examínalo y comprende las variables, su ámbito y su almacenamiento.
  2. Haz una predicción de los valores que aparecerán en pantalla.
  3. Ejecútalo y comprueba que lo has entendido.

3. Variables

```
void funcion (int arg1, int arg2)   // los argumentos con otros nombres
{
     auto char local1;   /* no seria necesario el auto */
     int  local2;                   /* variable local a funcion */
     int global = 54;    // oculta la global

     local1 = '@';
     local2 = 32;
     printf("\tdesde dentro valores de 1 2 y global:  %c  %d  %d\n", local1, local2, global);
     {       //bloque
             int local2; /* locales al nuevo ambito */
             register int local3;

             local2 = 15;   /* nos referimos a la interior */
             local3 = 4;
             printf("\t\tdesde muy dentro valores de 1 2 y 3:  %c  %d  %d\n", local1, local2, local3);
     }
     printf("\tdesde dentro valores de 1 2 y global:  %c  %d  %d\n", local1, local2, global);
     /* local2 valdra 32 y no 15, ya que es la de fuera */
     /* Esto no se puede hacer, ya que aqui local3 no existe y daria errores de compilacion*/
     // local3 = 232;
     arg1 = 345; arg2 = 35;   // no cambian fuera
     printf("\tdesde dentro valores de uno  otro  global:  %d  %d   %d\n", arg1, arg2, global);
}
```

3. Funciones: main

- En C existen muchas funciones ya realizadas que se conocen como la librería C estándar. Mira apuntes.
- En C es posible definir funciones con un tipo variable de argumentos. Ej.: *printf*.
- A la función *main* se le pueden pasar tres argumentos que hasta ahora no hemos usado:
  - `int argc;  // entero con el numero de argumentos pasados`
  - `char ** argv;  // array con nombres de los argumentos`
- Incluso existe un tercero que nos da las opciones de la shell dentro del programa:
  - `char ** opciones;`

Observa el ejemplo entorno.c