❑ Definición de funciones en C

CABECERA

- El ***tipo*** de valor que es devuelto por la función: Alguno de los tipos primitivos de C (void, char, int, float, double) con o sin modificadores (signed/unsigned,short/long), punteros y struct
- **Nombre** de la función
- Lista de los ***parámetros*** formales, indicando para cada uno su ***tipo*** y su ***nombre***, separados por coma: $tipo_1\ p_1, tipo_2\ p_2, \ldots, tipo_n\ p_n$

Por ejemplo:

```
double hipotenusa(float a, float b)
char pedirLetraInicial()  /*Sin parámetros*/
void mostrarDNI(int n, char l )/*Procedimiento*/
```

```
tipo identificador (lista de parámetros)
{
 Declaraciones de variables locales

Sentencias
return (expresión)
}
```

CUERPO

Excepto en funciones void, debe haber *al menos* una sentencia **return**(expresión):

- Devuelve un dato del tipo indicado en la cabecera.
- Devuelve el control al lugar donde se llamó a la función. ¡Cuidado!: Las sentencias que haya a continuación en la función, no se ejecutarán.

- ✓ Si una función no tiene argumentos, los paréntesis deben quedar vacíos.
- ✓ La manera de definir un procedimiento en C es definir una función cuyo tipo es **void** (nulo).
- ✓ No se pueden definir funciones dentro de un bloque (en particular no se pueden definir unas funciones dentro de otras), así que todas las funciones están definidas en un fichero 'al mismo nivel'.
- ✓ Se pueden definir variables al comienzo del cuerpo de una función (justo despues de '{').

  Estas variables son **locales**, lo que significa que solo existen mientras se esté ejecutando la función y además no son visibles (utilizables) fuera de la función

Ejemplo:

```
double hipotenusa(float a, float b)
{
 return(sqrt(a*a+b*b));
}

void trianguloRectangulo()
{
 float c1,c2,h;

 printf("¿Valor de los catetos?");
 scanf("%f %f",&c1,&c2);
 h=hipotenusa(c1,c2);
 printf("Catetos: %f,%f hipotenusa: %f",c1,c2,h);
}
```

❑ Declaración (prototipo) y definición

✓ **Declaración de una función (prototipo)**: cabecera, con el tipo de cada argumento, pero sin nombre.

Definición de una función: Cabecera y cuerpo.

✓ Antes de invocar a una función hay que declararla

⟹ En un fichero la declaración de una función se escribe antes de invocarla en el cuerpo de otra

⟹ Por esto la función main suele escribirse la última.

Es necesario escribir el prototipo de una función 'f' si queremos escribir el cuerpo de 'f' después de una función 'g', pero en el cuerpo de 'g' se invoca a 'f'. De hecho, es la única forma de definir funciones mutuamente recursivas

Ejemplo:

```
void f(int);        //declaración de f

void g ()
{
  . . .
   f(3) //Es necesario declarar f antes de invocarla
  . . .
}
void f(int n)       //definición de f
{
  . . .if (n!=3) g( );
}
```

❑ Llamada y salida de una función

✓Existen dos formas de **salir de una función**:

1. Llegar al final del cuerpo de la función (llave que cierra la función).

2. Ejecutando la sentencia **return**, lo que provoca la salida incondicional del cuerpo de la función.

   return puede terminar una función sin retornar ningún valor.

   En caso de que la función devuelva algún valor, la expresión o el valor debe ser colocado tras **return**.

   Las funciones de tipo **void** no pueden contener un **return** que devuelva un valor

✓Para **llamar a una función** se escribe en el sitio adecuado el nombre de la función seguido de parentesis abierto y cerrado. Si se pasan parámetros, estos deben colocarse dentro de los parentesis separados por coma.

Ejemplo:

```
#include <stdlib.h>

max(int x, y)
{
     return((x > y) ? x :  y);
}
main()
{
     int a = 10, b = 20;
     printf("%d\n", max(a, b));
}
```

❑ Paso de parámetros

- ✓ En C el paso de **parámetros a una función es siempre por valor,** por tanto los parámetros pasados a una función se guardan siempre en variables temporales, locales a la función de tal manera que si se llevan a cabo modificaciones en ellos, al salir de la función los cambios no se reflejaran en las variables originales.

  Sin embargo, C ofrece un mecanismo para conseguir que una función pueda modificar los parámetros que se le pasan. Eso mecanismo consiste en utilizar punteros.

- ✓ Un **puntero** es un tipo de dato que hace referencia a una dirección de memoria; por tanto el valor de una variable de tipo puntero se interpreta como una dirección de memoria. El tipo puntero siempre se refiere a uno de los tipos base definidos en C, así que se tiene puntero a carácter (char *), puntero a entero (int *), etc.

  Existen dos **operadores sobre punteros** que nos interesa destacar ahora:

  &: operador monario que aplicado a una variable cualquier permite obtener la dirección (un puntero pues) de esa variable

  *: operador monario que aplicado a una variable de tipo puntero permite acceder al contenido de la variable situada en la posición de memoria que indica la variable de tipo puntero

Ejemplo:

```
#include <stdlib.h>

main()
{
 int i=8;      //variable de tipo entero

 int * pi;  //variable de tipo puntero a entero

 pi = &i;  /*en la variable 'pi' se introduce la
           dirección en memoria de la variable 'i'*/

 j= *pi;  /*en 'j' se guarda el contenido de la
             posición de memoria cuya dirección
             está almacenada en 'pi'
           */

printf("%i",j); /*¿Qué valor se mostrará en
                   pantalla?
                 */
}
```

- ✓ Para **simular el paso de un parámetro por referencia**:

  1. Se declara en la cabecera de la función ese parámetro como un puntero al tipo correspondiente.

  2. En el cuerpo de la función, cuando se quiera hacer uso de ese parámetro se emplea el operador monario '*' (por ejemplo '*i')

  3. Al invocar a la función, se pasa una variable precedida del operador & (por ejemplo '&j');

Ejemplo:

```
#include <stdio.h>

void intercambio(int *,int *); //Intercambia valores


main()
{
    int a=1,b=2;
    printf("a=%d y b=%d",a,b);
    intercambio(&a,&b); /* <<<<<----- llamada */
    printf("a=%d y b=%d",a,b);
}
void intercambio (int *x,int *y)
{
    int aux;
    aux=*x;
    *x=*y;
    *y=aux;
    printf("a=%d y b=%d",*x,*y);
}
```

❑ Ambito, visibilidad y persistencia

Todo identificador (variable, función, macro) en un programa C posee tres propiedades: ámbito, visivilidad y peristencia

✓ Ambito o alcance

Región del código en el que la declaración del identificador está activa, es decir el objeto existe.

- Si la declaración está en un bloque de código ('**{**', '**}**'), el alcance es la región que va entre las llaves.

- Si la declaración está en la parte de arriba del fichero (normalmente) o en una parte que **no va entre llaves**, el alcance se establece en todo el fichero. Si va precedida de ***extern*** (ver a continuación) su alcance es todo el programa (el conjunto de todos los ficheros que lo componen)

- Los identificadores establecidos con la sentencia ***#define*** tienen alcance durante todo el fichero o hasta que lo elimina la sentencia ***#undef***.

✓ Visibilidad

Región del código en la que el objeto está activo.

La diferencia con el alcance es que en una misma región pueden estar definidos dos objetos con el mismo identificador, ocultando un objeto a otro.

Ejemplo:

```
{
 int a;
  . . .    /*Aquí la variable 'a' es de tipo int*/
{
  char a;
        /*Aquí la variable 'a' es de tipo char*/
}
 . . .   /*Aquí la variable 'a' es de tipo int*/
}
```

✓ Durabilidad

Tiempo, durante la ejecución del programa, en el cual el objeto existe en la memoria.
La durabilidad puede matizarse con los ***especificadores de clases de almacenamiento*** (**static, auto, extern, register**).

- **static**: El objeto perdura desde el comienzo de la ejecución del programa hasta su finalización. Esta durabilidad la tienen:

  - Todas las funciones declaradas
  - Las variables declaradas fuera del cuerpo de funciones (incluido *main*)
  - Variables declaradas con **static:** se crea un almacenamiento permanente

    - Dentro de un bloque (local): su ámbito es el bloque y se mantiene su valor entre llamadas (al terminar la ejecución de la función o bloque, su valor no desaparece)

      Se inicializa solo la primera vez que se llama a la función. Después el valor que se utiliza es siempre el anterior.

      Ejemplo:

```
int series () //Devuelve 23,46,…
{
 static int num; //Se inicializa a 0
 num=num+23;
 return num;
}
```

    - Fuera de un bloque (global): solo es conocida por el fichero donde ha sido declarada y no puede ser accedida por funciones declaradas en otros ficheros. Es una forma de ocultar información entre archivos.

- **auto**: El objeto es creado en la entrada de un bloque ({ }) y es borrado a la salida. La tienen:

  - Los argumentos formales

  - Las variables declaradas con auto (es la declaración por defecto y no se suele poner). Las variables declaradas en un bloque por tanto lo son (a menos que se especifique como static).

- **extern**: El objeto perdura desde el comienzo de la ejecución del programa hasta su finalización (como static), pero además especifica que el identificador ha sido definido en otro de los ficheros que componen el programa y no en este (por tanto debe existir un solo archivo en el que aparezca definido ese identificador, global, sin la clausula extern, y es además el único sitio donde se pueden inicializar las variables).

  Ejemplo:

Archivo 1	Archivo 2

int x,y=0;
char c;
void main ()
{
...
}
void func1()
{
x=123;
}
	extern int x,y;
	extern char c;
	void func22()
	{
	x=y/10;
	}
	void func23()
	{
	y=10;
	}

1- En el archivo 2 las variables x,y, c son globales, pero como van precedidas de extern se indica al compilador que ya han sido declarados en alguna otra parte (en este caso en archivo 1), sin que el compilador cree nuevos almacenamientos para ellas (se crean esos almacenamientos durante la compilación de archivo 1); es decir, el programa consiste en una sola copia de las variables x,y,c.

2- Despues, cuando se enlacen los dos módulos, se resolverán todas las referencias a las variables externas, es decir, las referencias a x,y,c que se hagan en archivo 2 se asocian a la (única) copia que existe de esas variables.

- **register**: aplicable solo a variables int y char, solicita al compilador que almacene las variables register en registros de la CPU en vez de en la memoria RAM, por lo que el acceso a estas variables es más rápido (un uso típico es como contadores de bucles).
  Debido a que C solo puede mantener dos valores en registro, se suelen declarar como máximo dos variables register por función. Algunos compiladores utilizan técnicas de optimización de código que ignoran este especificador.

Ejemplo: ¿Qué salida muestra el siguiente programa?

```
#include <stdio.h>

void a(void);
void b(void);
void c(void);

int x=10;

int main()
{
  int x = 4;

  printf ("local x en el entorno main es %d\n",x);
  {
    int x = 6;
    printf ("local x en un entorno interior de main es
              %d \n",x);
  }
  printf ("local x en el entorno main es %d\n",x);

  printf ("primera llamada a procedimientos\n");
  a();   b();   c();
  printf ("segunda llamada a procedimientos\n");
  a();   b();   c();
  printf ("local x en main es %d\n",x);
}
```

```
void a (void)
{
 int x = 20;

 printf ("local x en a es %d\n",x);
 x++;
 printf ("local x en a es %d antes de salir de a \n",x);
}

/*-------------------------------------------------*/


void b (void)
{
 static int x = 40;

 printf ("local x en b es %d\n",x);
 x++;
 printf ("local x en b es %d antes de salir de b \n",x);
}

/*-------------------------------------------------*/


void c (void)
{
 printf ("global x en c es %d\n",x);
 x++;
 printf ("global x en c es %d antes de salir de c \n",x);
}
```

Salida del programa:

Local x en entorno main es 4
Local x en el entorno interior main es 6
Local x en el entorno main es 4

primera llamada a procedimientos

---

local x en a es 20
local x en a es 21 antes de salir de a

local x en b es 40
local x en b es 41 antes de salir de b

global x en c es 10
global x en c es 11 antes de salir de c

segunda llamada a procedimientos

---

local x en a es 20
local x en a es 21 antes de salir de a

local x en b es 41
local x en b es 42 antes de salir de b

global x en c es 11
global x en c es 12 antes de salir de c

Local x en main es 4

❑ Directiva #define: Macros

✓La directiva **#define** efectúa una sustitución de un identificador por una cadena (constante o macro). Al identificador se le denomina **nombre de la macro** y a la cadena de reemplazamiento **sustitución de macro**. La forma general de la directiva es:

#define *nombre_de_macro cadena*

No lleva punto y coma ni "=", debido a que no es una sentencia. La cadena puede llevar cualquier numero de blancos y terminará con un enter.

Por ejemplo, macros de cts. literales enteras y cadena:

```
#define CIERTO 1
#define FALSO 0
#define ERROR "esto es un mensaje de error"
                                . . .
printf("%d %d %d", FALSO, CIERTO, CIERTO+1);
printf("%s",ERROR);
```

✓Una vez que se ha definido una macro se puede usar como parte de la definición de otros nombres de macro. Por ejemplo:

```
#define UNO 1
#define DOS UNO+UNO
```

✓ Si la cadena no coge en una línea, debemos utilizar la barra invertida para que la cadena no se corte.

```
#define CADENA_LARGA "esta es una \
        cadena muy larga que se usa como ejemplo"
```

✓Por convenio, los nombres de las macros se ponen en mayúsculas. El uso más común de las macros es para definir la longitud de los vectores.

```
#define TAM 100
float vector[TAM];
```

✓ Las macros pueden ser "pequeños procedimientos" incluso con parámetros.

Ejemplo:

```
#include <stdio.h>

#define MIN(a,b) (a<b) ? a : b

int main (void)
{
    int x, y;
    x=10;
    y=20;
    printf("el mínimo es %d", MIN(x,y));
}
```

- ✓ Hay que tener cuidado con la forma de definir las macros, ya que no son llamadas a subprograma, si no un mecanismo para sustituir un texto (el nombre de la macro) por otro (sustitución de macro). Confundir esto con autenticas llamadas a subrrutina puede dar lugar a errores sutiles:

```
#include <stdio.h>

#define PAR(a) a%2==0 ? 1 : 0

int main (void)
{
    if (PAR(9+1)) printf("es par");
    else printf("es impar");
}
```

El resultado es **impar**, ya que la sustitución se realiza de la siguiente forma:

```
9+1%2==0 ? 1 : 0
```

Lo primero que se hace es el módulo y después la suma. Para que funcione correctamente debería aparecer así:

```
#define PAR(a) (a)%2==0 ? 1 : 0
```

Ahora la sustitución se haría así:

```
(9+1)%2==0 ? 1 : 0
```

❑ Punteros a funciones

Toda función en C tiene una dirección, que puede verse como el punto de entrada a la misma o como la dirección en memoria de la primera instrucción máquina de la función.

Pueden declararse variables puntero a función:

- **Declaración**:
  Se declara una variable de tipo puntero a función

tipo (* nombrepf)(lista parámetros formales);

Si bien se puede declarar un puntero a función en cualquier sitio en el que se pueda declarar una variable, lo usual es que aparezca en la declaración de un parámetro de una función

tipo unaFunción(tipo p1, tipo p2, **tipo(*pf)(listapar)**)

- **Asignación**:
  Se asigna la dirección de una función que devuelva el mismo tipo

nombrepf= nombref; //Donde 'nombref' es una función

- **Invocación**:
  Se invoca a la función apuntada por nombrepf

(*nombrepf)(lista parámetros actuales)

Ejemplo:

```
char acarreo(char n1,char n2, char (*suma)(char,char));

char acarreoSumaDecimal(char d1, char d2);

char acarreoSumaHexadecimal(char h1, char h2);

int main()
{
 ch char,base;
 char (*pfsuma)(char,char);

 printf("¿base?: (d)ecimal, (h)exadecimal,  (s)alir)");
 scanf("%c"&base);

 while (base !='s')
 {
  if (base=='d')    pfsuma=acarreoSumaDecimal;
  else              pfsuma=acarreoSumaHexadecimal;

  printf("Introduce los dos números a sumar");
  scanf("%c %c",&a,&b);

  printf("El acarreo es %c", acarreo(a,b,pfsuma));

  printf("¿base?: (d)ecimal, (h)exadecimal, (s)alir)");
  scanf("%c"&base);
 };

}
```

```
char acarreo(char n1,char n2, char (*suma)(char,char))
{
  char c;

  c=(*suma)(n1,n2);
  return(c);
}

char acarreoSumaDecimal(char d1, char d2)
{
 /*
   pasar d1 y d2 a números enteros,
   sumarlos,
   dividir por 10 y
   devolver el cociente de la división
 */
}

char acarreoSumaHexadecimal(char h1, char h2);
{
 /*
   pasar d1 y d2 a numeros enteros decimales
   sumarlos,
   dividir por 16,
   pasar el cociente de la división a hexadecimal y
   devolverlo
*/
}
```

❑ main

✓ Todo programa en C debe contener obligatoriamente una función 'especial', a la que se le pasa el control cuando comienza a ejecutarse la aplicación. Esa función debe llamarse **main**, y su declaración es:

int main(int argc, char *argsv[])

- El valor entero que devuelve **main** es el valor que la aplicación devuelve al sistema operativo (que guarda en una variable de entorno) cuando esta termine de ejecutarse.

  Por lo general se devuelve **cero** si la ejecución fue correcta y valores **distintos de cero** para indicar diversos errores que pudieron ocurrir.

  Si bien no es obligatorio terminar el programa con un return, es conveniente indicarle a quien lo haya invocado, sea el Sistema Operativo o algún otro programa, si la finalización ha sido exitosa, o no.

- **argc** indica el número de argumentos con los que se ha invocado a la aplicación desde la línea de comandos, contando el nombre de la aplicación.

- **argsv** es un array de cadenas (tantas como indique argc), cada una de las cuales es uno de los argumentos con los que se invocó a la aplicación, siendo la primera el nombre de la aplicación

Ejemplo:

Supongamos una aplicación en C denominada "calculadora", a la que se le pasan tres argumentos: un número, una operación (+,-,*,/), un número y muestra por pantalla el resultado de la operación indicada.

La invocación desde la línea de comandos de esta aplicación sería así:

```
C:\> calculadora  3.8  *  2.1
```

espacios en blanco

Los argumentos de main serán:

- **argc: 4**

- **argsv[]: {"calculadora", "3.8", "*", "2.1"}**

❑ Estilo de programación

- ✓ Evitar el uso de variables globales; si una función necesita un dato, definir un parámetro en la cabecera de la función y pasarle ese dato a través del parámetro; esto permite crear funciones mas legibles, menos propensas a error y más fácilmente reutilizables

  Si se decide definir una variable global, forzar a que su alcance sea solo de fichero con el modificador **static**

- ✓ Usar juiciosamente los recursos que ofrece C para definir el alcance, visibilidad y durabilidad; en particular al usar el modificar static con variables locales y en el uso en bloques anidados de variables cuyos nombres coincidan con nombres de variables de bloques externos

- ✓ Declarar los prototipos de las funciones al principio del fichero, realizando una pequeña especificación en forma de comentario (PRECONDICIÓN Y POSTCONDICIÓN) encima de cada prototipo. Despues definir las funciones, dejando 'main' para el final.

- ✓ El orden en el que deben aparecer escrito los elementos del programa en un fichero es: #includes, #defines, variables globales (globales, static y extern), prototipos comentados, definiciones y main