Tema 11

El lenguaje C mediante ejemplos

Este capítulo pretende repasar las construcciones más habituales del lenguaje C. No pretende ser un tutorial del mismo: para eso hay excelentes obras que realizan este recorrido paso a paso, como las citadas en el capítulo anterior.

A lo largo del capítulo se mostrarán muchos ejemplos y se propondrán ejercicios para que el lector trabaje los conceptos que se introducen. Al final del capítulo se dan las soluciones de todos estos ejercicios.

Un programa en C tiene una estructura estándar bien definida, requerida por el compilador: comienza con declaraciones de lo que se va a emplear (directivas "`#include`"), declaraciones propias particulares del programa, declaraciones y, por último, definiciones de las funciones del programa. En particular, todo programa C tiene una función, llamada "`main`" que es, como su nombre indica, la función principal del programa: la primera que es ejecutada. Más en lo particular, cada instrucción de C acaba en "`;`" si es una instrucción autónoma, o va seguida de un "bloque" de instrucciones abrazado por llaves "`{`" y "`}`": así es como, por ejemplo, se indican las instrucciones que corresponden a una función.

¡Así no se debe escribir un programa en C!

Fuera de esta estructura general impuesta, un programa C debe ser legible por alguien más que el que lo escribe. El siguiente programa es correcto en su sintaxis y compila sin errores (salvo algún "Warning"):

```
long k=4e3,p,a[337],q,t=1e3;
main(j){for(;a[j=q=0]+=2,--k;)
for(p=1+2*k;j<337;q=a[j]*k+q%p*t,a[j++]=q/p)
k!=j>2?:printf("%.3ld",a[j-2]%t+q/p/t);}
```

Es más, al ejecutarlo produce un resultado (¡sorprendente!). Sin embargo, cuesta unos cuantos minutos hacerse una idea de cómo lo hace: el ordenador lo va a entender, pero si tuviéramos que modificarlo, no sabríamos por dónde empezar.

¡Así se debería escribir un programa en C!

El programa anterior, sin pérdida de efectividad, pero más fácilmente legible y con una salida más adecuada se escribiría como se muestra en el listado 11.1. La diferen-

cia con el anterior es evidente: el sangrado de las líneas indica (además de las llaves) los bloques de instrucciones que se ejecutan secuencialmente, los comentarios (entre "/*" y "*/") facilitan la comprensión de qué se está haciendo, ...

Listado 11.1: Versión bien documentada y formateada de un programa que calcula el número π.

```
1  #include <stdio.h>
2
3  /** Programa PI
4   *  Calcula 1002 cifras decimales del número PI
5   *  Algoritmo: pi3
6   *  Referencia: (buscarla)
7   *  Autor: (anónimo); Adaptación: DRP
8   *  Fecha: 20/10/2010
9   *  Observaciones: antes era así
10  *   long k=4e3,p,a[337],q,t=1e3;
11  *   main(j){for (;a[j=q=0]+=2,--k;)
12  *   for (p=1+2*k;j<337;q=a[j]*k+q%p*t,a[j++]=q/p)
13  *   k!=j>2?:printf("%.3d",a[j-2]%t+q/p/t);}
14  */
15 int main(int argc, char** argv) {
16         long k=4000; /* cuatro mil                    */
17         long t=1000; /* mil                          */
18         long a[337]; /* array de resultados parciales */
19         long p;      /* denominador de la serie modular*/
20         long q;      /* numerador de la serie modular */
21         long d;      /* variable auxiliar            */
22         int j;       /* contador                     */
23
24         /* PI=3. ... */
25         printf("PI=3.");
26
27         /* Bucle principal
28          * Nota: sólo imprime ternas de cifras en los
29          *       últimos dos pasos de cálculo
30          */
31         while(k>1)
32         {
33                 k--;     /* paso del bucle de impresión */
34                 q=0;     /* inicializar contador modular*/
35
36                 a[0]+=2; /* avanzar en a[0] por cada k  */
37                 p=1+2*k; /* inicializar el denominador
38                             de la serie modular         */
39
40                 /* bucle de cálculo de a[] */
41                 for(j=0; j<337; j++)
42                 {
```

```
43                          /* condición de cálculo completado:
44                             dos últimos pasos en k */
45                          if ( (j>2 && k==1) || k==0 )
46                          {
47                                  /* todas las divisiones
48                                     son enteras! */
49                                  d=a[j-2]%t+(q/p)/t;
50
51                                  /* imprimir las siguientes
52                                     3 cifras */
53                                  printf ("%.3ld",d);
54                          }
55                          /* actualizar el numerador de
56                             la serie */
57                          q=a[j]*k+q%p*t;
58                          /* guardar el valor en a (se
59                              utilizará dos j-pasos más
60                             adelante) */
61                          a[j]=q/p;
62                  }
63          }
64
65          /* salto de línea final */
66          printf ("\n");
67
68          /* Informar al S.O. de que no ha habido error
69           * al finalizar el programa
70           */
71          return 0;
72  }
```

En la realidad, nunca se hace ni como en el primer y oscuro ejemplo, ni como en éste tan bien documentado: parte de la explicación del programa y de su documentación se entiende que la proporciona el propio código C.

11.1. Estructura básica de un programa: la función “main”

Aunque ya la hemos venido viendo, la estructura básica de un programa en C es la del listado 11.2.

Listado 11.2: El programa “Hola mundo” (versión estándar ANSI C).

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv)
4  {
5          printf ("Hola mundo\n");
6
```

```
7         return 0;
8 }
```

La diferencia principal con el listado anterior es que aquí la función "main" recibe argumentos. El primero es un valor entero (tipo "`int`"); el segundo un vector de cadenas. Son dos variables usualmente llamadas así, "`argc`" y "`argv`". Otra sintaxis alternativa es:

```
int main(int argc, char* argv[])
```

El tipo de variable "`char*`" se refiere a una cadena, mientras que "`[]`" quiere decir que lo que recibe la función "`main`" es un vector (o "array") de dicho tipo de variables. A cada uno de los valores en el vector se accederá indicando el nombre de éste, seguido de los corchetes y, entre éstos, el número de componente del vector. Así, por ejemplo, el nombre del ejecutable es recibido como el primer elemento del vector "`argv`"; nos referiremos a dicho valor como "`argv[0]`".

Ejercicio 11.1. Modifíquese el programa anterior cambiando la llamada a "`printf`" por la siguiente:

```
printf ("Hola␣mundo,␣me␣llamo␣%s\n", argv[0]);
```

Aprovéchese esta oportunidad para comprobar el significado de la cadena de formato (primer argumento) de la función "`printf`".

Ejercicio 11.2. Compruébese que `argc` es un entero que contiene el número de argumentos que pasamos por línea de comandos cuando ejecutamos el programa (contando también el propio nombre del programa).

```
printf ("El␣número␣de␣argumentos␣es␣%d\n", argc);
```

El primer argumento de línea de comandos se recibirá en el valor "`argv[1]`". Este valor es el que escribamos a continuación del nombre del programa cuando lo llamemos. Por ejemplo, si hacemos

```
./holamundo Pepito
```

el valor de "`argv[1]`" será la cadena "`Pepito`".

Ejercicio 11.3. Modifíquese el programa para que salude, en lugar de "al mundo entero", al usuario que le indica su nombre como primer argumento en la línea de comandos (como en el ejemplo anterior). En este caso, "`printf`" será llamado como:

```
printf ("Hola␣%s,␣me␣llamo␣%s\n", argv[1], argv[0]);
```

11.2. Declaración de variables, asignación de valores, operaciones básicas e impresión de resultados

La declaración de variables, indicando su tipo y su nombre, es obligatoria en C. En el listado 11.3 se declaran tres variables de tipo "`float`", se asignan valores a dos de ellas y se calcula la tercera como suma de las dos primeras.

Nótese la sintaxis de los números (notación decimal y científica) y la de la asignación con el signo "=".

Además de la operación de la suma "+", se pueden emplear otros operadores binarios (que actúan sobre el valor dado por la expresión anterior y posterior a él): "-", "*", "/" [1]. Estos operadores binarios se aplican usando las "reglas de precedencia" habituales en álgebra. Por ejemplo, "a*b+c" equivale a "(a*b)+c".

El operador "-" también actúa como operador "unario". Así, "-a" significa (sorprendentemente), "el valor de la variable a con signo ($\pm$) cambiado".[2]

Listado 11.3: El programa que suma dos números reales.

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv)
4  {
5          float a, b, c;
6
7          a=1.5;
8          b=-3.2e-2;
9          c=a+b;
10
11         printf ("%g+%g=%g\n", a, b, c);
12
13         return 0;
14 }
```

Ejercicio 11.4. Búsquese información sobre la función "atof" que interpreta una cadena como un valor real (archivo H en el que está declarada, argumentos que recibe y su tipo). Utilícese para modificar el programa anterior en un programa que recibe por línea de comandos los dos números y muestra la suma y su resultado.

Ejercicio 11.5. Declárense las variables "a", "b" y "c" como enteras y utilícese la función "atoi" para asignarle valores (búsquese el archivo H en el que está declarada, los argumentos que recibe y su tipo). Cámbiese la operación de suma ("+") a división ("/"), y compruébese el comportamiento del programa.

11.3. Control de flujo: "if... else"

Hasta ahora la ejecución de los programas ha sido lineal: una instrucción tras otra. El "flujo" de un programa consiste en el orden en que sus instrucciones van siendo ejecutadas y si lo son o no, en función de lo que indique el "usuario" de dicho programa.

[1] A diferencia de otros lenguajes, en C no existe un operador de exponenciación; para ello se utilizará una función: "pow".

[2] En C también se puede utilizar "+" como operador "unario". Obviamente "+a" (valor de la variable a sin cambio de signo) indica lo mismo que "a". Sin embargo, a veces, puede ser conveniente esta notación para enfatizar lo que se está escribiendo.

La modificación más sencilla al flujo consiste en decidir si se ejecuta una instrucción u otra según una condición lógica.

En el listado 11.4 se calcula el valor absoluto de la diferencia entre dos números reales. Se utiliza la instrucción "`if...else`" para ejecutar un bloque de instrucciones u otro (bloques, entre "`{...}`", con una única instrucción en ellos) de modo que se garantice el signo adecuado en el resultado.

La condición se construye con operadores de comparación: "<", ">", "<=", ">=", "`!=`", "`==`". Nótese la notación peculiar del C para el operador "*distinto de*" ("`!=`") y para el operador "*igual a*" ("`==`"), que no se debe confundir con el operador "`=`" que sirve para asignar a la variable a su izquierda el valor de la expresión a su derecha. Los operadores de comparación tienen menos precedencia que los de suma o resta; esto no debe parecer evidente, ya que en C los resultados de operaciones lógicas se tratan como si fueran valores enteros. Es decir, se puede calcular algo así como "`(4 || 6) + (7 || 8)`"... otra cosa es el significado que tenga, pero en principio es legal.

Los resultados de las comparaciones se pueden combinar posteriormente con los operadores lógicos: "||" (*disyunción*, OR), "&&" (*conjunción*, AND), "^" (*disyunción exclusiva*, XOR). También se pueden "*negar*" con el operador unario "`!`". Estos operadores son los últimos en ser evaluados.

Listado 11.4: El programa que calcula la distancia entre dos números reales.

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv)
4  {
5          float a, b, c;
6
7          a=1.5;
8          b=3.2;
9
10         if ( a<b ) {
11                 c=b-a;
12         } else {
13                 c=a-b;
14         }
15
16         printf ("|%g-%g|=%g\n", a, b, c);
17
18         return 0;
19 }
```

Ejercicio 11.6. Modificar el listado anterior usando la función "`atof`" para leer los valores de "`a`" y "`b`" desde línea de comandos. Eliminar también la alternativa "`else`" haciendo primero la operación "`a-b`", comprobando el signo de "`c`" y, si no es adecuado, cambiándolo con el operador unario "`-`".

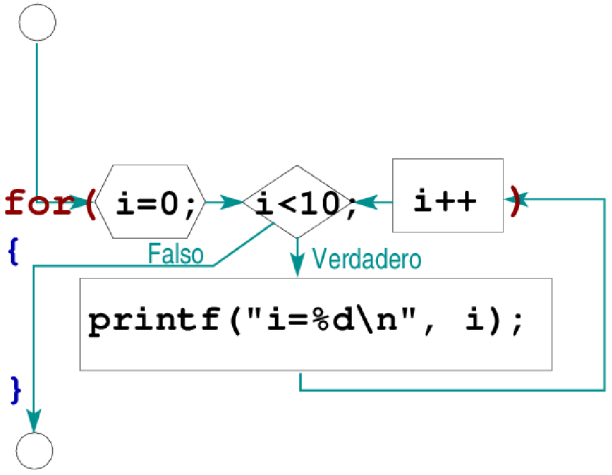

Figura 11.1: Flujo de ejecución del bucle “for”.

11.4. Bucle “for”

La mayor utilidad de un programa de ordenador es realizar una o varias operaciones tipo muchas veces, variando los valores de las variables involucradas. Una forma de hacer esto es usar bucles.

El más completo de los bucles es el bucle “for”. El flujo de ejecución del bucle “for” es como se muestra en la figura 11.1. Un ejemplo sencillo (cálculo de los seis primeros números pares), se muestra en el listado 11.5.

Listado 11.5: El programa que calcula los pares menores o iguales que 10.

```
1  #include <stdio.h>
2
3
4  int main(int argc, char** argv)
5  {
6
7          int n;
8
9          for(n=2; n<=10; n+=2)
10         {
11                 printf ("%d\n", n);
12         }
13
14         return 0;
15 }
```

Es habitual emplear en este tipo de bucles la sintaxis incremental, “n++”, que indica que la variable “n” (usualmente de tipo entero) sea incrementada en una unidad. Otra sintaxis habitual es la de incrementos no unitarios, de la forma “n+=2”; en este caso, la variable “n” es incrementada en dos unidades (como en el listado 11.5). Éste es un caso particular de las “autoasignaciones” que tienen operadores de la forma “-=”, “*=”, “/=”. Por ejemplo, un productorio se programaría a base de operaciones de la siguiente forma:

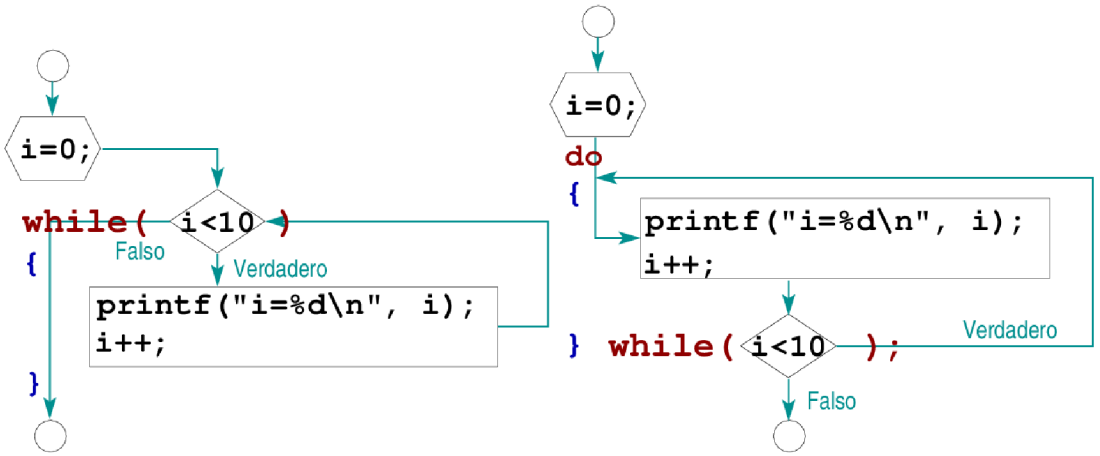

Figura 11.2: Flujo de ejecución de los bucles “`while`” y “`do...while`”.

productorio$*$=factor;

Ejercicio 11.7. Calcular el factorial de un número entero indicado por línea de comandos. Empléese para guardar este factorial un “`long int`”, para tener posibilidad de valores más grandes en el resultado[3] ¿Qué sucede para números enteros grandes (mayores que 17)?

Por supuesto, todo lo anterior no se ve limitado (como sucede en Fortran) a valores enteros de la variable del bucle. Para ello, pruébese el siguiente ejercicio.

Ejercicio 11.8. Calcúlense los puntos que dividen el intervalo $[0;1]$ en 100 subintervalos.

11.5. Bucles “while” y “do...while”

El flujo de ejecución de los bucles “`while`” y “`do...while`” es como se muestra en la figura 11.2. Estos bucles no añaden demasiado al bucle “`for`” ya visto.

Ejercicio 11.9. Rescribir el programa anterior para calcular los números pares menores que 10, usando un bucle “`while`” o “`do...while`”.

Ejercicio 11.10. Rescribir el programa anterior para calcular los puntos que dividen el intervalo $[0;1]$ en 100 subintervalos, usando un bucle “`while`” o “`do...while`”.

11.6. Selección con “switch...case...default”

En muchas ocasiones habrá que comparar una variable entera con ciertos valores que ésta adquiera, y ejecutar unas instrucciones u otras en función del significado que se les dé a estos valores[4].

[3] La efectividad de esta declaración “`long int`” dependerá de la “arquitectura” de nuestro procesador: 32 o 64 bits

[4] Una variable de este tipo se llamaría en estadística una variable “categórica”, ya que cada valor que toma pertenece a una categoría, de entre un número finito de ellas.

Un pedazo de código que podría hacer esto se basaría en una serie de "`if`" encadenados:

```
if ( x == 1 ) {
    ...
} else if ( x==2 ) {
    ...
} else if ( x==3 ) {
    ...
} else {
    ...
}
```

Sin embargo, es más elegante sustituir este código por una serie de "casos" en la siguiente forma equivalente al código anterior:

```
switch( x ) {
case 1:
    ...
    break;
case 2:
    ...
    break;
case 3:
    ...
    break;
default:
    ...
}
```

Este ejemplo, además de para explicar una *estructura de control* (de flujo) de C, sirve también para introducir la instrucción "`break`". Esta instrucción obliga a salir de la estructura de control de flujo en la que se esté[5]. En "`switch...case`", sale del ámbito (las "llaves") del "`switch`". Si se utilizara en un bucle "`for`", saldría del bucle "`for`". Si se utilizara en un bucle "`while`", finalizaría en ese punto el bucle, etc.

Fuera de "`switch...case`", el uso de "`break`" se debe evitar porque oscurece la lógica del programa. Sin embargo, muchas veces es la mejor manera de dar por terminado un bucle desde dentro...

El ejemplo de uso de "`switch...case`" más tradicional, es el uso en menús interactivos. Los menús interactivos más simples utilizan la función "`printf`" primero para mostrar distintas opciones de un menú, y después para pedir que se seleccione una opción. La opción seleccionada se lee en tiempo de ejecución usando la función "`scanf`".

La función "`scanf`" se parece a "`printf`": tiene una cadena de formato (qué datos se van a leer) y un argumento por cada dato que se leerá. La diferencia con "`printf`" es que, en lugar de indicar la variable "`sel`", por ejemplo, se indica su dirección de memoria "`&sel`". Como veremos más adelantes, esta dirección de memoria se representa en C por un puntero (algo declarado, por ejemplo, como "`int *ptr`"), pero en "`scanf`" casi siempre aparece con el operador de dirección "`&`" precediendo al nombre de la variable en la que se guardará el valor.

[5] Puede estar en todos los bucles, en "`switch...case`", pero no en "`if`".

Como ejemplo, véase el listado 11.6.

Listado 11.6: El programa que ofrece un menú y comenta la opción elegida (una y otra vez, hasta que se pide salir).

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv)
4  {
5          int sel;
6
7          printf ("1. Opción A\n");
8          printf ("2. Opción B\n");
9          printf ("3. Opción C\n");
10         printf ("0. Salir\n");
11
12         do {
13                 printf ("Opción: ");
14                 scanf("%d", &sel);
15
16                 switch(sel)
17                 {
18                 case 0:
19                         break;
20                 case 1:
21                         printf ("Buena opción la 1\n");
22                         break;
23                 case 2:
24                         printf ("No me parece tan buena la"
25                                 " 2\n");
26                         break;
27                 case 3:
28                         printf ("La 3, definitivamente,"
29                                 " no me gusta\n");
30                         break;
31                 default:
32                         printf ("Esta opción no se ha"
33                                 " ofrecido ...\ n"
34                                 "Pruebe de nuevo\n");
35                 }
36
37             } while(sel!=0);
38
39         return 0;
40 }
```

11.7. Vectores y matrices

Ya hemos visto un vector en C: los argumentos de línea de comandos, que se declaran como "`char* argv[]`", esto es, un vector de punteros a caracteres. Los vectores numéricos son mucho más familiares y útiles.

Un vector se declara en un programa indicando entre los corchetes que siguen al nombre que le damos el número de elementos que contiene dicho vector (la dimensión del espacio al que pertenece, que diríamos en terminología matemática). Así, un vector de 7 números enteros, se declarará como

```
int vector[7];
```

y a su *quinto* elemento se accederá como en

```
x=vector[4];
```

porque en C se empieza a contar por el 0 (el quinto elemento irá precedido de los 0, 1, 2 y 3).

En el listado 11.7 se muestra un ejemplo sencillo de uso de vectores de números reales.

Listado 11.7: El programa que suma dos vectores de números reales.

```
1  #include <stdio.h>
2
3
4  int main(int argc, char** argv)
5  {
6          int i;
7          float x[]={1.0, 2.0, 3.0};
8          float y[3];
9          float z[3];
10
11         y[0]=3.0;
12         y[1]=2.0;
13         y[2]=1.0;
14
15         for(i=0; i<3; i++)
16         {
17                 z[i]=x[i]+y[i];
18         }
19
20         printf ("La suma de vectores da: ");
21
22         printf ("[%g", z[0]);
23         for(i=1; i<3; i++)
24                 printf (", %g", z[i]);
25         printf ("]\n");
26
27         return 0;
28 }
```

Igual que se declaran vectores, se declaran matrices. Por ejemplo, una matriz de 2×2 números reales (con doble precisión, tipo "`double`"), se declarará como

```
double matriz[2][2];
```

y a sus elementos se accederá por fila y columna, como en

```
int i, j;
double M_ij;
...
   M_ij=matriz[i][ j ];
```

Ejercicio 11.11. Calcular el producto de dos matrices de números reales 3×3. Una de ellas codificada en el programa, y la otra pedida por consola, elemento a elemento, usando "`scanf`". Mostrar el resultado debidamente formateado (en una matriz).

11.8. Punteros

En este capítulo se han utilizado dos operadores que precedían a nombres de variables y que no han sido debidamente explicados, "`*`" y "`&`". Hemos visto al declarar los argumentos de la función "`main`" que "`char** argv`" representa un vector de cadenas[6], y que también puede expresarse como "`char* argv[]`" o[7] "`char *argv[]`", que leído "literalmente" representa un vector de punteros a caracteres. Las tres opciones declaran lo mismo y después de esta sección esperamos que se entienda el porqué. También se ha utilizado el operador de dirección "`&`" precediendo al nombre de una variable, de modo que "`&var`" indica la dirección de memoria que ocupa la variable "`var`", es decir, donde está. En esta sección vamos a ver que estos dos operadores están relacionados con una de las herramientas más interesantes del lenguaje C, los punteros.

Para el propósito de este curso y el nivel de programación que se espera conseguir, el uso de punteros no es en absoluto necesario. En muchos casos pueden ser sustituidos por otros elementos del lenguaje C como los vectores o matrices. En otros casos basta con cambiar el modo de almacenamiento de la variable en cuestión. Sin embargo, su uso permite explotar aún más a potencialidad del C y conseguir diseños de programa mucho más elegantes.

En C la dirección de memoria de una variable es, en sí, un tipo de dato. Un puntero es una variable del lenguaje C que puede contener una de esas direcciones de memoria. Por esta razón se denominan punteros, porque "apuntan" o "referencian" a otras variables. Los punteros se declaran de acuerdo con el tipo de dato al que apuntan, de modo general tenemos que su declaración es "`tipo *ptr`", donde "`tipo`" es el tipo de la variable cuya dirección se guardará en "`ptr`". Por ejemplo, si declaramos el puntero "`int *q`", esto quiere decir que el puntero "`q`" apunta a una dirección de memoria en la que se guarda una variable entera. El valor de la variable se obriene como "`*q`". Dos formas alternativas de leer la declaración anterior son: 1) lo que hay en "`q`" (esto es "`*q`") es una variable de tipo "`int`"; 2) "`q`" es el puntero a un valor "`int`".

[6] Recordamos que una cadena (del inglés *string*) es un vector de caracteres y que se define como "`char* cadena`".

[7] Recuérdese que en C los espacios entre operadores y operandos pueden ser ignorados.

Los operadores de indirección "*" y de dirección "&" son operadores inversos. Esto quiere decir que si tenemos una variable "var", se cumple que "*(&var)==var", que es como decir que lo que hay en la dirección de memoria ocupada por "var" es el valor de la propia "var". Por otro lado, si tenemos un puntero "q", se cumple que "&(*q)=q", que viene a decir que la posición en memoria de la variable a la que apunta el puntero "q" es la propia posición indicada por "q". En el ejemplo del listado 11.8 se muestra cómo se puede utilizar un puntero para modificar el valor de una variable a través de su dirección de memoria.

Listado 11.8: Uso de un puntero para cambiar el valor de una variable a través de su dirección de memoria.

```
1  #include <stdio.h>
2
3  int  main(int argc,  char** argv)
4  {
5      double a=43.4;
6      double *q;
7
8      /*asignamos al puntero q la dirección  de memoria de la variable a*/
9      q=&a;
10
11     /*resultado  el  valor  de a*/
12      printf ("El  puntero q apunta al  valor   %lg\n", *q);
13
14     /*modificamos el valor de la  variable  a  la  que apunta el puntero q*/
15     *q=0.3;
16
17     /*resultado  en nuevo valor*/
18      printf ("El  puntero q apunta al  valor   %lg\n", *q);
19
20     /*como la variable  "a" está  en la  misma dirección de memoria*/
21      printf ("El  nuevo valor de a es  %lg\n", a);
22
23     return 0;
24 }
```

Punteros y vectores

Supongamos que un puntero "q" apunta a un entero; por tanto ha sido declarado como "int *q". El valor "q+10" mueve la dirección de la memoria cuatro elementos hacia la derecha, es decir, si un entero ocupa 4 bytes, el nuevo puntero "q+10" apunta a una posición de la memoria 40 bytes más allá del punto inicialmente apuntado por "q". Es importante darse cuenta de que los valores de las variables a las que apuntan los punteros se pueden expresar como vectores, así podemos escribir indistintamente "*(q+10)" o "q[10]".

Esta característica hace que los punteros estén muy relacionados con los vectores y las matrices, ya que en estos últimos, los valores se almacenan consecutivamente

en memoria, de modo que, sabiendo la posición en la memoria que ocupa la primera entrada, podemos desplazarnos a lo largo de toda la matriz o el vector. De hecho, si declaramos un vector como "`int v[5]`", tenemos que, en realidad, "`v`" es un puntero que apunta a "`v[0]`", es decir, "`v==&v[0]`" y "`*v==v[0]`". Por lo tanto, "`v+1`" contendrá la dirección de memoria en la que está almacenado "`v[1]`" y apuntará a él: "`v+1=&v[1]`" y "`*(v+1)=v[1]`", y así sucesivamente: "`v+n=&v[n]`" y "`*(v+n)=v[n]`". Nótese que la segunda expresión se obtiene de aplicar el operador "`*`" en ambos lados de la primera expresión:

$$*(v+n) \leftrightarrow *(\&v[n]) \leftrightarrow v[n]$$

Queda ya claro que podemos trabajar con vectores o punteros indistintamente. En el siguiente listado se muestra este hecho asignando valores a los elementos de un vector, previamente declarado, mediante un puntero.

Listado 11.9: Uso de punteros para asignar valores a los elementos de un vector.

```
1  #include <stdio.h>
2
3
4  int main(int argc, char** argv)
5  {
6      int v[6];
7      int *q;
8
9      /* asignamos al puntero q la dirección de memoria
10        del primer elemento del vector */
11     q=v;
12
13     /* también podríamos haber hecho q=&v[0] */
14
15     *v=0;          /* v[0]=0 */
16     v[1]=1;        /* v[1]=1 */
17     q[2]=2;        /* v[2]=2 */
18     *(q+3)=3;      /* v[3]=3 */
19     *(v+4)=4;      /* v[4]=4 */
20     *(&v[5])=5;    /* v[5]=5 */
21
22     printf ("El vector es {%d, %d, %d, %d, %d, %d}\n",
23             v[0], v[1], v[2], v[3], v[4], v[5]) ;
24
25     return 0;
26 }
```

Punteros y matrices

Una matriz no es más que un vector de vectores. Por lo tanto, si un vector es un puntero, una matriz será un puntero a puntero. Vamos a ver qué significa esto.

Una matriz declarada como "`int M[i][j]`" se almacena en memoria por filas consecutivas: `M[0][0], M[0][1], M[0][2], ..., M[0][j-1], M[1][0], M[1][1], ..., M[i-1,j-1]`. En el caso de las matrices, el nombre de la matriz "`M`" es un puntero a la dirección de memoria de la primera entrada de la matriz, esto quiere decir que su valor es la dirección de memoria en la que está almacenada la dirección de memoria de la primera entrada de la matriz, por eso se dice que es un puntero a puntero:

*M=&M[0][0] **M=M[0][0].

En realidad, "`M`" es un puntero al primer elemento de un vector de punteros llamado "`M[]`", cuyos elementos (que son punteros) contienen las direcciones de memoria del primer elemento de cada fila de la matriz. Esto quiere decir que

M==&M[0] M[0]==&M[0][0],

o lo que es lo mismo (aplicando el operador "$*$" en ambos lados de las igualdades)

*M==M[0] *M[0]==M[0][0],

que dicho verbalmente significa que lo que hay en la dirección "`M`" es "`M[0]`", que apunta a la primera fila de la matriz, y que es, a su vez, la dirección de memoria en la que se encuentra almacenado el elemento de su primera columna.

Por consiguiente, al igual que ocurría con los vectores, podremos desplazarnos a lo largo de matriz incrementando adecuadamente el valor de "`M`".

Por ejemplo, si incrementamos el puntero "`M`" en un una unidad, "`M+1`", nos iremos a la dirección de memoria en la que está contenido el segundo elemento del vector "`M[]`", "`M[1]`", el cual contiene a su vez la dirección de memoria en la que está almacenado el primer elemento de la segunda fila de la matriz. En general podemos escribir:

M+n==&M[n] M[n]==&M[n][0],

o

*(M+n)=M[n]=&M[n][0],

o

**(M+n)=*M[n]=M[n][0].

Por lo tanto, al elemento "`M[i][j]`" de la matriz podremos acceder de muchas formas, aquí tenemos algunos ejemplos:

M[i][j] = *(*(M+i)+j)

M[i][j] = *(M+i)[j]

M[i][j] = *(M[i]+j)

Es interesante darse cuenta de que, si aplicamos la regla general "`*(q+i)=q[i]`" en la primera de las expresiones anteriores, obtenemos las dos siguientes. En el listado siguiente se muestran diferentes modos de asignar valores a los elementos de una matriz.

Listado 11.10: Uso de la definición de puntero para asignar valores a los elementos de una matriz.

```
1  #include <stdio.h>
2
3
4  int main(int argc, char** argv)
5  {
6      int i,j,*q;
7      int M[3][3];
8
9
10     /* diferentes formas de definir los elementos de
11        la matriz M mediante punteros y vectores */
12
13     q=&M[1][0];
14
15     **M=1;          /* M[0][0]=1 */
16     *(*M+1)=2;      /* M[0][1]=2 */
17     (*M)[2]=3;      /* M[0][2]=3 */
18     **(M+1)=4;      /* M[1][0]=4 */
19     *(M[1]+1)=5;    /* M[1][1]=5 */
20     *(*(M+1)+2)=6; /* M[1][2]=6 */
21     *(M+2)[0]=7;    /* M[2][0]=7 */
22     *(q+4)=8;       /* M[2][1]=8 */
23     q[5]=9;         /* M[2][2]=9 */
24
25
26     for(i=0; i<3; i++)
27     {
28         for(j=0; j<3; j++)
29         {
30                 printf ("  %d", M[i][j]) ;
31         }
32          printf ("\n") ;
33     }
34
35     return 0;
36 }
```

Todo lo que acabamos de ver también se aplica, obviamente, a las cadenas de caracteres o *strings*. Una cadena de caracteres es un vector de tipo "`char`". Consideremos por ejemplo la siguiente declaración:

```
char cadena[]="hola mundo";
```

Según lo que acabamos de ver, "`cadena`" es un puntero al primer elemento de la cadena, de modo que por ejemplo `*(cadena+3)=='a'`. Para imprimir en pantalla la cadena sólo tendremos que escribir

printf ("El contenido de la cadena es: %s\n", cadena);

Ahora podemos definir un puntero a caracter "`char *a`" y realizar la asignación "`a=cadena`" u otra asignación como por ejemplo

a="adiós mundo";

También se puede declarar una cadena utilizando el tipo de variable "`char*`" del siguiente modo:

char* cadena="hola mundo";

Ahora podemos entender el significado del segundo argumento de la función main, declarado como "`char **argv`". En este caso "`argv`" es un puntero a puntero de caracteres, lo que nos permite definir una matriz de caracteres, o lo que es lo mismo, un vector de vectores de caracteres (esto es, un vector de cadenas). Por eso podemos utilizar la sintaxis equivalente "`char *argv[]`", que significa vector de punteros a caracter. En este caso "`argv[n]`" contendrá la dirección de memoria del primer caracter del argumento "`n`" pasado por línea de comandos (le apuntará). Otra forma de verlo es escribir "`char* argv[]`". Como el tipo de variable "`char*`" se refiere a una cadena, de este modo también estamos declarando un vector de cadenas.

11.9. Lectura y escritura de datos

En lo anterior toda la entrada de datos al programa venía dada, o bien, desde el sistema operativo (encargado de cargar los contenidos de la variable "`*argv[]`" de la función "`main`"), o bien, leyéndolos interactivamente con la función "`scanf`". Por otro lado, toda la salida de datos se ha realizado con "`printf`", que construía cadenas y las mostraba en el terminal.

Aunque no lo sepamos, hemos estado haciendo lectura y escritura de "corrientes de datos", que procederan de los archivos de datos (pero también podrán proceder de los "sockets" que conectan un programa a un dispositivo de adquisición de laboratorio, u otro programa a un servicio en otro ordenador o "sitio" de Internet).

Antes de abrir el primer archivo de datos, una prueba.

Ejercicio. La instrucción "`echo`" (en Unix) hace una cosa muy simple: escribe en el terminal todos los argumentos que se le pasan por línea de comandos (¿sería capaz de escribir un programa semejante con lo que sabe de C? si está usando Windows, debería hacerlo). Lo que escribe en el terminal, se puede dirigir a otro programa, usando el indicador "|" ("tubería"), como si éste lo estuviera leyendo desde la consola. Pruebe lo siguiente con el programa resultante de compilar 11.6, al que llamaremos "`scanmenu`":

echo 0 | ./scanmenu

Explique qué ha sucedido.

Un archivo en C se abre con la función "`fopen`" y retorna un puntero a un dato de tipo "`FILE`" (véase la documentación sobre esta función). Este dato contiene mucha información sobre el archivo: su nombre, su estado y posición de lectura/escritura, etc. Es algo habitual que, cuando lo que se necesita conocer es un conjunto de informaciones a las que el programador no va a acceder directamente, lo que se devuelva es un puntero con la dirección de memoria en la que empieza toda esa información (la información en sí puede variar, por ejemplo, de un sistema operativo a otro, o incluso entre compiladores, en el caso de Windows).

Como ya se ha dicho, la lectura o escritura de datos que se ha hecho hasta ahora era un caso particular de lectura y escritura de corrientes como la que describen los "`FILE`". En particular, se ha estado leyendo de "`stdin`" y escribiendo en "`stdout`", que figuran declarados en "`stdio.h`" como

```
extern FILE* stdin;
extern FILE* stdout;
```

En "`printf`" se sobreentiende que el destino es stdout. Para hacerlo explícito, o para escribir en un archivo abierto con "`fopen`", utilizaríamos la función "`fprintf`". Igualmente, para leer explícitamente de la entrada estándar "`stdin`", utilizaríamos la función "`fscanf`". Por si no es evidente, la "`f`" inicial en estas funciones se refiere a "`FILE`". Como ejemplo:

```
float x;
...
    fscanf(stdin, "%f", &x);
    fprintf (stdout, "%g", &x);
```

Un ejemplo con más contenido, se muestra en el siguiente listado 11.11. Este programa lee desde consola un número arbitrario de pares de números (X, Y), y los guarda en un archivo que se podrá abrir y representar con "Gnuplot".

Listado 11.11: El programa que guarda pares de coordenadas en un archivo de texto (que puede interpretar "Gnuplot").

```
1  #include <stdio.h>
2
3
4  int main(int argc, char** argv)
5  {
6          int npts, n;
7          double x, y;
8          FILE *fout;
9
10         fout=fopen("datos.dat", "w");
11
12         printf ("Número de puntos: ");
13         scanf("%d", &npts);
14
15         for(n=0; n<npts; n++) {
16                 printf ("X_%d= ", n);
17                 scanf("%lf", &x);
```

```
18                  printf ("Y_%d= ", n);
19                  scanf(" %lf",  &y);
20
21                  fprintf (fout ,  " %g\t %g\n", x,y);
22          }
23
24          fclose(fout);
25
26          return 0;
27  }
```

Ejercicio 11.12. Escriba un programa que lea desde un archivo de texto N pares de coordenadas (X, Y) y calcule la recta de regresión correspondiente, $Y = m_y X + y_0$, así como el coeficiente de correlación, r, de dicha recta. Para ello, empléense las fórmulas:

$$m_y = \frac{N\sum x_n y_n - \left(\sum x_n\right)\left(\sum y_n\right)}{N\sum x_n^2 - \left(\sum x_n\right)^2}$$

$$m_x = \frac{N\sum x_n y_n - \left(\sum x_n\right)\left(\sum y_n\right)}{N\sum y_n^2 - \left(\sum y_n\right)^2}$$

$$y_0 = \frac{1}{N}\left(\sum y_n - m_y \sum x_n\right)$$

$$r^2 = m_x m_y$$

11.10. Definición de funciones: paso de argumentos por valor y por referencia

En muchas ocasiones, deberemos calcular una función que no es posible expresar en forma de una expresión matemática cerrada. En ese caso nuestros cálculos involucrarán bucles que, como en el ejercicio anterior de regresión, proporcionarán finalmente uno o más valores.

Para calcular estas funciones, arbitrariamente complicadas, tendremos que declararlas y definirlas. Como se explicó en el primer tema, en C, cuando se llega a un punto en que se emplea una función, el compilador tiene que saber de qué forma es dicha función: debe haber sido declarada. Una función está declarada si se indica su tipo (el del valor que retorna) y los tipos de sus argumentos. Si a continuación de esto vienen unas llaves con la definición de la función (las instrucciones que debe seguir el ordenador para acabar dando el resultado), bien. Si no, también. En el proceso de enlazado, será cuando se busque la función por su nombre; como ya se explicó el "linker" no funciona dentro de la cadena continua de procesado en que se hallan el preprocesador y el compilador de C.

En el listado 11.12 se muestra un ejemplo de declaraciones de funciones que son llamadas desde funciones.

Una de ellas es "`sqr`" que se declara como "`static inline`". De los dos modificadores, sólo explicaremos el segundo; una función es `inline` cuando es tan pequeña

que el compilador la puede "sustituir" a la hora de generar el objeto. Esto hace que se salte el paso de *llamar a la función*, realizando sólo las operaciones que contiene ésta, ahorrando así tiempo de ejecución. Esta es una optimización del código máquina sobre la que podemos decidir.

La otra función hace un cálculo matemático sencillo: calcula el valor de una función llamada lorenciana dados sus parámetros (x_0, y_0, w) y su argumento (x). Con la instrucción "`return y`" se indica cuál es el valor que devuelve la función a partir de sus argumentos.[8]

La función "`main`" se encarga de abrir un archivo para escritura, pedir un intervalo de valores y los parámetros, y guardar los valores de la función en un archivo que puede representar "Gnuplot".

Listado 11.12: El programa calcula la función lorenciana en un intervalo y guarda los puntos en un archivo para representarlos con "Gnuplot".

```
1  #include <stdio.h>
2
3  static inline
4  double sqr(double x)
5  {
6          return x*x;
7  }
8
9  double lorenciana(double x, double x0,
10                   double y0, double w)
11 {
12         double y, den;
13
14         den=1+sqr(x-x0)/(w*w);
15
16         y=y0/den;
17
18         return y;
19 }
20
21
22 int main(int argc, char** argv)
23 {
24         double x, y, xa, xb, dx;
25         double x0, y0, w;
26         int npts;
27         FILE *fout;
28
```

[8] La función "`main`" retorna también un valor. Este valor indica al sistema operativo cuál es el error que ha causado la finalización del programa. Retornar un valor de 0, equivale a decir que no ha habido ningún error. Cualquier otro valor indica un error; cuál depende del gusto del programador, no hay ningún estándar. Por lo general, el intérprete de comandos del sistema operativo ("`bash`" en Linux, "`cmd.exe`" en Windows) no indicará nada acerca de este valor retornado por "`main`", pero otro programa sí podría hacerlo.

```
29          fout=fopen("funcion.dat", "w");
30
31          printf("Intervalo [a, b]: ");
32          printf(" a = ");
33          scanf("%lf", &xa);
34          printf(" b = ");
35          scanf("%lf", &xb);
36          printf("Número de puntos en el intervalo: ");
37          printf(" N = ");
38          scanf("%d", &npts);
39
40          printf("Parámetros de la lorenziana:"
41                  " Y(X) = Y0/(1+(X-X0)^2/W^2)\n");
42          printf(" Y0 = ");
43          scanf("%lf", &y0);
44          printf(" X0 = ");
45          scanf("%lf", &x0);
46          printf(" W = ");
47          scanf("%lf", &w);
48
49          dx=(xb-xa)/npts;
50
51          for(x=xa; x<=xb; x+=dx)
52          {
53                  y=lorenciana(x, x0, y0, w);
54
55                  fprintf(fout, "%g\t%g\n", x, y);
56          }
57
58          fclose(fout);
59
60          return 0;
61  }
```

Las funciones habituales en matemáticas (y en C) devuelven un único valor en respuesta a una serie de argumentos (y parámetros). Sin embargo, en C es posible que uno de estos argumentos sea una dirección de memoria, de una variable, en la que queremos que se "deposite" una información. Este es un truco muy usado en C para devolver muchos valores desde una única función.

Se dice que los argumentos de la función que son direcciones de memoria de variables son pasados *por referencia* (el puntero "apunta" o "referencia" a las variables) y que las otras, las que hemos usado hasta ahora, se pasan *por valor*. Ya hemos visto un ejemplo de variables pasadas por referencia en el caso de las funciones "`scanf`" y "`fscanf`": los argumentos "`&variable`" son pasos por referencia.

Dentro de la función, a los valores de estas variables pasadas por referencia se accede precediendo el nombre de la variable por un "`*`". Es fácil entender esto: hemos visto que los punteros (las referencias) se declaraban como "`int *ptr`". Esto quiere decir que "`*ptr`" es una variable de tipo "`int`", cuando "`ptr`" es el puntero a un valor

"int".

En el listado 11.13 mostramos un ejemplo de cómo podemos pasar variables, vectores y matrices, por referencia a una función. El programa calcula el producto de una matriz y un vector definidos en la función "main". Para ello los pasamos por referencia a la función "producto", que además calcula el módulo del vector resultante. Como todas estas variables son pasadas por referencia, la función "producto" no necesita devolver ningún valor (por eso se utiliza el tipo "void" y por eso al "return" de la función no se le asigna ningún valor). Obsérvese que el paso por referencia de las matrices exige la declaración de la dimensión de los vectores que las componen. Esto es lógico pues el puntero "M[i]" apunta al primer elemento de la fila "i", pero no "sabe" cuántos elementos hay en ella. Comprobar que la función "producto" podría haber sido declarada también del siguiente modo sin que el resto del programa mude:

void producto (double (*M)[3], double *V, double *MxV, double *modulo)

En este caso "(*M)[3]" representa un puntero a un vector de tamaño 3, esto es, un puntero a puntero, y no debe ser confundido con "*M[3]", que es un vector de punteros de tamaño 3. El operador subíndice o componente de un vector "[]" es el operador con mayor precedencia; se podría decir que se añade al nombre de la variable y se tratan variable y operador como un todo.

Listado 11.13: Programa que calcula el producto de una matriz por un vector definidos pasándolos por referencia a una función que realiza todas las operaciones.

```
1  #include <stdio.h>
2
3
4  void producto(double M[][3], double V[], double MxV[], double *modulo)
5  {
6      int i, j;
7
8      /* calculamos el producto de la matriz y el vector */
9      for (i=0; i<2; i++)
10         {
11             MxV[i]=0;
12             for (j=0; j<3; j++)
13                 MxV[i]+=M[i][j]*V[j];
14         }
15
16     /* calculamos el módulo del vector resultante */
17     for (i=0; i<2; i++)
18         *modulo+=MxV[i]*MxV[i];
19
20     *modulo=sqrt(*modulo);
21
22     return;
23 }
24
25
```

```
26 int main(int argc, char** argv)
27 {
28     double M   [2][3]={{1.,1.,1.},{1.,1.,1.}};
29     double V [3]={1.,2.,3.};
30     double MxV[2], modulo;
31
32     producto(M,V,MxV,&modulo);
33
34      printf ("El producto de la matriz M por el vector V"
35              " es (%g, %g)\n",
36               MxV[0], MxV[1] );
37      printf ("El modulo del vector es %f\n", modulo);
38
39     return 0;
40 }
```

Ejercicio 11.13. Modificar el programa del listado 11.13 para que la matriz sea pasada por referencia en forma de puntero simple, es decir, que la función "producto" esté declarada del siguiente modo:

void producto (**double** *p, **double** *V, **double** *MxV, **double** *modulo)

Ejercicio 11.14. Modificar el programa de cálculo de la recta de regresión definiendo (antes que "`main`") la función

```
double regresionLineal(int npts,
                       double x[], double y[],
                       double *_m, double *_y0)
{
     ...
    *_m=m_y;
    *_y0=y0;
    return r2;
}
```

a la que se pasan dos vectores "`x[]`" e "`y[]`" con las coordenadas de los "`npts`" puntos y las variables "`*_m`" y "`*_y0`" por referencia. La función calculará como lo hacía el programa anterior los valores de la pendiente m_y, la ordenada en el origen y_0 y el cuadrado del coeficiente de regresión r^2, y los retornará como se indica en el "esqueleto" anterior.

Vamos a concluir esta sección viendo que también podemos aplicar las ventajas de los punteros a las funciones. Al igual que ocurría con los vectores y las matrices, los nombres de las funciones son también punteros. Esto nos permite pasar por referencia funciones a otras funciones, es decir, pasar como argumento a una función el nombre de otra función. Para ello se utilizan declaraciones como:

int (*func)(double a, int b, . . .)

que representa un puntero (“`func`”) a una función que devuelve un entero y tiene como argumentos “`a`”, “`b`”, . . . Esta declaración no debe ser confundida con

int *func(double, int, . . .)

que declara una función “`func`” que devuelve un puntero a entero y tiene como argumentos “`a`”, ”`b`”, . . .

En el listado 11.14 se muestra un programa que toma dos números reales introducidos por línea de comandos y realiza entre ellos la operación (“multiplicar” o “dividir”) indicada en el tercer argumento introducido por línea de comandos. Como se puede comprobar, para comparar dos cadenas “`s1`” y “`s2`” se utiliza la función “`strcmp(s1,s2)`”, declarada en la librería “`string.h`”, que devuelve un entero: `0` si las dos cadenas son iguales y `1` si no lo son (por eso es necesario introducir el operador “`!`”). La operación deseada es pasada por referencia desde “`main`” a la función “`oper`”.

Listado 11.14: Programa que toma dos números por línea de comandos y ejecuta la operación indicada por el usuario (“multiplicar” o “dividir”).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5
6  double division(double a, double b)
7  {
8          return (a/b);
9  }
10
11 double multiplicacion(double a, double b)
12 {
13         return (a*b);
14 }
15
16 double oper(double a, double b, double (*operacion) (double, double))
17 {
18         return operacion(a,b);
19 }
20
21
22 int  main(int argc,  char** argv)
23 {
24     double a,b;
25     char *c;
26
27
28     a=atof(argv[1]) ;
29     b=atof(argv[2]) ;
30     c=argv[3];
31
```

```
32
33     if (!strcmp(c," multiplicar "))
34           printf ("el resultado de %s %g por %g es %g",
35                    c, a, b, oper(a,b,&multiplicacion));
36     else
37     if (!strcmp(c," dividir "))
38           printf ("el resultado de %s %g por %g es %g",
39                    c, a, b, oper(a,b,&division));
40     else
41          printf ("La operación no ha sido debidamente introducida");
42
43     return 0;
44 }
```

Ejercicio 11.15. Modificar el programa del listado 11.14 para que haga lo mismo, pero declarando la función "`oper`" dentro de "`main`" como

double (*oper)(double, double);

para luego asignarle la función correspondiente a la operación introducida por línea de comandos:

oper=multiplicacion;

si el usuario tecleó la palabra "multiplicar".

11.11. Utilidad: guardar datos como una imagen

Alguna vez querremos prescindir del "Gnuplot" para representar datos. Este programa es muy bueno, y puede hacer representaciones gráficas de toda clase, pero nosotros también podemos.

Ahora que hemos revisado las operaciones básicas en lenguaje C, vamos a sintetizar todo lo visto escribiendo un programa que guarda en un fichero una imagen en blanco y negro. El brillo de la imagen vendrá dado por una expresión matemática y puede ser un medio de representar funciones en forma de imágenes.

Una imagen digital es un conjunto de elementos de matriz llamados "píxeles" (*picture elements*, elementos de la imagen). En C una imagen es una matriz bidimensional. Si la imagen es en blanco y negro (en grises, para ser más exactos), los elementos de esa matriz bidimensional serán números que indican lo cerca que el píxel se encuentra del blanco o lo lejos que se encuentra del negro; es decir, guardarán números proporcionales al brillo de la imagen en ese píxel. En una imagen en color, cada píxel contendrá la información de brillo de cada uno de los colores (de luz) primarios: rojo, verde y azul. En esta sección sólo trataremos de imágenes en escala de grises, aunque es muy fácil pasar a imágenes RGB (de Red, Green, Blue, nombres de los colores primarios en Inglés).

Para que un programa de procesamiento de imágenes como el "Gimp"[9] sepa cómo leer una imagen, es necesario proporcionarle tres datos: el número de filas y de columnas de la matriz, y cuánto vale el *nivel de gris* en un píxel totalmente blanco (los píxeles negros se asume siempre que tienen valor de brillo cero, por razones obvias).

El formato de imagen que vamos a usar, el PGM (Portable Gray Map, mapa de grises portable), proporciona, precisamente esos tres datos: el ancho y alto de la imagen, y el valor del blanco en ella; a continuación, sigue toda la ristra de valores enteros que indican los niveles de gris de los píxeles, en el orden de lectura habitual de los elementos de una matriz (de arriba a abajo, de izquierda a derecha, o lo que es lo mismo, en orden creciente de filas, recorriendo todas las columnas en cada fila).

Todos los archivos que contienen imágenes comienzan con unos cuantos "*bytes mágicos*", valores que indican el formato en que se ha guardado la imagen en el archivo. En el caso del PGM, este "*magic*" es una línea de texto que dice "P2" (el formato 2 de la familia PNM; véanse las páginas de manual, "`man pgm`" y "`man pnm`" para más detalles). Así, un ejemplo de archivo de imagen PGM es el icono  , que se muestra a continuación:

```
P2
24 7
15
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Los valores 24 y 7 son las columnas y filas de la imagen, respectivamente; en la imagen, el negro vale 0, y el blanco vale 15.

El siguiente ejemplo (listado 11.15) genera una imagen de 512×512 píxeles que representa el cuadrado $[-1;+1] \times [-1;+1]$. Su brillo viene dado por la función

$$f(x,y) = \frac{1}{1+2x^2+3y^2}$$

donde x e y varían en dicho cuadrado $[-1;+1] \times [-1;+1]$.

Como novedades de este programa 11.15, destacamos:

1. El tamaño de la imagen se ha definido usando una directiva del preprocesador, "`#define`" que lo que hace es que, cuando se ejecuta el preprocesador, las palabras "`ALTURA`" y "`ANCHURA`" sean sustituidas por el número `512`. "`ALTURA`" y "`ANCHURA`" no son variables del programa; no se pueden asignar, simplemente porque cuando llegan al compilador ya no son literales, sino el número `512`, como se han "`#definido`".

[9] Hablaremos del "Gimp" (el GNU Image Manipulation Program, o programa de manipulación de imágenes de GNU) porque es el programa estándar de retoque fotográfico en Linux y porque también está disponible en Windows. Otros programas válidos para nuestro propósito (es decir, capaces de abrir imágenes en formato PGM) son "ImageMagick" (en Linux), "PaintShop Pro" (distribuido como "shareware") y el "Adobe PhotoShop" (comercial). Este formato no está soportado por los programas del Office de Microsoft.

2. El tipo de la función "`guardaPGMd`", es "`void`". Este tipo denota la falta de tipo y valor de retorno: basta ver que al "`return`" de la función no le sigue ninguna expresión. Usaremos el tipo "`void`" en C cuando algo no tenga tipo. Una variable nunca será "`void`", porque no tiene sentido guardar algo que es nada (o está indefinido lo que es). Sin embargo sí será habitual tener punteros a "`void`", declarados como "`void *ptr`": apuntan a una dirección de memoria, en la que no se sabe qué es lo que hay.

3. La matriz de píxels en la función "`main`", llega a la función "`guardaPGMd`" como un vector. Esto se debe a que los elementos de una matriz se guardan en memoria uno tras otro, precisamente en el orden en que los leeríamos (y en el que los va a guardar "`guardaPGMd`"). Al llamar a la función, esta conversión de una matriz "`double f[ALTURA][ANCHURA]`" a un vector "`double pixels[]`" o, equivalentemente, "`double *pixels`", se indica con un *retipado* (un "cast", en Inglés): "`(double*)f`". Esta expresión, un tipo de dato entre paréntesis precediendo a una variable, quiere decir: sea del tipo que sea la variable, trátala como si fuera de este tipo. Siempre que sea posible, el compilador C tomará las medidas oportunas y la tratará así. Otro ejemplo en este programa son las conversiones de las variables enteras "`i`" y "`j`" a tipo "`double`" antes de ser divididas por la "`ALTURA`" y la "`ANCHURA`", respectivamente (¿qué pasaría si no se hiciese este retipado?).

Listado 11.15: El programa que guarda una imagen en escala de grises (formato PGM) calculada a partir de una expresión matemática.

```
1  #include <stdio.h>
2
3  #define ANCHURA 512
4  #define ALTURA 512
5
6  void guardaPGMd(char* nombre,
7                  int anchura, int altura,
8                  double *pixels,
9                  double pixel_min, double pixel_max)
10 {
11         int i, j, ij, p;
12         FILE* imagen;
13
14         imagen=fopen(nombre, "wb");
15         fprintf (imagen, "P2");
16         fprintf (imagen, "#guardaPGMd %s\n", nombre);
17         fprintf (imagen, "%d %d\n", anchura, altura);
18         fprintf (imagen, "255\n");
19
20         ij=0;
21         for(i=0; i<altura; i++)
22         {
23                 for(j=0; j<anchura; j++)
24                 {
25                         p=(int)(255*(pixels[ij]-pixel_min))
```

```
26                                      /(pixel_max-pixel_min);
27                              if ( p<0 ) p=0;
28                              if ( p>255) p=255;
29
30                               fprintf (imagen, " %d", p);
31
32                              ij ++;
33                      }
34                       fprintf (imagen, "\n");
35              }
36
37              fclose(imagen);
38
39              return;
40      }
41
42      int main(int argc, char **argv)
43      {
44              int i, j;
45              double x, y, z;
46              double f[ALTURA][ANCHURA];
47
48              for(i=0; i<ALTURA; i++)
49              {
50                      for(j=0; j<ANCHURA; j++)
51                      {
52                              x=2*(double)j/(ANCHURA-1)-1.0;
53                              y=2*(double)i/(ALTURA-1)-1.0;
54                              z=1.0/(1+(2*x*x+3*y*y));
55
56                              f[i][j]=z;
57                      }
58              }
59
60              guardaPGMd("prueba.pgm",
61                              ANCHURA, ALTURA, (double*)f, 0, 1.0);
62
63              return 0;
64      }
```

11.12. Estructuras de datos y definición de tipos

Una estructura de datos es un conjunto de datos que el programa debe tratar siempre juntos y que, por ello, es conveniente agrupar o encapsular en una única

Figura 11.3: Imagen de ejemplo generada por el programa del listado 11.15.

entidad, una estructura[10]. En C, esta estructura de datos se declara usando "`struct`". Por ejemplo, la siguiente estructura define un vector bidimensional, formado por dos componentes reales: "`x`", "`y`".

```
struct Vector2d {
    double x, y;
};
```

En un programa, se puede operar entre dos vectores explícitamente como:

```
struct Vector2d u, v;
 ...
u.x=1.0; u.y=2.0;
v.x=2*u.x; v.y=2*u.y;
 ...
```

o creando funciones que retornen el resultado, exactamente igual a como si la operación se realizase entre dos tipos de datos usuales (primitivos):

```
struct Vector2d porescalar(struct Vector2d u, double l)
{
    struct Vector2d v;
    v.x=l*u.x;
    v.y=l*u.y;
    return v;
}
```

[10] En otros lenguajes a las estructuras se las llama "record", porque cuando se lee o guarda el conjunto de datos se hace de forma agrupada (atómica). En algún contexto se llamará a una estructura de datos un objeto, aunque este concepto es más genérico. También se emplea el término "estructura de datos" para indicar un modo de jerarquizar los datos, dotándolos de relaciones entre ellos (formando listas, árboles, grafos, etc.). Aquí nos quedaremos en el primer concepto, más sencillo, aunque en la base de todos los demás.

Esta notación obliga a utilizar siempre la palabra clave "`struct`" a la hora de declarar el uso de una estructura de tipo "`Vector2d`". Cuando el uso es muy frecuente, puede ser interesante instruir al compilador para que trate a las estructuras como otro tipo de datos más (ya que así lo estamos haciendo ya, por otro lado). Para ello utilizaremos la instrucción "`typedef`".

Por ejemplo, para trabajar con números reales, nos puede interesar definir el tipo "`Real`" como:

```
typedef double Real;
```

Del mismo modo, para definir el tipo "`Vector2d`", haríamos los dos pasos siguientes[11]:

```
struct _Vector2d
{
    double x, y;
};
typedef struct _Vector2d Vector2d;
```

Obsérvese que, en lo que sigue a "`typedef`" es como si estuviéramos declarando una variable llamada "`Real`", en el primer ejemplo, o "`Vector2d`", en el segundo.

Ejemplo. Escribir una función que, dado un vector del plano (variable de tipo "`Vector2d`"), lo rote en un ángulo dado. Para ello, se pasará la variable por referencia a una función "`void rotar(Vector2d, double)`", donde el segundo argumento es el ángulo en radianes: esto se muestra en el listado 11.16. Obsérvese en esta función la notación para acceder a los campos de una variable pasada a través de un puntero (por ejemplo, "`v->x`"). Explíquese el uso de la variable auxiliar "`double aux`" empleada en la función "`void rotar()`".

Listado 11.16: El programa que rota un vector en un ángulo de $+10°$.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  struct _Vector2d
5  {
6          double x, y;
7  };
8
9  typedef struct _Vector2d Vector2d;
10
11 void rotar(Vector2d *u, double theta)
12 {
13         double aux;
```

[11] También se podrían usar...

la sintaxis abreviada:

```
typedef struct _Vector2d
{
    double x, y;
} Vector2d;
```

la sintaxis compacta (con una estructura anónima):

```
typedef struct
{
    double x, y;
} Vector2d;
```

```
14
15          aux=u->x;
16          u->x=u->x*cos(theta)-u->y*sin(theta);
17          u->y=aux*sin(theta)+u->y*cos(theta);
18
19          return;
20  }
21
22  #define ROTACION (M_PI*10/180)
23
24  int  main(int argc,  char** argv)
25  {
26          int  n;
27          Vector2d vec;
28          vec.x=1.0;
29          vec.y=0.0;
30
31          for(n=1; n<=9; n++)
32          {
33                  rotar(&vec, ROTACION);
34                  printf (" %d grados: (%g,  %g)\n", 10*n, vec.x, vec.y);
35          }
36
37          return 0;
38  }
```

Notación.

Es muy conveniente adherirse a un estilo de escritura de los programas, porque ayuda mucho a comprender el propósito y el funcionamiento de los mismos.

Por ejemplo, en todo esta unidad didáctica nos adheriremos al convenio de abrir las llaves en la línea siguiente a la instrucción que las rige, y a cerrarlas en una línea después de la última instrucción. Esta forma de estructurar el texto le es indiferente al preprocesador o al compilador, pero a nosotros nos facilita la lectura.

Otra norma a la que conviene adherirse es a nombrar las variables o funciones con la primera letra minúscula (por ejemplo "`double coordX;`", o "`extern double sin(double x);`") o a las `#defini`iciones de símbolos con todas las letras mayúsculas (por ejemplo "`#define M_PI 3.14159265354`").

Nosotros usaremos una norma más referida a los nombres de las estructuras y nuevos tipos que definamos: comenzarán por una letra mayúscula. De este modo, cuando veamos "`Vector2d`", sabremos que es una estructura o nuevo tipo definido por nosotros (no un tipo primitivo del lenguaje), y no una función o variable. En los ejemplos anteriores hemos visto ya un contraejemplo de esta norma autoimpuesta: "`_Vector2d`", que empieza por "`_`".