Capítulo

4

Funciones y punteros

En el cuarto capítulo se describe cómo realizar la programación modular, que en C se realiza a través de funciones. Las funciones tienen sus propias variables (locales) y su propio código, y podrán ser llamadas por otras funciones. Además, se las pueden pasar datos (argumentos) de dos maneras, por valor y por referencia, para esta última manera hace falta describir un nuevo tipo de datos que es el puntero.

Creación, utilización y tipos de funciones

Uno de los principios básicos de la programación modular y estructurada es la división del programa en módulos, lo que se llaman subprogramas o subrutinas (en C funciones). La división:

1. Hace a un programa más fácil de escribir, entender y corregir (programación estructurada). Es el clásico principio de divide y vencerás, pudiéndose testear los módulos independientemente a través de un programa *driver* (programa que exclusivamente pasa argumentos al módulo a testear para comprobar su correcto funcionamiento).

2. Acorta los programas, ya que cada vez que se quiera utilizar el código de la subrutina, no habrá que repetir ese código, simplemente habrá que llamarla (reusabilidad).

3. Permite crear librerías de subprogramas de utilidades (ya compilados) que pueden ser usadas por otros programadores en otros programas (reusabilidad y modularidad) y ampliar las funciones que tiene el C (como la librería matemática que hemos usado en el capítulo segundo y que veremos posteriormente o las de entrada/salida).

4. Permite dividir un programa grande (varios miles de líneas) en varias partes y para varios programadores (división del trabajo). Para ello las cabeceras de los subprogramas y lo que realizan deben estar claras desde un principio.

5. Permite concebir el programa como un conjunto de cajas negras, que se pueden probar separadas y que se comunican datos entre ellas (a través de sus argumentos y del tipo de dato que devuelven), permitiendo que detalles internos de las funciones queden ocultos (abstracción). Esto dará lugar a la programación orientada a objetos, cubierta por otros tipos de lenguaje C como C++.

En C ya hemos usado una función que está definida por defecto y que es la primera que se ejecuta, la función *main*[1]. Después están las funciones definidas por el usuario y las predefinidas o funciones de librería. Las primeras requerirán su definición, declaración de uso y llamada desde otra función, las segundas sólo esto último, ya que los dos primeros puntos están realizados al hacer el `#include` del correspondiente fichero de cabecera ".h".

En lo que se refiere a la definición de la función, en el primer capítulo veíamos que la estructura de una función, ya sea la principal u otra, constaba de dos partes: cabecera y cuerpo. En la cabecera se hacían declaraciones hacia el preprocesador y se definía el nombre de la función (aunque no tenga argumentos siempre habrá que poner paréntesis). En el cuerpo, encerrado entre llaves, se declaraban los datos propios (locales) de la función y sus sentencias.

La definición de una función depende de si ésta tiene argumentos o no y del tipo de función que es. Si no tiene argumentos y es de tipo entero bastará en C con poner su nombre y una pareja de paréntesis (es buena práctica de programación, aunque sea redundante, definir explícitamente el tipo de dato de la función). Si la función tiene argumentos, éstos hay que declararlos como otro tipo de dato más, pero fuera del cuerpo de la función (entre llaves) e inmediatamente después del nombre. Se puede hacer de dos maneras: fuera de los paréntesis (C K&R) (parte izquierda) o dentro de ellos de la función (ANSI C) (parte derecha) que es la recomendable:

```
funcion (arg)                int funcion(int arg)
   int arg;                  {...}
{...}
```

Si una función tiene varios argumentos habrá que separar estos entre comas. Así, una función que tenga dos argumentos uno de tipo entero y uno de tipo flotante se tendría que declarar:

```
otra_funcion(i,f)            otra_funcion(int i, float f)
   int i;                    {...}
   float f;
{...}
```

Una vez realizada la definición de la función, llega la segunda parte, que es la declaración dentro de la otra función que la vaya a usar (llamar). Una función se puede utilizar como una variable, por eso mismo las funciones tienen tipo. El tipo por defecto es el entero, ya se ha comentado que en estas funciones no es necesario (si conveniente) poner tipo; si es otro, habrá que declararlas antes de usarlas.

Esta declaración se llama el *prototipo* de la función y puede ser más o menos compleja (incluyendo los tipos e identificadores de argumentos), dependiendo de las comprobaciones que queramos que haga el compilador, lo normal es sólo poner los tipos de datos de los argumentos, con lo cual el compilador, a la hora de probar el código, revisará el número de argumentos y su tipo. Si hay discrepancia dará error (en C clásico no existían los prototipos y si los argumentos no coincidían se producía una "coerción de argumentos"):

Declaración **prototipo**	float funcion ();
Con tipos de argumentos	float función (int , float)
Hasta con nombres	float función (int i, float f)

A diferencia de otros lenguajes, en C no existen dos tipos de subrutinas (por ejemplo en Pascal los procedimientos [*procedures*] y las funciones), sólo hay funciones, pero podemos hacer

[1] La función *main*() se diferencia del resto, a parte del nombre, en que en ella comienza la ejecución del programa, y que sus argumentos ya están definidos de antemano. Es sumamente exótico, pero una función también podría llamar a *main*().

que esas funciones devuelvan algo o no, en este último caso se utiliza la palabra reservada "void", para señalar algo que no tiene tipo o no existe.

Esto se puede ver en el ejemplo de ANSI C del capítulo 1 con la función asteriscos:

```
#include <stdio.h>
int main ()
{
        void asteriscos(void);     /* declaracion o prototipo */

        asteriscos ();      /* llamada o uso*/
        printf("Aqui pongo un mensaje\n");
        asteriscos ();      /* llamada */
}

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

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

El flujo de control de ejecución es el habitual, cuando se llama a asteriscos(), se hace una copia de los argumentos en la pila (si existen), se crea espacio para las variables locales en la misma, se empieza a ejecutar la primera sentencia de la misma, y cuando termina, se devuelve la ejecución a la función que la llamó, *main*(), en la sentencia posterior a la llamada y se elimina la pila usada por la función.

Algo que llama fuertemente la atención es que, independientemente de los argumentos definidos en la función, en la llamada se puede utilizar otro número distinto de argumentos. Si este número es menor, parte de los argumentos quedarán indefinidos, si es mayor serán ignorados (existe la posibilidad de crear funciones con número variable de argumentos). Como veremos, he aquí una de las ventajas de declarar los prototipos de las funciones completamente, ya que el compilador nos lo advertirá.

A diferencia de otros lenguajes, en C las funciones se pueden poner en cualquier orden y no se pueden anidar, es decir, no se pueden declarar y crear funciones dentro de otras. Normalmente se sigue un estilo de programación convencional que consiste en poner primero la función principal y después el resto de funciones por orden de llamada. Si usamos prototipos, este orden no presentará ningún problema, ya que si no, deberíamos definir las funciones de acuerdo a su uso y dejar la *main()* para el final.

También hay que tener en cuenta que si tenemos un buen estilo de programación, la función *main()* debería ser (pequeña) una serie de llamadas a otras funciones

Veamos un ejemplo de utilización de funciones. Se trata de hacer una función que calcule el factorial de un número, el código en ANSI C puede ser:

```
#include <stdio.h>
int main()
{
        long factorial();        /* declaracion */
        int i;

        for (i = 1; i < 15; i++)
        {
                if (factorial(i) == 0)    // llamada
                    printf("Error en la funcion\n");
                else
                    printf("El factorial de %d es: %ld\n", i, factorial(i));
        }
}
```

```
long factorial (int argu)  // definición, para maquinas de 32 bits
{
        long resultado;

        if (argu > 12)  /* excede la capacidad de cuatro bytes*/
        {
                fprintf(stderr, "Error en argumento");
                return (0L);      /* asignacion de valor a funcion */
        }
        else            /* caso correcto */
        {
                resultado = 1;
                while (argu != 1)       /* calculo factorial */
                {
                        resultado = resultado * argu;
                        argu -= 1;
                }
                return (resultado);    /* asignación de valor a funcion */
        }
}
```

En el programa hemos visto: como hemos creado una función de tipo *long* para utilizar números más grandes (hemos supuesto 4 bytes, aunque ahora son 8); como para cualquiera de las dos salidas de la función, (al utilizar el *if else*) hemos tenido que asignar valores a la misma para que no quede indeterminada, esto se hace con la sentencia *return;* y como hemos podido usarla en el *printf* como si fuera una variable. La sentencia *return* provocará la terminación de la función, si se usa un argumento en la misma, éste será el valor que devuelve la función.

De hecho, en la función *main()* también deberíamos usar la sentencia *return*. Ya que la función main() es por defecto de tipo entero pero no devuelve nada, se suele usar *return(0)* para terminarla o en muchas ocasiones exit(numero), donde numero suele ser un código de error (cero si no lo hay).

Una peculiaridad de los lenguajes modernos es la recursividad (incluyendo el C), es decir, que una subrutina se pueda llamarse a sí misma. La función que calcula el factorial podría modificarse utilizando la definición recursiva de factorial:

$$n! = n \times (n-1)!$$

sin comprobar el rango de los datos del argumento, que deben ser menores que 13, la función podría quedar:

```
long factorial(int argu)
{
  if (argu == 1)
        return(1);  /* esta es la salida */
  else
        /* n! = n x (n-1)! */
        return (factorial(argu-1) * argu);
}
```

Vemos como la función se llama a sí misma hasta que el factorial valga 1, si no pusiéramos esa condición, la función se llamaría continuamente hasta que el argumento fuera negativo y muy pequeño para la máquina o hasta que ocurra un error por falta de memoria (cada llamada a una función llena la pila (el *stack)* del programa con sus argumentos y dato devuelto).

Ya hemos visto como se define una función, su tipo, su lugar (antes o después de ser usada con prototipo), y como se llama. Pero a una función, incluida la *main()*, se la pueden pasar argumentos, y podemos hacer que estos sean modificados o no en la función. Para lo primero usaremos los datos de tipo puntero.

Paso de parámetros a una función. Punteros

Hemos dicho que en una función se pueden declarar variables locales, ¿Qué ocurre con los identificadores que se utilizan?, ¿Se conoce una variable declarada en una función en otra? La respuesta es que en C una variable declarada en una función (declaración local) no es conocida por las demás funciones (se almacena en la pila –o *stack*, una zona especial de la memoria– al llamarse a la función y se destruye al terminar). Además, como todas las funciones tienen la misma "categoría", las variables declaradas en la función *main* tampoco son conocidas por las demás funciones.

Otro "clase" de variables son las globales, también llamadas en C externas, que son las declaradas fuera de las funciones y que serán conocidas por todas las funciones posteriores de ese programa (en ese mismo fichero si hay varios). Se puede tener acceso a cualquier variable externa anterior o posteriormente declarada, incluso fuera de ese fichero, utilizando el calificador *extern.* Por tanto las variables declaradas en el función *main()* no son variables globales.

El no poder anidar funciones y la independencia del lugar de declaración, simplifican mucho las reglas de visibilidad y ámbito de variables (donde son conocidas), de tal manera que podemos decir que sólo existen tres:

1. Las variables locales sólo son conocidas en las funciones que las declaran y crean. Si son variables dentro de un bloque, sólo son conocidas en ese bloque.

2. Las variables globales pueden ser conocidas por todas las funciones (siempre es conveniente utilizar *extern,* para declarar que esa variable es global, no local).

3. Las funciones podrán ser llamadas por todas las funciones, independientemente de donde se declaren (siempre usando el prototipo).

Algo que define un buen estilo de programación, independientemente del lenguaje, es usar un número mínimo de variables globales, ya que éstas pueden ser cambiadas en las funciones, y puede que por varios programadores, con posibles conflictos. Supongamos que tenemos que hacer un programa extenso entre varias personas, si dividimos el trabajo por funciones (método normal) y no tenemos ninguna variable global; cada uno, en sus funciones, es libre de poner los identificadores adecuados a sus datos, sin tener en cuenta los que han puesto los demás programadores (no hay interacción al ser declaraciones locales), con lo cual, se evitarán muchos fallos inesperados: "*yo pensaba que estaba cambiando mis variables... y resulta que era la global*..." Además, cuando se hace programación multihilo (multithreading) puede ser causa de carreras críticas.

Como hemos dicho, una variable global puede ser conocida, y por lo tanto se puede leer y cambiar, en todas las funciones; y además, podemos pasar a la función argumentos, por lo que podemos decir que existen tres maneras de transferir datos a una función:

1. Utilizando variables globales. No es conveniente salvo que se declaren *extern*.

2. Pasando argumentos por valor, es decir, copiando el valor que tiene en ese momento la variable que usamos de argumento (parámetro actual) al argumento de la función que estamos llamando (parámetro formal). De esta forma si cambiamos el valor del argumento en la función sólo cambiamos el valor de la copia, pero no el de la llamada (es la forma que hemos visto hasta ahora).

3. Pasando los argumentos por referencia. No se pasa una copia del valor de la variable sino su dirección en memoria. Esto quiere decir que los parámetros actuales y formales son los mismos (misma dirección) y si se cambia su valor en la función, se cambia la variable que ha sido colocada como argumento en la llamada.

En otros lenguajes, como el Fortran, el paso de argumentos es siempre por referencia, en otros como el Pascal hay que utilizar la palabra reservada *var* delante de un argumento para decir

que éste ha sido pasado por referencia. En C existe otro mecanismo para el paso por referencia que es la utilización de punteros. Podemos entender un tipo de dato puntero como una identificación simbólica, es decir una variable, que contiene una dirección. Antes teníamos variables que contenían un valor y que estaban almacenadas en una determinada posición, ahora tenemos un tipo de variable que puede contener en vez de datos, direcciones.

Para manejar direcciones, el C nos proporciona dos operadores, el de dirección "&" y el de indirección "*". Los dos son de tipo unario y de alta precedencia. El primero usado delante de una variable nos dará la dirección de la misma y el segundo delante de un puntero nos dará el valor que contiene la dirección almacenada en ese puntero.

Para declarar punteros tendremos que utilizar el asterisco, que designa al identificador como un puntero (ojo con la confusa nomenclatura, no es el operador indirección):

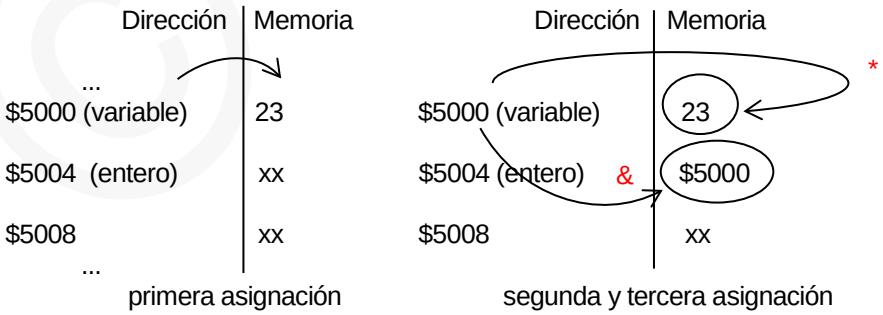
```
int *entero;       /* esto es un puntero entero */
float *real;       /* esto es un puntero flotante */
int variable;      /*variable normal de tipo entero */
```

Aunque todos los punteros sean una dirección, es decir, algo parecido a un entero *unsigned*, en la dirección que contienen puede haber distintos tipos de datos, por lo que en la declaración del puntero también se pone ese tipo de dato y se dice para abreviar que ese puntero es del tipo de dato apuntado. Así, se hablará de punteros a enteros, flotantes, etc., dependiendo de lo que contenga la dirección que está en el propio puntero.

La utilización de los operadores se ve reflejada en estos ejemplos:

```
variable = 23;  /* supongamos que está en la dirección 5000 */
entero = &variable;  /* almacenado en 5004 pero contiene 5000 */
variable = *entero;    /* equivalente a la primera */
```

En el primer caso hacemos una asignación normal de una variable entera (supongamos que esa variable está almacenada en la dirección $5000), en el segundo caso estamos asignando la dirección de la variable ($5000) a través del operador "&", al puntero entero (el puntero podría estar almacenado en la dirección $5004 –dirección consecutiva– y después de la asignación contener el valor $5000) y en el tercer caso, estamos asignando el valor de lo apuntado por un puntero, operador "*", a una variable normal; el puntero que está en $5004, contiene la dirección $5000, en esta dirección está el valor 23, y es este valor el que se asigna a "variable" de nuevo). En el siguiente capítulo veremos otras operaciones (operandos) que se pueden utilizar con punteros dada su equivalencia con los *arrays*.



Hay que recordar, y esto es importante, que para usar un puntero, antes hay que asignarle una dirección válida, no se puede trabajar con punteros no asignados, porque contendrán direcciones inesperadas, normalmente cero.

Teniendo una idea general de lo que es un puntero, podremos conseguir con él hacer una llamada por referencia en una función, simplemente haciendo que los argumentos de la misma sean de este tipo. Supongamos que queremos crear una función que nos intercambie dos enteros, como una función sólo devuelve un valor, tendremos que pasar uno de los argumentos (o los dos) por

referencia, para ello haremos los argumentos de tipo puntero y en la llamada utilizaremos también las direcciones de las variables (por ejemplo usando el operador dirección "&"):

```
void intercambia(int *u, int *v)    /* punteros a enteros */
{
   int temp; /* se necesita una variable local temporal de bufer */

   temp = *u;   /* lo apuntado por u se pone en temp */
   *u = *v;     /* lo apuntado por v se pone en lo apuntado por u */
   *v = temp;  /* el valor de temp se pone en la dirección v */
}
```

La llamada a esta función sería (falta el resto del código):

```
…
int entero1 = 5;
int entero2 = 4;

intercambia(&entero1, &entero2);
…
```

Si no hubiéramos puesto los argumentos de tipo puntero, el cambio sólo se habría realizado en el interior de la función (en la pila del programa, que una vez terminada la llamada se sobreescribe con otras cosas), después de ella los valores de entero1 y entero2 serían los mismos que antes de la llamada.

Ámbito de datos y modos de almacenamiento

Hemos visto en el apartado anterior que existen dos tipos de variables, las locales y las globales. Además en C podemos controlar el modo de almacenamiento de las mismas, dado que es un lenguaje de medio nivel relativamente cercano al computador.

Hay cuatro tipos de adjetivos que se pueden dar a las variables:

- AUTOMÁTICAS: es el modo por defecto, y pertenecen a este tipo, todas las variables locales que hemos visto hasta ahora. Se declararán con la palabra reservada `auto` delante, aunque todas son *auto* por defecto. Las variables automáticas se crearán cuando la función se llame y se destruirán cuando la función termine de ejecutarse. Su alcance será el del bloque donde estén encerradas. Aunque normalmente siempre se declaren las variables al comienzo de la función (después de la llave), no hay ningún problema en crear dentro de las funciones otros bloques con { } y declarar dentro de ellos otras variables. No están inicializadas por defecto, pero se pueden inicializar con cualquier cosa que tenga ya un valor, incluso una llamada a una función.

```
funcion (int arg1, char arg2)
{
       auto char local1;    /* no seria necesario el auto */
       int  local2;         /* variable local a funcion */
       local1 = '\n';
       local2 = 32;
       {            /* otro ambito por las llaves */
        int local2; /* locales al nuevo ambito oculta la anterior */
        int local3;
        local2 = 15;   /* nos referimos a la interior */
        local3 = 4;
       }
       /* ahora local2 valdra 32 y no 15, ya que es la de fuera */
       /* Esto no se puede hacer, local3 no existe y daria errores*/
       local3 = 232;
}
```

- EXTERNAS: es la forma de acceder explícitamente a las variables globales ya creadas, utilizándose para ello la palabra reservada `extern` (hemos dicho que cuando accedemos a una variable global en una función es muy conveniente declararla con *extern* para saber con seguridad que es global y que se puede cambiar en la función). Con esta declaración no se crean variables globales, sino que sólo se accede a ellas, por lo tanto, no se pueden inicializar en la declaración. Además, la podemos inicializar con una constante (se hace antes de ejecutar el programa). Veamos algunos casos:

```
int fuera;                          fun ()
int main ()                                 {
{                                           extern int fuera;
        extern int fuera;                   ...
         ...                        }
}
                    ____________________

int fuera;
main ()                             fun ()
{                                   {
        extern int fuera;                       ...
   ...                              }
}
                    ____________________

int fuera;
main ()                             fun ()
{                                   {
        int fuera;                              int fuera;
          ...                                    ...
}                                   }
```

En el primer caso se crea una variable global, que por lo tanto tendrá validez durante toda la ejecución del programa y se declara explícitamente en las dos funciones, sólo hay una variable. En el segundo caso no existe la declaración en la segunda función, pero como está declarada antes que la función, es conocida implícitamente y puede ser utilizada por su nombre, también sólo hay una variable. En el tercer caso, se declaran dos variables locales en las funciones con el mismo identificador, por lo que tenemos tres variables en total y además no podremos acceder a la variable externa (puede ser origen de errores difíciles de detectar si creemos que cambiamos la global en vez de la local o viceversa por no haber usado un *extern*) ya que hemos utilizado su identificador localmente (accederemos a las variables locales no a la global).

Cuando se habla del ámbito de validez de una variable global, hay que tener en cuenta tres aspectos: primero en que lugar del programa está declarada, ya que sólo las funciones declaradas después pueden acceder a ella de forma implícita, segundo el fichero en que está declarada (un programa puede estar confeccionado en varios ficheros), ya que sólo se la conocerá implícitamente en ese fichero y tercero si se la utiliza explícitamente con el adjetivo *extern*, siempre tendremos acceso a ella, esté como esté declarada. Esta declaración se puede hacer dentro de una función, para tener acceso a la variable en esa función, o fuera de ella, para tener acceso a la variable en todas las funciones posteriores (no recomendado).

- ESTÁTICAS: Este adjetivo o atributo se refiere a la permanencia de las variables en el programa, por lo que puede haber variables estáticas locales o globales. Para ello se utilizará la palabra reservada `static`. Las estáticas locales permanecerán después de la terminación de las funciones que las declaran, por lo cual, conservarán su valor para posteriores llamadas. Las estáticas globales ya son permanentes de por sí, por lo que en este aspecto no cambian. La inicialización es igual que las externas, pero además si se inicializan, sólo lo hacen una vez, ya que esto dentro de una función podría dar problemas.

- REGISTROS: Las variables declaradas de esta forma son iguales que las automáticas, sólo se le pide al compilador, que si puede, las almacene en los registros internos del

procesador. Si no puede hacerlo, simplemente será almacenada de la forma usual. La efectividad de esta declaración depende en gran manera de la máquina en cuestión, por lo cual puede haber grandes restricciones en su uso. La palabra reservada para ello es `register`. Se suele hacer si consideramos que esa variable es muy importante y muy usada, con lo cual ganaremos rendimiento en la ejecución del programa.

Librería de C estándar

Ahora que ya sabemos crear nuestras propias funciones, debemos recordar que en C existen un gran número de funciones agrupadas en librerías, cuyos prototipos (variables, macros y también constantes) se pueden incluir en un programa a través de la instrucción del preprocesador *#include* con su correspondiente fichero de cabecera (".h").

Hay una librería especial que es la librería estándar de C (ANSI C Standard Library, ISO C library o `libc`), con rutinas estandarizadas por un comité de la Organización Internacional para la Estandarización (ISO). Esta librería implementa operaciones comunes, tales como las de entrada y salida o el manejo de cadenas y siempre se incluye por defecto. En otros casos será necesario enlazar (linkar) una librería ya compilada como la matemática.

En el estándar de C hay 15 ficheros de cabecera[1]:

<assert.h>	Contiene la macro assert (aserción), utilizada para detectar errores lógicos y otros tipos de fallos en la depuración de un programa sobre todo cuando se hacen llamadas al sistema.
<ctype.h>	Contiene funciones para clasificar caracteres según sus tipos o para convertir entre mayúsculas y minúsculas independientemente del conjunto de caracteres (típicamente ASCII o alguna de sus extensiones).
<errno.h>	Para analizar los códigos de error devueltos por las funciones de biblioteca. errno. Nos da los códigos y mensajes de error estándar.
<float.h>	Contiene la definición de constantes que especifican ciertas propiedades de la biblioteca de coma flotante, como valores máximos y mínimos de precisión o rango. También una serie de macros que nos dan los límites que podemos alcanzar con los flotantes y los dobles.
<limits.h>	Contiene la definición de constantes que especifican ciertas propiedades de los tipos enteros (en general desde char a long) como por ejemplo el rango.
<locale.h>	Para la función setlocale() y las constantes relacionadas. Se utiliza para seleccionar el entorno local apropiado (configuración regional) de la máquina, como la moneda.
<math.h>	Contiene las funciones matemáticas comunes.
<setjmp.h>	Declara las macros setjmp y longjmp para proporcionar saltos de flujo de control de programa no locales.
<signal.h>	Para controlar algunas situaciones excepcionales como la división por cero.
<stdarg.h>	Posibilita el acceso a una cantidad variable de argumentos pasados a una función.

[1] http://www.csse.uwa.edu.au/programming/ansic-library.html

<stddef.h>	Para definir varios tipos de macros de utilidad.
<stdio.h>	Proporciona el núcleo de las capacidades de entrada/salida del lenguaje C. Tanto sobre consola como sobre ficheros.
<stdlib.h>	Para realizar ciertas operaciones como conversión de tipos, generación de números pseudo-aleatorios, gestión de memoria dinámica, etc.
<string.h>	Para manipulación de cadenas de caracteres. Próximo tema.
<time.h>	Para tratamiento y conversión entre formatos de fecha y hora.

De las cuales, las más importantes (en rojo) son la conocida *stdio.h*, la de caracteres ctype.h, la *math.h* (para funciones matemáticas que después veremos), la librería estándar *stdlib.h*, y la de manipulación de cadenas *string.h*[1]:

1. Funciones de caracteres: En el estándar ANSI están en la librería *ctype.h*. Las funciones/macros más importantes son: *isalpha*(ch), para saber si un carácter es alfanumérico, isdigit(ch) si es dígito, *islower*(ch), si es letra minúscula, *isspace*(ch), si espacio y si es mayúscula *isupper*(ch).

2. Funciones matemáticas. Están definidas en la librería *math.h* y las principales funciones son: valor absoluto *fabs*(x), raíz cuadrada *sqrt*(x), potencia base exponente *pow*(x,y), techo *ceil*(x), suelo *floor*(x), exponente e *exp*(x), logaritmo neperiano *log*(x) y decimal *log10*(x). Las trigonométricas son: coseno *cos*(x), seno *sin*(x), tangente *tan*(x), arcocoseno *acos*(x), arcoseno *asin*(x), arcotangente *atan*(x), arcotangente de y/x *atan2*(y,x), coseno hiperbólico *cosh*(), seno hiperbólico *sinh*() y tangente hiperbólica *tanh*(). Todas ellas son de tipo *double*. Además habrá que *linkar* con la librería matemática (-lm) ya que la inclusión de *math.h* sólo introduce las definiciones de constantes, tipos de datos y prototipos, pero no los cuerpos de las funciones.

3. Funciones de strings, *string.h*. Las más importantes se verán en el siguiente tema.

4. Conversión de caracteres y otras funciones: Se incluyen en la librería *stdlib.h* y son de uso cotidiano.

   - Las de conversión como de tira a entero *atoi*(tira), de tira a flotante *atof*(tira), etc.
   - Matemáticas como valor absoluto *abs*(), o la de generación aleatoria *rand*().
   - Entorno como la de salida *exit*(), llamadas al sistema system(), entorno de ejecución como getenv().
   - Petición de memoria dinámica. Como hemos repetido, un puntero necesita inicializarse antes de ser usado, esto se puede hacer dándole la dirección de alguna variable (con "&") o *array* (próximo capítulo) o también pidiendo memoria dinámica al sistema[2]. Para hacerlo, existen dos funciones definidas en la librería estándar *stdlib.h:*

     ```
     puntero = malloc(longitud);
     puntero = calloc(unidades, bytes por unidad);
     puntero = realloc(puntero, nueva_longitud);
     ```

     Y después del uso liberarla:

[1] http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

[2] Cuando realizamos un programa, lo compilamos y lo cargamos en el computador para ejecutarlo, estamos reservando suficiente memoria para almacenar en ella todas las variables del programa (variables externas o globales) y un espacio de pila (*stack)* para ir creando y destruyendo las variables locales. Existe un tercer "tipo" de memoria, la memoria dinámica o montón (*heap*) que se busca en tiempo de ejecución a través de llamadas del lenguaje que tienen correspondencia directa con llamadas del sistema operativo.

```
free(puntero);
```

Existen otros ficheros estándar posteriores (NA1 del añadido del 95, C99 del nuevo estándar del 99 y C11 del reciente C de 2011):

- <complex.h> Conjunto de funciones para manipular números complejos (nuevo en C99).
- <fenv.h> Para controlar entornos en coma flotante (nuevo en C99).
- <inttypes.h> Para operaciones de conversión con precisión entre tipos enteros (nuevo en C99).
- <iso646.h> Para utilizar los conjuntos de caracteres ISO 646 (nuevo en NA1).
- <stdalign.h> Para alineamiento de datos (del C11).
- <stdatomic.h> Para operaciones atómicas entre threads (del C11).
- <stdbool.h> Para el tipo booleano (nuevo en C99).
- <stdint.h> Para definir varios tipos enteros (nuevo en C99).
- <stdnoreturn.h> Específico para funciones que no devuelven nada (del C11).
- <tgmath.h> Contiene funcionalidades matemáticas de tipo genérico (type-generic) (nuevo en C99).
- <threads.h> Para manejar threads y sus mecanismos de sincronización (del C11).
- <wchar.h> Para manipular flujos de datos anchos y varias clases de cadenas de caracteres anchos (2 o más bytes por carácter), necesario para soportar caracteres de diferentes idiomas (nuevo en NA1).
- <uchar.h> Para manejar caracteres del Unicode (del C11).
- <wchar.h> Define funciones para manejar cadenas de caracteres largas (del NA1).
- <wctype.h> Para clasificar caracteres anchos (nuevo en NA1).

Aunque el nombre y las funciones incluidas en cada librería pueden cambiar, uno de los intentos del estándar ANSI C es fijar las librerías definidas en todos los sistemas.

Argumentos de main

La función principal *main()* es diferente a las otras ya que normalmente no es llamada por nadie y se ejecuta automáticamente al inicio del programa. Entonces ¿para qué usar argumentos en ella? La respuesta es sencilla, los argumentos de main son los argumentos del propio programa (pensad en línea de comandos sobre todo en sistemas Linux). Así, cuando ejecutamos un programa desde el sistema y le pasamos unos argumentos debe existir una manera de obtenerlos dentro del programa.

```
bash$ programa argumento1 argumento2
```

Para ello *main* toma dos argumentos clásicos (incluso un tercero que no mencionamos): una variable de tipo entero que nos indica el número de argumentos que nos han pasado desde la Shell y una variable que es un array de cadenas (*strings*) de caracteres con los valores de los argumentos. En el ejemplo, la primera variable valdría 2 y el array de cadenas tendría tres elementos: “programa”, “argumento1” y “argumento2”.

De arrays y cadenas trata el siguiente tema.

Ejemplos

```c
/* Ejemplo 17.c
comprende las funciones y su 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');
}
```

```c
/* Ejemplo 18.c
comprende las funciones y su uso
Observa los diferentes tipos de datos que devuielven las funciones
Este es un programa iterativo              */


#include <stdio.h>
int main()
{
        long factorial();        /* declaracion */
        int i;
        for (i = 1; i < 15; i++)
        {
                if (factorial(i) == 0)
                    printf("Error en la funcion\n");
                else
                    printf("El factorial de %d es: %ld\n", i, factorial(i));
        }
    return(0);
}

long factorial (int argu)
{
        long resultado;

        if (argu > 12)  /* excede la capacidad de cuatro bytes*/
        {
                printf("Error en argumento");
                return (0L);      /* asignacion de valor a funcion */
        }
        else            /* caso correcto */
        {
                resultado = 1;
                while (argu != 1)       /* calculo factorial */
                {
                        resultado = resultado * argu;
                        argu -= 1;
                }
                return (resultado);    /* asignaciÃ³n de valor a funcion */
        }
}
```

```c
/* Ejemplo 19.c
comprende las funciones y su uso
Observa los diferentes tipos de datos que devuelven las funciones
Este es un programa recursivo          */


#include <stdio.h>
int main()
{
        long factorial();       /* declaracion */
        int i;
        for (i = 1; i < 15; i++)
        {
                if (factorial(i) == 0)
                    printf("Error en la funcion\n");
                else
                    printf("El factorial de %d es: %ld\n", i, factorial(i));
        }
    return(0);
}

long factorial(int argu)
{
    if (argu == 1)
        return(1);  /* esta es la salida */
    else
    /* n! = n x (n-1)! */
        return (factorial(argu-1) * argu);
}
```

```c
/* Ejemplo 20.c
comprende las funciones y su uso
Observa los diferentes tipos de datos que devuelven las funciones
Este es un programa recursivo          */

#include <stdio.h>
int main()
{
        long factorial();       /* declaracion */
        int i;
        for (i = 1; i < 15; i++)
        {
                if (factorial(i) == 0)
                    printf("Error en la funcion\n");
                else
                    printf("El factorial de %d es: %ld\n", i, factorial(i));
        }
    i=main();       /* curiosidad : llamar al propio programa main. nunca terminará */
    return(0);
}

long factorial(int argu)
{
    if (argu == 1)
        return(1);  /* esta es la salida */
    else
    /* n! = n x (n-1)! */
        return (factorial(argu-1) * argu);
}
```

```c
/* Ejemplo 21.c
comprende las funciones y su uso, en este caso el paso por referencia
Observa los diferentes tipos de datos que devuelven las funciones
Entiende el uso de punteros para el paso por referencia          */


#include <stdio.h>
int main()
{
        int uno=1, otro=2;
        void intercambia(int *, int *);

    printf("Valores de uno y otro %d - %d\n", uno, otro);

    intercambia(&uno, &otro);

    printf("Valores de uno y otro %d - %d\n", uno, otro);


    return(0);
}

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 direcciÃ³n v */
    *v = temp;
}

```

```c
/* Ejemplo 22.c
comprende las variables, su ámbito y su almacenamiento.
Haz una predicción de los valores que aparecerán en pantalla.     */


#include <stdio.h>

int global;

int main()
{
    int uno=1, otro=2;
    // extern int global =3;  // no se pude inicializar, ya existe
    extern int global;
    void funcion(int, int);

    global = 1234;
    printf("Valores de uno  otro  global:  %d  %d   %d\n", uno, otro, global);

    funcion(uno, otro);

    printf("Valores de uno  otro  global:  %d  %d   %d\n", uno, otro, global);

    return(0);
}

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);
}
```

```c
/* Ejemplo 23.c

Programa para recuperar los datos de la linea de ejecucion
A un programa se le pueden para argumentos a la hora de ejecutarlo y son recuperables de esta manera   */


#include <stdio.h>

int main(int cuenta, char *argumentos[], char **entorno)
{
    int i;

    printf("Numero de arguemntos %d\n", cuenta);
    for(i=0;i<cuenta;i++)
        printf("Nombres de argumentos %d %s\n", i, argumentos[i]);

    while (*entorno != (char *)0)
    /* si sumo el puntero paso al siguiente elemento */
        printf("%s\n", *entorno++);

    return(0);
}
```