Introducción al lenguaje C

4. Funciones

Un programa C va a estar compuesto de una o más funciones repartidas en uno o más módulos de código. Como ya se vio, en un programa en C al menos siempre existirá una función `main` que será el punto de entrada al programa.

4.1. Declaración y definición de funciones

En C, se hace distinción entre una declaración y una definición. Una declaración, simplemente indica al programa que se va a utilizar un identificador y cómo se va a utilizar. Una definición indica al compilador cómo se va a almacenar un identificador.

Cuando se declara una variable, por ejemplo con

```
double total;
```

se está haciendo al mismo tiempo una declaración y una definición: se indica al programa que se va a utilizar una variable `total` como un dato de tipo `double` y, al mismo tiempo, se está reservando espacio en memoria para almacenar un dato de tipo `double`.

Sin embargo, cuando se trata de funciones se considera diferente la declaración de la función de la definición de la misma.

4.1.2. Definición de funciones

El formato de definición de una función sería el siguiente:

```
tipoDato nombreFunción([listaParámetrosFormales]){
   //Cuerpo de la función
   …
   [return expresiónDeRetorno]
}
```

Una función devuelve un dato, por lo que será necesario indicar el tipo de dato que devuelve. *`tipoDato`* podrá contener alguno de los tipos de datos utilizados para la declaración de variables y que aparecen en la tabla 1

Tabla 1. Tipos de datos

Especificador de tipo
`char`
`signed char`
`unsigned char`
`short, signed short, short int,` o `signed short int`
`unsigned short, o unsigned short int`
`int, signed, or signed int`
`unsigned, or unsigned int`
`long, signed long, long int,` o `signed long int`

BY NC SA

Especificador de tipo
`unsigned long, or unsigned long int`
`long long, signed long long, long long int,` o `signed long long int`
`unsigned long long,` o `unsigned long long int`
`float`
`double`
`long double`

En algunos casos la función no devolverá un valor. En esos casos se utiliza la palabra reservada **`void`** como tipo de la función para indicar que el valor de retorno de la función no está requerido, como ocurre en los procedimientos.

El *`nombreFunción`* será un identificador válido. Normalmente, una función devolverá un valor a partir de los datos que se le pasan como argumentos. En la cabecera de la declaración deberán constar el nombre de los argumentos y el tipo de cada uno de ellos en la *`listaParámetrosFormales`*. Si una función no tiene parámetros, en su llamada se incluirá un parámetro **`void`** para indicar que se trata de un dato no requerido.

Dentro del cuerpo de la función podrá aparecer la sentencia **`return`**. **`return`** indicará cuál es el valor de retorno de la función. Por ejemplo, en la siguiente función:

```
int funcion1(void){
   return 2;
}
```

se devolverá el valor entero 2.

Ejemplo: Función `Factorial`

```
int factorial(int n){
   int f=1;

   for(int i=2;i<=n;i++)
      f *= i;
   return f;
}
```

La función `factorial` recibe un argumento entero (`n`) y devuelve el acumulador de multiplicaciones `f` que, al final del proceso, guardará el factorial de `n`.

La llamada a la función `factorial` se podría hacer…

```
printf("Factorial de 5: %i",factorial(5));
b = factorial(a+3);
scanf("%i",n);
printf("Factorial de 5: %i",factorial(n));
c = factorial(factorial(3));
factorial(6);
```

La última llamada devolvería el valor 720. En C no sería sintácticamente incorrecta, pero no tendría ningún efecto en el programa.

La función main

La función `main` de un programa C es una función como cualquier otra. Su formato estándar es:

```
int main(void){
   …
   return 0;
}
```

Esto indica que se trata de una función que devuelve un argumento de tipo entero y que no recibe parámetros. Normalmente no se indica el valor de retorno, aunque es posible ponerlo. El valor de la expresión de retorno sería el que la función devolvería al sistema operativo.

Existe otro formato de la función `main`.

```
int main(int argc,char *argv[]){
   ...
   return 0;
}
```

En este caso, la función recibe dos argumentos, un argumento entero (`argc`) con el número de parámetros que se le pasan y array de cadenas (`argv`) con cada uno de los argumentos que se han pasado. En ambos casos el nombre del programa al que se llama se trataría también de un argumento.

Por ejemplo, en el siguiente programa:

```
int main(int argc, char *argv[]){
    int i;
    printf("Nombre del programa: %s\n",argv[0]);
    printf("Primer argumento: %s\n",argv[1]);
    printf("Segundo argumento: %s\n",argv[2]);
    return 0;
}
```

La llamada `miprograma hola adios`

`argc` se cargaría con el valor 3 y `argv` tendría tres elementos: el propio nombre del programa (en el elemenento 0, al que se accede con `argv[0]`), la cadena “hola” (en el elemento 1 al que se accede por `argv[1]`) y la cadena “adios” (en el elemento 2 al que se accede por `argv[2]`).

4.1.3. Declaración de funciones: prototipos de funciones

Cuando se define una función se indica, además del nombre, tanto el tipo de dato que devuelve como el número, orden y tipo de argumentos. Sin embargo el compilador de C no comprueba si el tipo de la función o el tipo, número y posición de los argumentos coinciden. Por ejemplo si tenemos definida la siguiente función…

```
int f1(int a, int b){
    return a+b;
}
```

y se la llama con un número de argumentos distinto,

```
c=f1(3);
```

el compilador no detecta ningún error sintáctico, pero, sin embargo, los resultados serán incongruentes ya que en la llamada falta un argumento entero.

Para evitar esto, es conveniente declarar previamente la función para indicar al compilador la existencia de una función de un tipo determinado, con un nombre concreto y unos argumentos precisos. A esta declaración se le llama el **prototipo de la función** y advierte al compilador de la existencia de la función. Si añadimos un prototipo de la función y se realiza una llamada errónea, el compilador dará un error.

El prototipo de la función se declara de la siguiente forma:

```
tipoDato nombre(tipoArgumento1, tipoArgumento2,…tipoArgumentoN);
```

Los prototipos de funciones se declaran en el módulo de código principal fuera de cualquier función y antes de realizar la primera llamada a la función. Normalmente se agruparán todos los prototipos de funciones al comienzo del archivo fuente, después de las directivas del pre-procesador.

En el siguiente archivo fuente…

```
#include <stdio.h>

int f1(int,int); //Prototipo de la función

int main(void){
   //Llamada a la función con un solo argumento
   printf("%i",f1(3));
}
```

El compilador detectará que la llamada a la función `f1` sólo presenta un argumento entero, mientras que el prototipo requiere dos enteros, por lo que se generará un error de compilación. Es preferible que el compilador detecte un error sintáctico que intentar averiguar por qué los resultados no son los esperados, por lo que siempre que se utilicen funciones es conveniente definir su prototipo.

4.1.4. Funciones que no devuelven ningún valor

Aunque en C podemos hacer caso omiso del valor que devuelve una función, también es posible definir funciones que no devuelven ningún valor. El tipo de dato **`void`** se utiliza para referenciar un tipo de dato en aquellos casos en los que no está definido (ver más abajo la función `contador`).

También es posible definir funciones que no devuelven ningún valor (procedimientos) indicando que el tipo de dato devuelto es **`void`**. Por ejemplo, una función que escriba un número al revés.

```
void escribirNumeroAlReves(int n){
```

```
        while(n >= 10){
              printf("%i",n % 10);
              n = n / 10;
        }
        printf("%i\n",n);

}
```

La función no devuelve ningún valor, simplemente se limita a realizar una acción: escribir un número al revés

4.2. Ámbito o alcance de las variables

El ámbito o alcance de un identificador es la parte del programa dónde se puede utilizar, es decir, la parte del programa desde dónde es visible. En C podemos distinguir tres tipos de ámbitos:

- Global. Su ámbito es el archivo fuente dónde han sido definidas. Se trata de variables o funciones externas que se definen en el archivo fuente fuera de cualquier función y antes de su definición.
- Local. Su ámbito es el bloque dónde han sido definidas. Se trata de aquellas variables definidas dentro de un bloque, es decir entre los símbolos { y }. Las variables declaradas dentro de una función, serán variables locales.
- Ámbito del prototipo. Cuando se declaran los argumentos en un prototipo de función, su ámbito queda reducido a la declaración del propio prototipo.

4.2.1. Variables externas

Las variables externas se definen fuera de cualquier función y estarán disponibles desde cualquier función. En C todas las funciones son externas ya que no es posible definir una función dentro de otra.

El ámbito de una variable externa abarca desde el punto en que se declara hasta el final del archivo que se está compilando. Por ejemplo, en este fragmento de código…

```
int main(void){
   a = f1(3);
   f2(4);
}
int a = 0;        //definición de la variable a

int f1(double b){
   return a;
}
int f2(double c){
   return a;
}
```

a la variable `a` se puede acceder desde `f1` o desde `f2`, pero no desde `main`. El compilador detectaría un error, ya que la variable `a` no ha sido declarada cuando se ha utilizado en la función `main`.

Si se quiere utilizar una variable o función externa desde otro módulo de código o si se va a hacer uso de ellas antes de su definición, es necesario utilizar la declaración con la palabra reservada **`extern`**.

Es importante distinguir entre la declaración de una variable externa y su definición. Una *declaración* indica al compilador que se va a utilizar una variable (sobre todo su tipo de dato), pero no reserva espacio para almacenarla. Sería el equivalente de los prototipos de funciones aplicado a variables. Una *definición* provoca que el compilador reserve espacio para almacenar la variable. En el siguiente fragmento de código…

```
extern int a;      //declaración de la variable a

int main(void){
   a = f1(3);
   f2(4);
}
int a = 0;         //definición de la variable a

int f1(double b){
   return a;
}
int f2(double c){
   return a;
}
```

el compilador no detectaría ningún error, ya que antes de utilizar la variable `a` en la función `main`, se ha declarado mediante **`extern`**. Esa variable `a` podría ser utilizada desde cualquier parte del programa, incluso desde fuera del archivo fuente dónde ha sido declarada. Es importante indicar que no podría haber más declaraciones de dicha variable en ninguno de los archivos fuente del programa.

Suponiendo que un programa tiene el código repartido entre dos programas fuente, `main.c` y `funciones.c`.

```
/*main.c*/
#include <stdio.h>

extern int i; //Declaración de la variable i

int main(void)
{
  printf("%i\n",f1());
  printf("%i\n",i);
  return 0;
}

/*funciones.c*/
```

[CC BY-NC-SA]

```
int i;  //Definición de la variable i

int f1(void){
    i = 5;
    return i;
}
```

La variable `i` se declara en el módulo `main.c` como externa, pero todavía no se ha hecho su declaración (no se ha reservado espacio en memoria). La declaración se hace en el módulo `funciones.c`, dónde se reserva espacio en memoria. La declaración **`extern`** hace que la variable pueda utilizarse en cualquier módulo del programa, por lo que la `i` que aparece en ambos módulos de código es la misma.

También es posible indicar que la definición de una variable externa no sea accesible desde otros módulos. Si al declarar una variable se utiliza la palabra **`static`**, el ámbito de esa variable será únicamente el módulo dónde ha sido definida, de forma que el identificador podrá utilizarse fuera del módulo.

En el caso de las funciones, ya se ha hablado sobre la conveniencia de utilizar los prototipos de funciones. Sin embargo, los prototipos de funciones sólo serían válidos para el archivo fuente dónde han sido declarados. Si se desea que el prototipo sirva para todos los archivos fuente del programa, también habría que declararlos con **`extern`**.

4.2.2. Variables locales

Las variables declaradas dentro de una función son variables locales. Esto incluye tanto los argumentos de la función, como aquellas variables declaradas dentro de las llaves que definen la función.

En general, el ámbito de una variable declarada dentro de un bloque será el bloque dónde ha sido definida. Esto incluye cualquier bloque de código, es decir, cualquier grupo de instrucciones encerrados entre llaves. También incluye las variables declaradas dentro de un bucle **`for`**.

En el siguiente fragmento de código…

```
int variableExterna;

int funcion1(int arg1, int arg2){
   int variableLocal1=0;

   for(int i=0;i<=5;i++){
      variableLocal1 +=i;
   do{
      int i = 2;
      variableLocal1 -= i;
   }while(variableLocal1 > 0)
}
```

- `variableExterna1` es una variable global y se puede utilizar desde cualquier punto del archivo fuente.

- Los argumentos `arg1` y `arg2` son declaraciones locales y sólo se pueden utilizar desde cualquier punto de `funcion1`.
- `variableLocal1` se ha definido dentro de `funcion1` y se puede utilizar desde cualquier parte de `funcion1`.
- `i` sería una variable de bloque, que sólo se podría utilizar dentro del bucle **`for`**.
- La variable `i` declarada dentro de la declaración **`do`**…**`while`** es una variable de bloque y sólo se podría utilizar dentro del bloque **`do`**…**`while`** dónde se ha declarado. Hay que indicar que esta variable `i` sería distinta de la variable `i` utilizada en el bucle **`for`**.

4.2.3. Tiempo de vida de una variable

El tiempo de vida o duración de un objeto indica el tiempo en el que el objeto está activo. Desde ese punto de vista, un objeto (una variable) puede ser:

- Permanente. Su tiempo de vida es la duración del programa. Las variables externas son permanentes ya que tienen un ámbito global. También son permanentes las variables locales marcadas con la palabra reservada **`static`**. Una variable local **`static`** se inicializa la primera vez que se llama a la función, pero su valor permanece entre llamadas, aunque su ámbito sea local. Con la siguiente función:

  ```
  int contador(void){
      static int i=0;
      i++;
      return i;
  }
  ```

  La primera llamada a la función `contador` hará que la variable `i` valga 1, pero en las siguientes llamadas valdrá 2, 3, 4, etc.

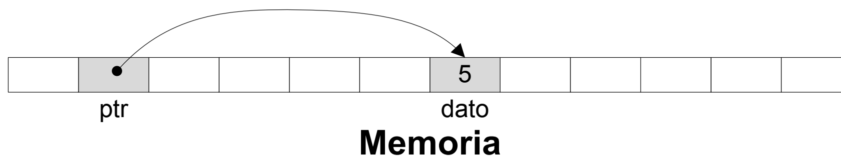
- Transitorio. Su tiempo de vida es la duración del ámbito dónde ha sido declarado. Las variables locales (también llamadas variables automáticas en contraposición a las variables locales **`static`** del punto anterior) tienen una duración transitoria.

4.3. Paso de argumentos

En C los argumentos de una función se pasan por valor, por lo que no existe una forma directa de que una función modifique una variable que se le pasa como argumento. Para realizar esto hay que utilizar punteros: enviar como parámetros actuales la dirección de memoria dónde se aloja la variable y referirse a ella como el objeto que se almacena en dicha dirección de memoria dentro de la función.

4.3.1. Punteros

Un puntero es una variable que contiene la dirección de memoria dónde está almacenado un objeto. Si `ptr` es un puntero, contendrá la dirección de memoria donde se almacena otro dato, por ejemplo, la variable `dato`.



Cuando se utilizan punteros, el operador unario `&` devuelve la dirección de memoria de un objeto, de forma que `ptr = &dato;` guardaría en la variable `ptr` la dirección de memoria de la variable `dato` (tal y como aparece en la figura).

El operador de *indirección* o *desreferencia* es el operador unario `*` y devuelve el contenido del objeto referenciado, es decir, el contenido de la dirección de memoria a la que apunta el puntero. De esta forma, `printf("%i",*ptr);` devolvería el contenido de `dato`, es decir, un 5.

La declaración de las variables `ptr` y `dato`, se haría de la siguiente forma:

```
int dato = 5;
int *ptr;
```

Con esto se está declarando a la variable `ptr` como un puntero capacitado para apuntar a una variable de tipo entero.

4.3.2. Paso de argumentos por referencia

Cuando se hace el paso de argumentos por valor, lo que se pasa es una copia del valor del parámetro actual y el parámetro formal se carga con esa copia. Al tratarse de una copia, cualquier modificación se realizará sobre la copia, sin alterar el parámetro actual.

Cuando se hace el paso de argumentos por referencia, lo que se pasa es la referencia de una variable, es decir, la dirección dónde está almacenada. De esta forma, el parámetro formal se carga con la dirección de la variable, por lo que cualquier modificación que se haga sobre el argumento, se estará haciendo directamente sobre la dirección de memoria del parámetro actual modificando su valor. Puesto que lo que se pasa es la dirección de la variable (un puntero) los argumentos deberán de ser también punteros. Para acceder al contenido de la dirección de memoria habrá que utilizar el operador de indirección.

Por ejemplo, si se quiere hacer un procedimiento (una función **`void`**) que intercambie el contenido de dos variables enteras, los parámetros formales tendrán que intercambiar su valor, por lo que el paso de argumentos se deberá hacer por referencia.

```
…
int a = 5;
int b = 8;
intercambiarEnteros(&a,&b);
…
```

Los parámetros actuales del procedimiento `intercambiarEnteros` son dos referencias, es decir, la referencia a la dirección de memoria dónde están almacenadas las variables `a` y `b`.

Si los parámetros actuales guardan la dirección de memoria de dos variables enteras, en el procedimiento `intercambiarEnteros`, los parámetros formales deberán ser dos punteros a entero.

```
void intercambiarEnteros(int *x, int *y){
   int aux;
   aux = *x;
   *x = *y;
   *y = aux;
}
```

Dentro de la función, para hacer referencia al contenido al que apuntan `x` e `y`, habrá que usar el operador de indirección.

5. Ejercicios

Nota: Hay que tener en cuenta que para probar estos ejercicios, se deberá hacer también un programa principal que realice la llamada

1. Diseñe una función lógica que indique si una fecha es válida y otra que indique si un año es bisiesto. Para la primera se pasará a la función tres números enteros con el día, el mes y el año. Habrá que comprobar si el mes está entre 1 y 12 y el día entre 1 y 31. Si esto es cierto habrá que comprobar que si el mes es de 30 días, el día no sea mayor que 30. Si se trata de febrero habrá que comprobar si es un año bisiesto, y si es así comprobar que el día no sea mayor que 29 o, en caso contrario, mayor que 28. Para la segunda se comprobará si el año que se pasa como argumento es bisiesto. Un año es bisiesto si es divisible por cuatro, a excepción de las centenas que no sean divisibles por 400.
2. Diseñe un subprograma que reciba la posición de un punto en coordenadas polares (radio y ángulo) y devuelva su posición en coordenadas cartesianas (x,y). Para pasar de coordenadas cartesianas a polares se utilizan las siguientes fórmulas.

   x = radio * cos(a)

   y = radio * sen(a)

3. El número Π se puede calcular mediante la serie de Leibnitz:

   $$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \dots \pm \frac{1}{2n-1}\right)$$

   Calcule el valor del número hasta que la diferencia entre dos términos sea menor a 10^{-15}.
4. Diseñe una función lógica que determine si un número entero de 4 cifras que se pasa como argumento es capicúa.
5. Suponiendo que nuestro lenguaje de programación no dispone de los operadores de división entera y resto, diseñe un módulo que permita obtener la división entera y el resto

de dos números enteros positivos mediante restas sucesivas. Justifique la elección del tipo de módulo utilizado.

6. El coseno de un ángulo x se puede calcular mediante la serie

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

   Diseñe una función que permita calcular el valor del coseno de x hasta que la diferencia entre dos términos sea menor que 10^{-3}.

7. Se desea convertir una longitud expresada en metros a pies y pulgadas, sabiendo que un metro son 39,37 pulgadas y que 12 pulgadas son un pie. Codifique un subprograma que reciba una longitud en metros y devuelva el número de pies y de pulgadas que son.
8. Diseñe una función que devuelva la suma de divisores de un número pasado como argumento. Por ejemplo, si el argumento pasado es 24, deberá devolver 1+2+3+4+6+8+12+24.
9. Diseñe una función lógica que indique si un número es perfecto. Un número es perfecto si es igual a la suma de sus divisores. Por ejemplo, 6 es un número perfecto ya que 6 = 1+2+3.
10. Diseñe una función lógica que indique si dos números son amigos. Dos números son amigos si cada uno es igual a la suma de divisores del otro excepto el propio número. Por ejemplo, 284 y 220 son amigos:

    Divisores de 284 1, 2, 4, 71, 142 y 284.
    Suma de divisores excepto 284 = 220.
    Divisores de 220: 1, 2, 4, 5, 10, 11, 20, 22, 44, 55, 110 y 220.
    Suma de divisores excepto 220 = 284

    Diseñe además un programa principal que saque por pantalla todas las parejas de números amigos entre 1 y n.
11. Diseñe una función que devuelva el máximo común divisor por el algoritmo de Euclides.
12. Codifique un procedimiento permita leer por teclado una serie de caracteres. La lectura terminará cuando el carácter introducido sea ‘0’ (cero). El procedimiento deberá devolver al programa que lo llamó el número de vocales que se introdujeron. Por ejemplo, si la serie de caracteres introducida es ‘a’, ‘x’, ‘h’, ‘e’, ‘a’, ‘v’, ‘0’, el procedimiento deberá devolver el valor 3.
13. Dada una fecha expresada en día, mes y año, escriba una función que devuelva el número de días transcurrido desde el comienzo del año.
14. Dada una fecha posterior al 1 de enero de 1980, escriba un módulo que devuelva el día de la semana sabiendo que el 1 de enero de 1980 fue martes.
15. Diseñe una función que reciba una hora expresada en horas, minutos y segundos y devuelva el número de segundos totales.
16. Diseñe un subprograma que reciba un número de segundos transcurridos y devuelva el número de horas, minutos y segundos que representan.
17. Diseñe un subprograma que reciba dos horas expresadas en horas, minutos y segundos y devuelva otra hora que represente la suma de ambas.