Tema 5:

Funciones y Módulos

***Objetivos del tema**: Uno de los pilares de la programación estructurada es el realizar un diseño modular y descendente (top-down) del problema a resolver. Hasta ahora todos los problemas que hemos hecho constan de una única unidad: la función main o principal. De este modo no es posible realizar un diseño descendente del problema ya que todo el cómputo debe realizarse en una única unidad.*

En este capítulo mostraremos cómo el lenguaje de programación C da soporte al diseño descendente mediante los programas multi archivo y el uso de funciones, fragmentos de código relativamente independientes que reciben una serie de parámetros y devuelven un dato como resultado del proceso que realiza.

Por otro lado, las funciones dan soporte a la representación computacional de la funcionalidad de los objetos del mundo real, de modo similar a como las variables permiten representar sus propiedades.

.

Índice

1 Funciones y diseño estructurado

Hasta ahora todos los ejemplos que hemos mostrado constan únicamente de un bloque de código, el que contenía la función main. En los primeros años de la programación todos los programas constaban únicamente de un sólo bloque. Este bloque podía llegar a ser tan grande como se quisiera. Esta aproximación pronto mostró notables desventajas y debilidades:

- La complejidad del programa resultante era muy alta ya que había demasiadas líneas que dependían las unas de las otras.
- Como consecuencia del enorme número de dependencias el número de errores era muy alto y éstos eran difíciles de encontrar.
- Al estar formado todo el programa por una única unidad era muy difícil dividirlo entre un grupo de programadores.

Esto llevó a los diseñadores a buscar formas de dividir los programas en partes lo más independientes posibles. Como resultado de este esfuerzo surgió la programación estructurada; en este paradigma la unidad mínima de programación con sentido, nombre propio y relativa independencia del resto es la función o el procedimiento. Una **función** o un **procedimiento** es un pequeño programa que a partir de unos datos de entrada obtiene unos resultados. La diferencia entre una función y un procedimiento es que la función devuelve siempre un valor a la línea de código que la invoca, al igual que sucede con las funciones matemáticas. Los procedimientos están formados por un fragmento de programa que realiza una determinada tarea sin devolver un valor de retorno.

Las funciones o procedimientos de un programa se pueden agrupar en módulos. Un módulo es un archivo de código que incluye una serie de funciones o procedimientos que realizan tareas similares o relacionadas entre sí. Por tanto, en el caso general, un programa estará compuesto por un conjunto de módulos, cada uno de los cuales a su vez estará compuesto por un conjunto de funciones o procedimientos.

En todo programa existirá una función o un procedimiento principal, que es el primero en ser ejecutado; en C es la función main. Esta delega parte de sus tareas ejecutando diversas funciones, que pueden pertenecer a módulos diferentes. Estas funciones, a su vez, se pueden apoyar en otras funciones. De esta forma se va desglosando las tareas complejas en tareas más simples hasta que dichas tareas pueden ser realizadas por un único bloque de código más o menos pequeño.

2 Funciones en el lenguaje C

En cualquier lenguaje de programación las funciones y/o procedimientos son fragmentos de código independientes con nombre y entidad propia, y se agrupan en módulos de programación, que no son más que archivos de código independientes que se compilan por separado y luego se enlazan entre sí para formar el programa completo. La forma de definir dichos módulos con sus respectivas funciones difiere en cada lenguaje de programación. En el caso de C debemos de tener en cuenta los siguientes puntos:

- Sólo existen funciones. No se puede escribir ninguna línea de código ejecutable (excluyendo declaraciones y definiciones) fuera del cuerpo de una función. No se dispone de procedimientos; para emular la funcionalidad de procedimientos se utilizan funciones que devuelven como valor el tipo de dato *void*, esto es, no devuelven nada.
- No se puede definir una función dentro de otra función; todas las funciones de un módulo deben de estar en el mismo nivel.
- Siempre debe de existir la función principal main dentro del código del programa; esta función es la que permite arrancar la ejecución del programa.

2.1 Definición de una función

La sintaxis de definición de una función es la siguiente:

```
tipo_retorno nombre ( lista de parametros )
    {
    declaración de variables locales
    sentencias
    }
```

El tipo_retorno es el tipo de dato correspondiente con el valor devuelto por la función. Puede ser cualquiera de los del C: char , int, float, etcétera. Por ejemplo, para la función pow(), una función incluida en las librerías del compilador que permite calcular potencias, el valor devuelto es el primer argumento elevado al segundo (argumento es lo mismo que parámetro) y es de tipo double. Si una función no devuelve ningún valor, se debe poner como tipo de retorno void, que significa “nulo” o “vacio”. Si no se indica ningún tipo de retorno el compilador supone que es int; no es una buena práctica de programación confiar en el tipo de retorno por defecto y no indicarlo explícitamente ya que es propenso a fallos y complica la legibilidad del programa.

El nombre de la función puede ser cualquiera que elija el programador, siguiendo las mismas reglas que los nombres de las variables. Este nombre es el que se utilizará cuando se la llame desde la función principal (main) o desde otra función.

Los parámetros o argumentos son los valores sobre los que actúa la función: por ejemplo, el primer argumento de pow() es la base y el segundo el exponente. La lista de parámetros es de la forma

```
tipo1 parametro1, tipo2 parametro2, ... , tipoN parametroN
```

Hay que especificar de forma separada el tipo de cada parámetro, aunque haya varios del mismo tipo. Si no se pone el tipo de un parámetro, el compilador supone que es int. Si una función no utiliza parámetros, se pone dentro de los paréntesis la palabra void. Si no se pone nada entre los paréntesis, la función no tiene parámetros, pero sin embargo el compilador no avisa con un error si se llama a la función pasándole algún parámetro.

Las variables definidas dentro del cuerpo de una función se denominan **variables locales. S**on variables propias de la función y no existen para el resto del programa. Se declaran, inicializan y emplean igual que cualquier otra variable. Las variables definidas fuera del cuerpo de una función podrán ser accedidas por cualquier función del programa.

Hay una sentencia que está íntimamente relacionada con el uso de funciones: **return.** Esta sentencia hace que la función termine y vuelva a la sentencia siguiente a la de llamada. Si se quiere devolver un valor, se pone al lado de return, de la forma:

```
return expresión;
```

Donde el resultado a devolver, "expresión", debe ser un dato del mismo tipo que aquél especificado como tipo de dato para el valor devuelto en la definición de la función. Una función puede tener varias sentencias return, pero en un momento dado sólo se ejecuta una de ellas, ya que en cuanto se ejecuta una, la función devuelve el control a aquella función que la llamo.

Todos los componentes que forman parte de la definición de una función, salvo el nombre, los paréntesis y las llaves, son opcionales; de forma que la siguiente función es válida (aunque no hace nada):

```
funcion_simple(){
}
```

A continuación mostraremos un ejemplo de programa en el cual se emplea una función para multiplicar dos números reales:

```
/*
*ejemplo6_1.c
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

float  multiplicar(float num1,float num2) {
  return num1*num2;
}

main()
{
  float A,B, resultado;
  srand(time(NULL));
  A = rand();
  B = rand();
  resultado=multiplicar(A,B);
  printf("La multiplicación es %f \n",resultado);
}
```

La función rand() genera números enteros aleatorios entre 0 y la constante RAND_MAX. La función srand () se utiliza para inicializar la rutina de generación de números aleatorios con una semilla. Esta semilla debe ser un número entero sin signo. Para generarlo empleamos la función time(NULL), que devuelve el instante actual de tiempo en un formato que es opaco para el programador. Para este problema no nos importa que el

instante de tiempo sea opaco ya que lo único que deseamos es un entero positivo que varíe de una ejecución a otra del programa. Si no se proporcionase una semilla diferente en cada ejecución del programa siempre obtendríamos los mismos números "aleatorios". Estas funciones están definidas en stdlib.h. y en time.h.

Cuando se ejecuta la línea resultado= multiplicar(A,B) se ejecutan los cálculos indicados en la función y al final, mediante la sentencia return, se devuelve un valor, que es asignado a la variable resultado del programa principal.

Como se puede ver, el nombre de los parámetros o argumentos de la función no tiene ninguna relación con los de las variables que recibe (A y B); simplemente el valor de la primera variable (A) se introduce en el primer parámetro (num1) y el de la segunda en el segundo (num2).

2.2 Variables locales

Tanto los parámetros de una función como cualquier variable que definamos dentro de ella son variables locales de la función. Esto es, no existen fuera de la función donde se declaran (por ejemplo, en el programa ejemplo6_1.c num1 y num2 sólo existen dentro de la función multiplicar). Si fuera de la función se declaran otras variables con el mismo nombre, las variables serán distintas. Tomemos como ejemplo el siguiente programa:

```
/*
*ejemplo6_2.c
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int evaluar(int x) {
  x = 4* pow(x,2)+ 3*x +5;
  return x;
}
main()
{
  int x;
  srand(time(NULL));
  x = rand();
  printf("La función 4x^2 + 3x + 5 en el punto %i vale %l
\n",x,evaluar(x));
}
```

Este programa genera un valor aleatorio x y calcula el valor de la función 4x^2 + 3x + 5 en el punto dado por x. Para ello se apoya en la función evaluar, en la cual se llama x tanto al parámetro que se le pasa como al resultado del cálculo del valor de la función en dicho punto. A pesar de que la función machaca el valor original de la variable x que se le pasa como argumento, en la función principal se imprime el valor de x correspondiente con el punto del eje de ordenadas donde se evaluó la función. Esto es así porque la variable x de la función principal y la variable x de la función "evaluar" son distintas y se corresponden con direcciones de memoria diferentes; el guardar un valor en una de esas direcciones de memoria no afecta en absoluto la otra.

Las variables globales son las declaradas fuera de cualquier función. Son utilizables por cualquier función, salvo que exista una variable local con el mismo nombre. Las diferencias más importantes de las variables locales frente a las globales son:

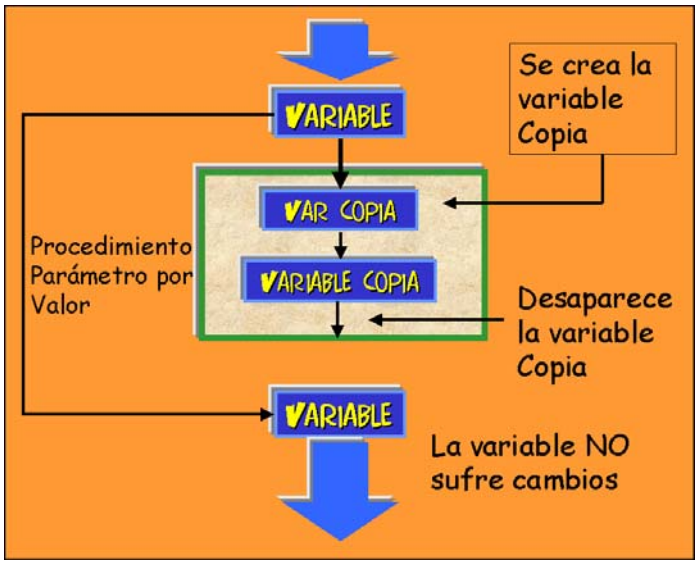
- Sólo existen dentro de la función en la que se declaran.
- Cuando termina la función donde están declaradas, su valor se pierde. Aunque se vuelva a llamar a la función, no se conserva el valor anterior.

Hasta C99 las variables que se declaraban dentro de la función tenían que ir necesariamente al principio de toda la función; esto es, toda función debería de constar primero de una secuencia de declaración de variables donde había que declarar todas y cada una de las variables usadas por la función, y a continuación iba la secuencia de instrucciones de la función. C99 permite definir variables dentro de la secuencia de instrucciones de cada función, lo cual es de gran utilidad y permite escribir un código más legible. Sin embargo, dado lo reciente de esta especificación, es posible que muchos compiladores todavía no soporten esta característica.

2.3 Paso de parámetros por valor y por referencia

En los ejemplos anteriores, se ha utilizado el paso de parámetros **por valor**: cuando se llama a la función se introducen los valores de las variables de llamada en los parámetros de la función. Los valores que utiliza la función son una copia de las variables originales. La función opera con dicha copia, produciendo los mismos resultados que si utilizase las variables originales. La única diferencia es que no se puede modificar las variables

originales: aunque cambie el valor de los parámetros, este valor no se copia de vuelta al terminar la función a la variable que fue pasada como argumento.



Por ello, la siguente función, que se supone sirve para intercambiar los valores de dos variables, en realidad no hace nada:

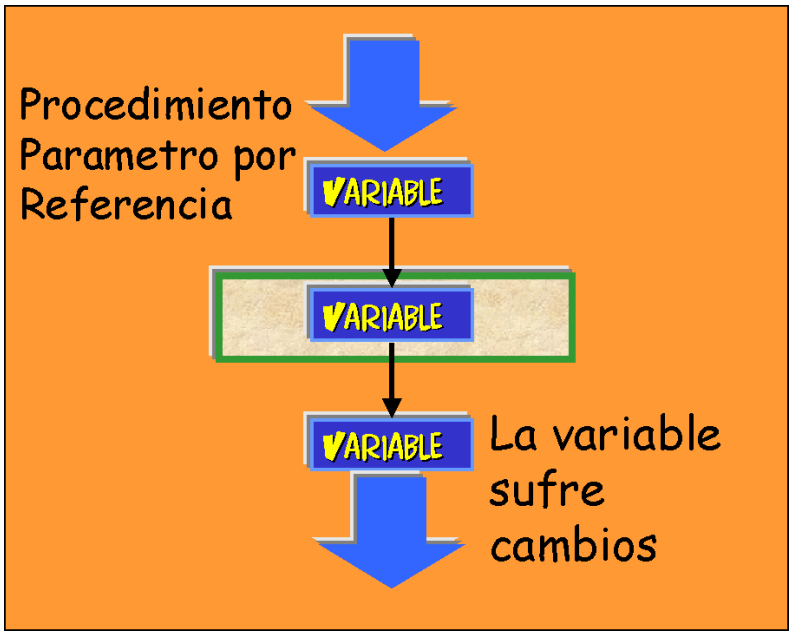
```
/*
*ejemplo6_3.c
*/

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

void intercambia(int A, int B) {
  int aux;
  printf("\nA=%i y B=%i en la función\n",A,B);
  aux=A;
  A=B;
  B=aux;
  printf("\nA=%i y B=%i en la función\n",A,B);
}
main()
{
int A,B;
  srand(time(NULL));
  A = rand();
  B = rand();
  printf("\nA=%i y B=%i \n",A,B);
  intercambia(A,B);
  printf("\nAhora valen A=%i y B=%i \n",A,B);
}
```

La función recibe una copia de los valores de A y B. Después intercambia los valores de las variables locales de la función. Pero cuando la función acaba, estos valores no se copian de vuelta en las variables de la función principal, sino que se pierden.

Para que una función pueda cambiar los valores de las variables que se le pasan como parámetros, es necesario que el paso de parámetros sea **por referencia**. En este caso, lo que la función recibe es el dato original y no una copia de éste. En C esto se consigue



pasándole a función un *puntero* a cada parámetro, esto es, la dirección en memoria de la variable. De esta forma, la función puede acceder a la variable original y cambiar su valor. Para pasar variables por referencia se les antepone el carácter & en la llamada (de esta forma se pasa la dirección de memoria donde se halla la variable y no el valor contenido en dicha dirección de memoria) y en la función se emplean los parámetros con un * delante (indicando que se accede al contenido de la dirección de memoria apuntada por el puntero). La función intercambia quedaría así:

```
/*
*ejemplo6_4.c
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void intercambia(int *A, int *B) {
  int aux;
  printf("\nA=%i y B=%i en la función\n",*A,*B);
  aux=*A;
  *A=*B;
  *B=aux;
  printf("\nA=%i y B=%i en la función\n",*A,*B);
}
main()
{
int A,B;
  srand(time(NULL));
  A = rand();
  B = rand();
  printf("\nA=%i y B=%i \n",A,B);
  intercambia(&A,&B);
  printf("\nAhora valenseis A=%i y B=%i \n",A,B);
}
```

La expresión *A significa “lo apuntado por A”; y la declaración int *A significa “lo apuntado por A es un entero”. En la función, la variable A a secas contiene &A, o sea, “la dirección de memoria de A”.

2.4 Independencia de las funciones

Es deseable que una función sea lo más independiente posible del resto del programa, de forma que un cambio en el programa no implique hacer cambios en la función, y viceversa. Por ejemplo, si en el programa ejemplo6_2.c la función evaluar obtuviese el punto donde se debe evaluar el polinomio a partir de una variable global y no se le pasase como argumento:

```
/*
*ejemplo6_5.c
*/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int x;

int evaluar() {
  int y = 4* pow(x,2)+ 3*x +5;
  return y;
}
main()
{
  x = rand();
  srand(time(NULL));
  printf("La función 4x^2 + 3x + 5 en el punto %i vale %l
\n",x,evaluar());
}
```

la función evaluar será menos independiente: dependerá de que la variable global se siga llamando x; es más, no sirve para calcular el valor del polinomio en otro punto que no sea el contenido en dicha variable. Si empleásemos esta función en otro programa, tendríamos que tener presente que el valor del punto tiene que estar guardado en una variable global llamada x. Claramente esta modificación es perjudicial, ya que la función tal como estaba antes era totalmente independiente de los nombres de las variables que recibe como parámetros.

Como regla general y, para que una función sea lo más independiente posible, sólo debe usar en sus operaciones los parámetros que recibe y las variables locales que necesite, sin utilizar ninguna variable global.

2.5 Declaración de funciones: prototipos

Hasta ahora siempre hemos escrito las funciones antes del código que las usa (por eso el main siempre es la última función que escribimos). De este modo al compilar el programa la función desarrollada por el programador está definida antes de que aparezca el primer acceso a la misma. Sin embargo, se pueden poner las definiciones de las funciones después del código que accede a ellas. Para ello es necesario incluir una declaración de la función antes de su primer acceso. Las declaraciones de funciones aparecen usualmente al comienzo del programa delante de todas las funciones definidas por el programador (incluida main). De esta forma informamos al compilador de todas las funciones que van a ser definidas en el archivo, aunque el código de las mismas aparezca después. Así, hasta ahora nuestros programas seguían el siguiente esquema:

```
#include <stdio.h>
long factorial(int n)
{
......
/* codigo de la funcion factorial */
}
main()
{
.......
x=factorial(3);
}
```

Podemos escribir la función factorial después de la función main siempre que incluyamos antes del main el prototipo de la función factorial:

```
#include <stdio.h>
long factorial(int);
main()
{
.......
x=factorial(3);
}
long factorial(int n)
{
......
/* codigo de la funcion factorial */
```

```
}
```

Nótese que en el prototipo se puede omitir el nombre de los argumentos. Sólo es necesario incluir el tipo que devuelve la función y el tipo de los argumentos, pero no sus nombres. Vemos cómo quedaría el código del ejemplo 6_2 Con este cambio:

```
/*
*ejemplo6_6.c
*/
#include <stdio.h>
#include <stdlib.h>

int evaluar(int);

main()
{
  int x;
  x = rand();
  printf("La función 4x^2 + 3x + 5 en el punto %i vale %i
\n",x,evaluar(x));
}
int evaluar(int x) {
  x = 4* pow(x,2)+ 3*x +5;
  return x;
}
```

Cuando la función llama otra función que no ha sido definida anteriormente algunos compiladores asumen que la función devuelve un entero (int) y que todos sus parámetros son también del mismo tipo, lo cual puede que no se corresponda con la definición de la función. Es por ello que es recomendable declarar todos los prototipos de las funciones al principio de cada fichero.

3 Modificadores de almacenamiento de las variables

Además de por su tipo de dato, las variables se caracterizan por su *tipo de almacenamiento*. Éste viene dado por la permanencia de la variable y su ámbito dentro del programa. Existen cuatro tipos básicos de almacenamiento: automático (auto), externo (extern), estático (static) y registro (register). Veamos en más detalle cada uno de ellos:

3.1 register

Se pueden almacenar algunas variables en los registros del procesador mediante la inclusión de register en su definicion: *register int a;* Como el número de registros es limitado, sólo un cierto número de variables pueden ser almacenadas de esta forma. Por ello, no tenemos garantías de que una variable definida como register sea realmente almacenada en un registro del procesador. Sólo se pueden almacenar en registros variables definidas dentro de funciones y no es posible acceder a la dirección de memoria de las variables de tipo registro a partir del operador dirección.

Para algunos programas el tiempo de ejecución se puede reducir considerablemente si ciertos valores pueden almacenarse dentro de los registros. Esto se debe a que se ahorran una gran cantidad de accesos al bus de sistema para intercambiar datos entre la CPU y la memoria. El número de variables que se pueden almacenar en los registros depende tanto del compilador de C como de la máquina concreta, aunque lo normal es tener dos o tres variables registro por función.

3.2 auto

Las variables automáticas se declaran siempre dentro de una función y su ámbito se restringe a dicha función, esto es, no pueden ser accedidas desde ninguna otra función. Las variables automáticas definidas en funciones diferentes serán independientes las unas de las otras aunque compartan el nombre. Todas las variables definidas dentro de una función por defecto se interpretan como automáticas, a no ser que se indique lo contrario (esto es, no es necesario definir algo como *auto int a;*).

Si una variable automática se inicializa con un valor la inicialización se realizará cada vez que se ejecute el cuerpo de la función, y el valor de la variable en una ejecución anterior no se conservará en la siguiente ejecución. Si una variable automática no es inicializa de ninguna manera su valor es impredecible.

3.3 extern

Las variables externas son aquellas que se definen fuera del cuerpo de las funciones. Son accesibles desde la línea que se definen hasta el fin del archivo; con frecuencia también

son accesibles por todos los módulos del programa, lo que les vale el nombre de "variables globales". Éstas variables, si son de tipo numérico, son inicializadas a 0 de un modo automático por el compilador.

Este tipo de variable permite transferir información entre distintas funciones sin necesidad de usar argumentos, ya que diversas funciones pueden acceder a un mismo dato. Sin embargo, esto aumenta las dependencias de las funciones entre sí y, en muchas ocasiones, limita su reutilización en otros problemas.

Al trabajar con variables externas hay que distinguir entre definiciones de variables externas y declaraciones de variables externas. Para definir una variable externa no es necesario emplear el especificador de tipo de almacenamiento "extern" (de hecho muchos compiladores de C prohiben su uso). Si la función requiere una variable que ha sido definida antes en el programa la función podrá acceder libremente a ella. Si la variable se define después o en un módulo distinto para qué la función pueda acceder a la variable deberá realizar una declaración de dicha variable. La declaración comienza con el especificador del tipo de almacenamiento y el nombre y tipo de datos de la variable tienen que coincidir con la correspondiente definición de variable externa que aparece fuera de la función. Así, el código siguiente no compila correctamente porque la variable i no es reconocida en la función main:

```
main()
 {
  printf ("%i",i);
}

  int i = -1000;
```

si incluimos una declaración de la variable este problema se solucionará:

```
main()
 {
 extern int i;
  printf ("%i",i);
}

  int i = -1000;
```

En una declaración de variable externa nunca se puede realizar una asignación de valor inicial; así, el siguiente código da un error al compilar por este motivo:

```
main()
 {
 extern int i=1;
  printf ("%i",i);
}

  int i = -1000;
```

3.4 static

En esta sección suponemos que el programa está formado por un único archivo. Más adelante abordaremos el caso de los programas multi archivo.

En el caso que nos atañe, las variables static se definen dentro de funciones y, por ello, su ámbito se reduce a la función en la que son definidas. A diferencia de las variables automáticas, retienen su valor entre sucesivas llamadas a la función en la que son definidas. Si se incluyen valores iniciales en su declaración (inicialización), estos valores iniciales se asignarían sólo la primera vez que la función es invocada. Se definen de la forma: static int k=3;. Aquellas variables estáticas a las que no se le asigne un valor inicial se inicializarán al valor cero.

A continuación presentamos un programa que permitirá comprender mejor el funcionamiento de estas variables:

```
/*
*ejemplo6_7.c*/
#include<stdio.h>
int b;  //VARIABLE GLOBAL
int sub();

int main()
 {
  printf("Introduzca un Numero: ");
  scanf("%d",&b);
  sub();
  sub();
  sub();
  sub();
 }

int sub()
  {
   static int c, i=1;
   int d= 0;
   b=b+2;
   c=b+1;
```

```
   d++;
   printf("\nLlamada %d: c= %d, b= %d, d=%d ",i, c,b,d);
   i++;
   return c;
  }
```

4 Programas multi archivo

En C un solo programa puede constar de varios archivos. De este modo se da soporte a la programación modular: cada archivo puede representar un módulo; habrá un módulo principal (aquél que contiene la función main) que arranca la ejecución del programa y éste va delegando sus tareas en otros módulos (archivos), los cuales a su vez pueden apoyarse en más módulos. Cada uno de estos módulos está compuesto por un conjunto de variables y funciones. Es posible decidir qué variables y qué funciones de un módulo serán visibles para los demás; esto es, podemos determinar cuál será la interfaz que este módulo ofrezca a los demás. El resto del contenido del módulo será una caja negra que los programadores de los demás módulos no necesitan conocer para realizar su trabajo.

Para generar archivo ejecutable los distintos archivos se compilan de forma separada y a continuación se enlazan para formar un programa ejecutable.

A continuación veremos cómo se determina qué variables y qué funciones forman parte de la interfaz de cada módulo.

4.1 Funciones

Una función puede ser tanto estática como externa. Una función externa podrá ser accedida desde cualquier módulo del programa, mientas que la función estática sólo podrá ser accedida desde el archivo en el cual se define. Para determinar el tipo de almacenamiento de una función se emplea la palabra "extern" o "static", según corresponda, al principio de la definición de la función:

```
/*
*ejemplo6_7.c
*resta.c
*/
static float rest(float dato1, float dato2)
```

```
{
  return dato1 - dato2;
}

extern float restar(float dato1, float dato2)
{
  return rest(dato1, dato2);
}
```

Por defecto las funciones son externas. Esto quizás no haya sido la mejor decisión de diseño, ya que en la programación estructurada, idealmente, cada módulo debe ocultar el mayor número de detalles posibles de su implementación y ofrecer una interfaz lo más simple posible. En lo referente a las funciones, esto se traduce en mostrar al exterior el mínimo número posible de funciones, por lo que es conveniente en los programas multi archivo definir todas aquellas funciones que no se vayan a emplear desde otros módulos como estáticas.

Cuando se define una función en archivo y se accede a ella desde otro en el segundo se debe incluir una declaración de la función. Éstas declaraciones se suelen colocar al principio del archivo y es recomendable (aunque no imprescindible) que lleven el modificador extern para indicar que esa función no se encuentra definida en este archivo, sino en otros externo.

```
/*
*ejemplo6_7.c
*/
#include <stdio.h>

extern float sumar(float, float);
extern float restar(float, float);
main()
{
  float suma, resta, dato1=15, dato2=20;
  suma = sumar(dato1, dato2);
  resta = restar(dato1,dato2);
  printf("La suma es %f y la resta %f", suma, resta);
  getch();
}
```

Si un archivo tiene una función estática y deseamos declarar su prototipo habrá que incluir la palabra "static" en dicha declaración.

4.2 Variables

En un programa multi archivo se pueden definir variables externas (globales) en un archivo y acceder a ellas desde otros. Para ello debemos declarar la variable en aquellos archivos que quieran acceder a ella empleando la palabra "extern", del mismo modo que ocurría con las funciones.

En la definición de las variables se puede incluir valores iniciales; cualquier variable externa a la que no se le asigne un valor inicial se inicializará a cero. Al igual que con las funciones, no es necesario emplear la palabra "extern" en la definición de la variable (de hecho, nuevamente, muchos compiladores dan errores si se incluye).

El valor asignado a una variable externa puede alterarse dentro de cualquier archivo donde se puede acceder a dicha variable. Los cambios serán vistos en todo los archivos que estén en el ámbito de la variable. Esto convierte las variables externas en los programas multi archivo en una forma de compartir información entre varios módulos y, al mismo tiempo, en una forma de crear dependencias (nunca deseables) entre ellos.

En un archivo, las variables que estén definidas fuera de cualquier función pueden definirse como "static". El ámbito de estas variables va desde la línea de la definición hasta el final del archivo. El uso de variables externas (globales) estáticas dentro de un archivo permite que un grupo de variables esté oculto al resto de los módulos.

4.3 Definiendo la interfaz de un módulo: archivos de cabecera

Del mismo modo que no es deseable que un módulo ofrezca más detalles sobre su implementación que aquellos que son estrictamente necesarios para acceder a su funcionalidad, es deseable especificar lo que se desea exportar de cada módulo del modo más simple posible. Para ello, se pueden emplear los archivos de cabecera: basta con construir para cada módulo un archivo de cabecera que contenga todas las declaraciones externas que sean necesarias para emplear el módulo. Cuando otro módulo desea acceder a la funcionalidad de éste y bastará con importar el archivo de cabecera correspondiente. Así, podemos crear un archivo de cabecera para el archivo resta.c (que forma parte del ejemplo6_7.c) cuyo nombre será resta.h y su contenido:

```
/*
*ejemplo6_7.c
*resta.h
*/
extern float restar(float, float);
```

de este modo el código del ejemplo6_7.c quedará así:

```
/*
*ejemplo6_7.c
*/
#include <stdio.h>
#include "suma.h"
#include "resta.h"
main(){
  float suma, resta, dato1=15, dato2=20;
  suma = sumar(dato1, dato2);
  resta = restar(dato1,dato2);
  printf("La suma es %f y la resta %f", suma, resta);
}
```

4.3.1 Ejemplo de un programa multi archivo

A continuación presentamos un programa muy simple que, por motivos didácticos, hemos dividido en varios archivos. El programa calcula la suma y la resta de dos números, estando las funciones que calculan la suma y la resta en un archivo separado del que contiene la función principal.

```
/*
*ejemplo6_7.c
*/
#include <stdio.h>
#include "suma.h"
#include "resta.h"
main()
{
  float suma, resta, dato1=15, dato2=20;
  suma = sumar(dato1, dato2);
  resta = restar(dato1,dato2);
  printf("La suma es %f y la resta %f", suma, resta);
  getch();
}
```

```
/*
*ejemplo6_7.c
*sumar.h*/
```

```
extern float sumar(float, float);
```

```
/*
*ejemplo6_7.c
*resta.h
*/
extern float restar(float, float);
```

```
/*
*ejemplo6_7.c
*suma.c
*/
#include "suma.h"
 float sumar(float dato1, float dato2)
{
  return dato1 + dato2;
}
```

```
/*
*ejemplo6_7.c
*resta.c
*/
#include "resta.h"
static float rest(float dato1, float dato2)
{
  return dato1 - dato2;
}

extern float restar(float dato1, float dato2)
{
  return rest(dato1, dato2);
}
```

El incluir el archivo cabecera que describe la interfaz de un archivo en el propio archivo descrito no es imprescindible, pero le simplifica la labor al compilador y le permite comprobar que las definiciones del archivo cabecera son correctas.

5 Principales funciones de las librerías estándar

Del mismo modo que cuando escribimos un programa multi archivo los distintos archivos objeto fruto de la compilación deben enlazarse entre sí para formar el ejecutable final, nuestro programa también se enlazará con archivos objeto de la librería del compilador. Para acceder a la funcionalidad de las librerías basta con incluir el correspondiente archivo de cabecera, el cual contendrá declaraciones de las partes públicas del archivo, y el enlazador se encargará de realizar el resto del trabajo.

A continuación se recogen las funciones más comunes de las librerías estándar de C.

5.1 Standard C Math

- abs valor absoluto
- acos arco coseno
- asin arco seno
- atan arco tangente
- atan2 arco tangente usando signos para determinar los cuadrantes
- ceil devuelve el entero más pequeño no menor que el valor que se le pasa
- cos coseno
- cosh cosen hiperbólico
- exp devuelve "e" elevador o argumentos que se le pasa
- fabs valor absoluto de números reales
- floor devuelve mayor entero no menor que la cantidad que se le pasa
- labs valor absoluto de un long
- log logaritmo natural
- log10 logaritmo en base 10
- pow devuelve el primer parámetro elevado segundo
- sin seno
- sinh seno hiperbólico
- sqrt raíz cuadrada
- tan tangente
- tanh tangente hiperbólica

5.2 Standard C String and carácter

- atof convierte un string en un double (stdlib.h)
- atoi convierte un string en un integer (stdlib.h)
- atol convierte un string en un long (stdlib.h)
- isalnum devuelve 1 si el carácter argumento es alfanumérico (ctype.h)
- isalpha devuelve 1 si el carácter argumento es alfabético (ctype.h)
- iscntrl devuelve 1 si el carácter argumento es un carácter de control (ctype.h)
- isdigit devuelve 1 si el carácter argumento es un dígito (ctype.h)
- islower devuelve 1 si el carácter argumento está en minúscula (ctype.h)
- isspace devuelve 1 si el carácter argumento es el espacio en blanco (ctype.h)
- isupper devuelve 1 si el carácter argumento está en mayúscula (ctype.h)
- tolower convierte un carácter a minúscula (ctype.h)
- toupper convierte un carácter en mayúscula (ctype.h)
- strcat concatena dos strings (string.h)
- strcmp compara dos strings (string.h)
- strcpy se copia una cadena de caracteres en otra (string.h)
- strlen devuelve la longitud de una cadena de caracteres (string.h)
- strncat concatena dos cadenas de caracteres (string.h)
- strtod convierte un string en un double (string.h)
- strtol convierte un string en un long (string.h)
- strtoul convierte un string en un unsigned long (string.h)

6 Ejercicios

1. Escribe una función que devuelva el doble del valor real que se le pasa como argumento y probar su funcionamiento en un programa.
2. Escribir una función que calcule las raíces reales de una ecuación de segundo grado. Los parámetros que se pasarán a la función serán los tres coeficientes de la ecuación.
3. Escribe una función que acepte como parámetro una letra minúscula y devuelva la correspondiente letra mayúscula. No podrá usarse ninguna función de las librerías estándar de C. El programa debe funcionar para toda las letras comprendidas entre la a y la z a excepción de la ñ. Ayuda: emplear la tabla de caracteres ASCII.
4. Modificar el programa anterior de tal modo que la función modifique el carácter que se le pasa como argumento y lo convierta en mayúscula.
5. Construye dos funciones que calculen una el producto escalar y otra el producto vectorial de dos vectores y probar su funcionamiento un programa.
6. Modificar el programa del ejercicio anterior de tal modo que las dos funciones estén definidas en un archivo aparte del programa que las emplea. No emplees archivos de cabecera.
7. Modificar el programa del ejercicio anterior empleando archivos de cabecera.
8. Escribe la función que acepte como argumento un número real y devuelva la raíz cuadrada del logaritmo en base 10 del menor número entero mayor que el argumento de la función.
9. Escribe una función que calcule los pagos mensuales de una hipoteca a partir del capital del préstamo, el interés anual y el número de años; y otra que dada la cuota a pagar todo los meses, el interés y el número de años devuelva el capital máximo que se puede prestar. Escribir ambas funciones en un archivo independiente que exporte su interfaz mediante un archivo cabecera y úsalas desde otro archivo. Para este cálculo se emplea la fórmula: cuota$=\frac{C \cdot R}{1-(\frac{1}{1+R})^N}$, donde C es el capital del préstamo, R es la tasa de interés mensual en tanto por uno y N es el número de pagos a realizar.