CAPÍTULO 11

ALGUNOS USOS CON FUNCIONES

En un capítulo anterior hemos visto lo básico sobre funciones. Con todo lo dicho en ese tema se puede trabajar perfectamente en C, e implementar multitud de programas, con buena modularidad.

En este tema queremos presentar muy brevemente algunos usos más avanzados de las funciones: distintas maneras en que pueden ser invocadas. Punteros a funciones, vectores de punteros a funciones, el modo de pasar una función como argumento de otra función. Son modos de hacer sencillos, que añaden, a todo lo dicho en el tema anterior, posibilidades de diseño de programas.

Otra cuestión que abordaremos en este tema es cómo definir aquellas funciones de las que desconozcamos *a priori* el número de parámetros que han de recibir. De hecho, nosotros ya conocemos algunas de esas funciones: la función *printf* puede ser invocada con un solo parámetro (la cadena de caracteres que no imprima ningún valor) o con tantos como se quiera: tantos como valores queramos que se impriman en

nuestra cadena de caracteres. Veremos también aquí la manera de definir funciones con estas características.

Punteros a funciones

En los primeros temas de este manual hablábamos de que toda la información de un ordenador se guarda en memoria. No sólo los datos. También las instrucciones tienen su espacio de memoria donde se almacenan y pueden ser leídas. Todo programa debe ser cargado sobre la memoria principal del ordenador antes de comenzar su ejecución.

Y si una función cualquiera tiene una ubicación en la memoria, entonces podemos hablar de la dirección de memoria de esa función. Desde luego, una función ocupará más de un byte, pero se puede tomar como dirección de memoria de una función aquella donde se encuentra la entrada de esa función.

Y si tengo definida la dirección de una función… ¿No podré definir un puntero que almacene esa dirección? La respuesta es que sí, y ahora veremos cómo poder hacerlo. Por tanto, podremos usar un puntero para ejecutar una función. Ese puntero será el que también nos ha de permitir poder pasar una función como argumento de otra función.

La declaración de un puntero a una función es la declaración de una variable. Ésta puede ser local, y de hecho, como siempre, será lo habitual. Cuando se declara un puntero a función para poder asignarle posteriormente la dirección de una o u otra función, la declaración de ese puntero a función debe tener un prototipo coincidente con las funciones a las que se desea apuntar.

Supongamos que tenemos las siguientes funciones declaradas al inicio de un programa:

tipo_función nombre_función_1 (tipo1, …, tipoN);

tipo_función nombre_función_2 (tipo1, …, tipoN);

Y supongamos ahora que queremos declarar, por ejemplo en la función principal, un puntero que pueda recoger la dirección de estas dos funciones. La declaración del puntero será la siguiente:

tipo_función (*puntero_a_funcion)(tipo1,…,tipoN);

De esta declaración podemos hacer las siguientes importantes observaciones:

1. *tipo_función* debe coincidir con el tipo de la función a la que va a apuntar el puntero a función. De la misma manera la lista de argumentos debe ser coincidente, tanto en los tipos de dato que intervienen como en el orden. En definitiva, los prototipos de la función y de puntero deber ser idénticos.
2. Si **puntero_a_función* NO viniese recogido entre paréntesis entonces no estaríamos declarando un puntero a función, sino una función normal que devuelve un tipo de dato puntero: un puntero para una recoger la dirección de una variable de tipo *tipo_función*. Por eso los paréntesis no son opcionales.

Una vez tenemos declarado el puntero, el siguiente paso será siempre asignarle una dirección de memoria. En ese caso, la dirección de una función. La sintaxis para esta asignación es la siguiente:

puntero_a_función = nombre_función_1;

Donde *nombre_función_1* puede ser el nombre de cualquier función cuyo prototipo coincide con el del puntero.

Una observación importante: al hacer la asignación de la dirección de la función, hacemos uso del identificador de la función: no se emplea el operador &; tampoco se ponen los paréntesis al final del identificador de la función.

Al ejecutar *puntero_a_funcion* obtendremos un comportamiento idéntico al que tendríamos si ejecutáramos directamente la función. La sintaxis para invocar a la función desde el puntero es la siguiente:

resultado = (*puntero_a_función)(var_1, …,var_N);

Y así, cuando en la función principal se escriba esta sentencia tendremos el mismo resultado que si se hubiera consignado la sentencia

resultado = nombre_a_función_1(var_1, …,var_N);

Antes de ver algunos ejemplos, hacemos una última observación. El puntero función es una variable local en una función. Mientras estemos en el ámbito de esa función podremos hacer uso de ese puntero. Desde luego toda función trasciende el ámbito de cualquier otra función; pero no ocurre así con los punteros.

Veamos algún ejemplo. Hacemos un programa que solicita al usuario dos operandos y luego si desea sumarlos, restarlos, multiplicarlos o dividirlos. Entonces muestra el resultado de la operación. Se definen cuatro funciones, para cada una de las cuatro posibles operaciones a realizar. Y un puntero a función al que se le asignará la dirección de la función que ha de realizar esa operación seleccionada.

El código podría quedar como sigue:

```
#include <stdio.h>

float sum(float, float);
float res(float, float);
float pro(float, float);
float div(float, float);

void main(void)
{
      float a, b;
      unsigned char op;
      float (*operacion)(float, float);

      printf("Primer operador ... ");
      scanf("%f",&a);
      printf("Segundo operador ... ");
      scanf("%f",&b);
      printf("Operación ( + , - , * , / ) ... ");
      do
            op = getchar();
      while(op !='+' && op !='-' &&  op !='*' &&  op !='/');

      switch(op)
```

```
      {
      case '+':   operacion = sum;  break;
      case '-':   operacion = res;  break;
      case '*':   operacion = pro;  break;
      case '/':   operacion = div;
      }

      printf("\n%f %c %f = %f",a, op, b, (*operacion)(a, b));
}

float sum(float x, float y)
{ return x + y; }

float res(float x, float y)
{ return x - y; }

float pro(float x, float y)
{ return x * y; }

float div(float x, float y)
{ return y ? x / y : 0; }
```

La definición de las cuatro funciones no requiere a estas alturas explicación alguna. El puntero *operación* queda definido como variable local dentro de ***main***. Dependiendo del valor de la variable *op* al puntero se le asignará la dirección de una de las cuatro funciones, todas ellos con idéntico prototipo, igual a su vez al prototipo del puntero.

Evidentemente, esto es sólo un ejemplo. Hay otras muchas formas de resolver el problema, y quizá alguno piense que es más complicado el uso del puntero, y que podría hacerse recogido en cada ***case*** de la estructura ***switch*** la llamada a la función correspondiente. Y no le faltará razón. Ya hemos dicho muchas veces que aquí tan hay tantas soluciones válidas como programadores. Pero desde luego las posibilidades de implementación que ofrece el puntero a función son claras.

Vectores de punteros a funciones

No aportamos aquí ningún concepto nuevo, sino una reflexión sobre otra posibilidad que ofrece el tener punteros a funciones.

Como todo puntero, un puntero a función puede formar parte de un array. Y como podemos definir arrays de todos los tipos que queramos, entonces podemos definir un array de tipo de dato punteros a funciones. Todos ellos serán del mismo tipo, y por tanto del mismo prototipo de función. La sintaxis de definición será la siguiente:

tipo_función (*puntero_a_función[dimensión])(tipo1, … tipoN);

Y la asignación puede hacerse directamente en la creación del puntero, o en cualquier otro momento:

tipo_función (*puntero_a_función[n])(tipo1, … tipoN) =
{ funcion_1, función_2, …, función_n }

Donde deberá haber tantos nombres de función, todas ellas del mismo tipo, como indique la dimensión del vector. Como siempre, cada una de las funciones deberá quedar declarada y definida en el programa.

El vector de funciones se emplea de forma análoga a cualquier otro vector. Se puede acceder a cada una de esas funciones mediante índices, o por operatoria de punteros.

Podemos continuar con el ejemplo del epígrafe anterior. Supongamos que la declaración del puntero queda transformada en la declaración de una array de dimensión 4:

```
float(*operacion[4])(float,float)= {sum,res,pro,div};
```

Con esto hemos declarado cuatro punteros, cada uno de ellos apuntando a cada una de las cuatro funciones definidas. A partir de ahora será lo mismo invocar a la función *sumaf* que invocar a la función apuntada por el primer puntero del vector.

Si incorporamos en la función ***main*** la declaración de una variable *i* de tipo entero, la estructura ***switch*** puede quedar ahora como sigue

```
switch(op)
{
case '+':      i = 0;      break;
case '-':      i = 1;      break;
case '*':      i = 2;      break;
case '/':      i = 3;
```

```
}
```

Y ahora la ejecución de la función será como sigue:

```
printf("\n\n%f %c %f = %f", a, op, b, (*operación[i])(a, b));
```

Funciones como argumentos

Se trata ahora de ver cómo hemos de definir un prototipo de función para que pueda recibir a otras funciones como parámetros. Un programa que usa funciones como argumentos suele ser difícil de comprender y de depurar, pero se adquiere a cambio una gran potencia en las posibilidades de C.

La utilidad de pasar funciones como parámetro en la llamada a otra función está en que se puede hacer depender cuál sea la función a ejecutar del estado a que se haya llegado a lo largo de la ejecución. Estado que no puede prever el programador, porque dependerá de cada ejecución concreta. Y así, una función que recibe como parámetro la dirección de una función, tendrá un comportamiento u otro según reciba la dirección de una u otra de las funciones declaradas y definidas.

La sintaxis del prototipo de una función que recibe como parámetro la dirección de otra función es la habitual: primero el tipo de la función, seguido de su nombre y luego, entre paréntesis, la lista de parámetros. Y entre esos parámetros uno o algunos pueden ser punteros a funciones. La forma en que se indica ese parámetro en la lista de parámetros es la siguiente:

tipo_función (*puntero_a_funcion)(parámetros)

(Lo que queda aquí recogido no es el prototipo de la función, sino el modo en que se consigna el parámetro puntero a función dentro de una lista de parámetros en un prototipo de función que recibe, entre sus argumentos, la dirección de una función.)

Supongamos que este parámetro pertenece al prototipo de la función *nombre_función*. Entonces cuando se compile *nombre_función* el compilador sólo sabrá que esta función recibirá como argumento, entre otras cosas, la dirección de una función que se ajusta al prototipo declarado como parámetro. Cuál sea esa función es cuestión que no se conocerá hasta el momento de la ejecución y de la invocación a esa función.

La forma en que se llamará a la función será la lógica de acuerdo con estos parámetros. El nombre de la función y seguidamente, entre paréntesis, todos sus parámetros en el orden correcto. En el momento de recoger el argumento de la dirección de la función se hará de la siguiente forma:

puntero_a_función(parámetros)

De nuevo será conveniente seguir con el ejemplo anterior, utilizando ahora una quinta función para realizar la operación y mostrar por pantalla su resultado:

```
#include <stdio.h>
#include <conio.h>

float sum(float, float);
float res(float, float);
float pro(float, float);
float div(float, float);
void mostrar(float, char, float, float(*f)(float, float));

void main(void)
{
      float a, b;
      unsigned char op;
      float (*operacion[4])(float, float) ={sum,res,pro,div};
      do
      {
            printf("\n\nPrimer operador ... ");
            scanf("%f",&a);
            printf("Segundo operador ... ");
            scanf("%f",&b);
            printf("Operación ... \n)");
            printf("\n\n1. Suma\n2. Resta”);
            printf(“\n3. Producto\n4. Cociente");
            printf("\n\n\tSu opción (1 , 2 , 3 , 4) ... ");
```

```
            do
                  op = getche();
            while(op - '0' < 1 || op - '0' > 4 );
            mostrar(a,op,b,operacion[(short)(op - '1')]);
            printf("\n\nOtra operación (s / n) ... ");
            do
                  op = getche();
            while(op != 's' && op != 'n');
      }while(op == 's');
}

float sum(float x, float y)
{ return x + y; }

float res(float x, float y)
{ return x - y; }

float pro(float x, float y)
{ return x * y; }

float div(float x, float y)
{ return y ? x / y : 0; }

void mostrar(float x,char c,float y,float(*f)(float,float))
{
      if(c == '1') c = '+';
      else if(c == '2') c = '-';
      else if(c == '3') c = '*';
      else c = '/';

      printf("\n\n%f %c %f = ", x, c, y);
      printf("%f.", (*f)(x,y));
}
```

Vamos viendo poco a poco el código. Primero aparecen las declaraciones de cinco funciones: las encargadas de realizar suma, resta, producto y cociente de dos valores **float**. Y luego, una quinta función, que hemos llamado *mostrar*, que tiene como cuarto parámetro un puntero a función. La declaración de este parámetro es como se dijo: el tipo del puntero de función, el nombre del puntero, recogido entre paréntesis y precedido de un asterisco, y luego, también entre paréntesis, la lista de parámetros del puntero a función. Así ha quedado declarada.

Y luego comienza la función principal, ***main***, donde viene declarado un vector de cuatro punteros a función. A cada uno de ellos le hemos asignado una de las cuatro funciones.

Y hemos recogido el código de toda la función ***main*** en un bloque ***do–while***, para que se realicen tantas operaciones como se deseen. Cada vez que se indique una operación se hará una llamada a la función mostrar que recibirá como parámetro una de las cuatro direcciones de memoria de las otras cuatro funciones. La llamada es de la forma:

```
mostrar(a,op,b,operacion[(short)(op - '1')]);
```

Donde el cuarto parámetro es la dirección de la operación correspondiente. *operacion[0]* es la función *sum*; *operacion[1]* es la función *res*; *operacion[2]* es la función *pro*; y *operacion[3]* es la función *div*. El valor de *op – 1* será 0 si *op* es el carácter '1'; será 1 si es el carácter '2'; será 2 si es el carácter '3'; y será 3 si es el carácter '4'.

Y ya estamos en la función mostrar, que simplemente tiene que ejecutar el puntero a función y mostrar el resultado por pantalla.

Ejemplo: la función ***qsort***

Hay ejemplos de uso de funciones pasadas como parámetros muy utilizados, como por ejemplo la función *qsort*, de la biblioteca **stdlib.h**. Esta función es muy eficaz en la ordenación de grandes cantidades de valores. Su prototipo es:

void qsort(void *base, size_t nelem, size_t width, int (*fcmp)(const void*, const void*));

Es una función que no devuelve valor alguno. Recibe como parámetros el puntero *base* que es quien recoge la dirección del array donde están los elementos a ordenar; *nelem*, que es un valor entero que indica la dimensión del vector pasado como primer parámetro; *width* es el tercer parámetro, que indica el tamaño que tiene cada uno de los elementos del array; y por fin, el cuarto parámetro, es una función que devuelve un valor entero y que recibe como parámetros dos direcciones de dos variables. La función que se pase como parámetro en este puntero debe devolver un 1 si su primer parámetro apunta a un valor mayor que el

segundo parámetro; el valor -1 si es al contrario; el valor 0 si el valor de ambos parámetros son iguales.

Hay que explicar porqué los tipos que recoge el prototipo son siempre ***void***. El motivo es porque la función *qsort* está definida para ser capaz de ordenar un array de cualquier tipo. Puede ordenar enteros, reales, letras, u otros tipos de dato mucho más complejos, que se pueden crear y que veremos en un capítulo posterior. La función no tiene en cuenta el tipo de dato: simplemente quiere saber dos cosas:

1. El tamaño del tipo de dato; y eso se le facilita a la función a través del tercer parámetro, *width*.
2. Cómo se define la ordenación: como *a priori* no se sabe el tipo de dato, tampoco puede saber la función *qsort* con qué criterio decidir qué valores del dominio del tipo de dato son mayores, o iguales, o menores. Por eso, la función *qsort* requiere que el usuario le facilite, mediante una función muy simple que debe implementar cada usuario de la función *qsort*, ese criterio de ordenación.

Actualmente el algoritmo que da soporte a la función *qsort* es el más eficaz en las técnicas de ordenación de grandes cantidades de valores.

Vamos a ver un ejemplo de uso de esta función. Vamos a hacer un programa que ordene un vector bastante extenso de valores enteros que asignaremos de forma aleatoria. Para ello deberemos emplear también alguna función de generación de aleatorios. Pero esa es cuestión muy sencilla que aclaramos antes de mostrar el código de la función que hemos sugerido.

Existe una función en **stdlib.h** llamada ***random***. Esa función pretende ser un generador de números aleatorios. Es un generador bastante malo, pero para nuestros propósitos sirve. Su prototipo es:

int random(int num);

Es una función que devuelve un entero aleatorio entre 0 y $(num - 1)$. El valor de *num* que se le pasa a la función como parámetro también debe ser un valor entero.

Cuando en una función se hace uso de la función *random*, antes debe ejecutarse otra función previa: la función ***randomize***. Esta función inicializa el generador de aleatorios con un valor inicial también aleatorio. Su prototipo es:

void randomize(void);

Y se ejecuta en cualquier momento del programa, pero siempre antes de la primera vez que se ejecute la función *random*.

Una vez presentadas las funciones necesarias para general aleatorios, veamos como queda un posible programa que genera una serie de enteros aleatorios y que los ordena de menor a mayor, haciendo uso de la función *qsort*:

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

#define TAM 10
#define RANGO 1000
int ordenar(void*,void*);

void main(void)
{
      long numeros[TAM];
      long i;
      randomize();
      for(i = 0 ; i < TAM ; i++)
            numeros[i] = random(RANGO);

// Vamos a ordenar esos numeros ...

      qsort((void*)numeros, TAM, sizeof(long), ordenar);

// Mostramos resultados

      for(i = 0 ; i < TAM ; i++)
            printf("numeros[%4ld] = %ld\n", i, numeros[i]);
}
```

```
// La función de ordenación ...
int ordenar(void *a, void *b)
{
      if(*(long*)a > *(long*)b)
            return 1;
      else if(*(long*)a < *(long*)b)
            return -1;
      return 0;
}
```

Hemos definido la función *ordenar* con un prototipo idéntico al exigido por la función *qsort*. Recibe dos direcciones de memoria (nosotros queremos que sea de enteros largos, pero eso no se le puede decir a *qsort*) y resuelve cómo discernir la relación mayor que, menor que, o identidad entre dos cualesquiera de esos valores que la función recibirá como parámetros.

La función trata a las dos direcciones de memoria como de tipo de dato ***void***. El puntero a ***void*** ni siquiera sabe qué cantidad de bytes ocupa la variable a la que apunta. Toma la dirección del byte primero de nuestra variable, y no hace más. Dentro de la función, el código ya especifica, mediante el operador forzar tipo, que la variable apuntada por esos punteros será tratada como una variable ***long***. Es dentro de nuestra función donde especificamos el tipo de dato de los elementos que vamos a ordenar. Pero la función *qsort* van a poder usarla todos aquellos que tengan algo que ordenar, independientemente de qué sea ese “algo”: porque todo el que haga uso de *qsort* le explicará a esa función, gracias al puntero a funciones que recibe como parámetro, el modo en que se decide quien va antes y quien va después. Lo que aporta *qsort* es la rapidez en poner en orden una cantidad ingente de valores del mismo tipo.

Y, efectivamente, hay muchas formas de resolver los problemas y de implementarlos. Y el uso de punteros a funciones, o la posibilidad de pasar como parámetro de una función la dirección de memoria de otra función es una posibilidad que ofrece enormes ventajas y posibilidades.

Estudio de tiempos

A veces es muy ilustrativo poder estudiar la velocidad de algunas aplicaciones que hayamos implementado en C.

En algunos programas de ejemplo de capítulos anteriores habíamos presentado un programa que ordenaba cadenas de enteros. Aquel programa, que ahora mostraremos de nuevo, estaba basado en un método de ordenación llamado método de la burbuja: consiste en ir pasando para arriba aquellos enteros menores, de forma que van quedando cada vez más abajo, o más atrás (según se quiera) los enteros mayores. Por eso se llama el método de la burbuja: porque lo liviano “sube”.

Vamos a introducir una función que controla el tiempo de ejecución. Hay funciones bastante diversas para este estudio. Nosotros nos vamos ahora a centrar en una función, disponible en la biblioteca **time.h**, llamada ***clock***, cuyo prototipo es:

clock_t clock(void);

Vamos a considerar por ahora que el tipo de dato ***clock_t*** es equivalente a tipo de dato ***long*** (de hecho así es). Esta función está recomendada para medir intervalos de tiempo. El valor que devuelve es proporcional al tiempo trascurrido desde el inicio de ejecución del programa en la que se encuentra esa función. Ese valor devuelto será mayor cuanto más tarde se ejecute esta función *clock*, que no realiza tarea alguna más que devolver el valor actualizado del contador de tiempo. Cada breve intervalo de tiempo (bastantes veces por segundo: no vamos ahora a explicar este aspecto de la función) ese contador que indica el intervalo de tiempo transcurrido desde el inicio de la ejecución del programa, se incrementa en uno.

Un modo de estudiar el tiempo trascurrido en un proceso será el siguiente:

```
time_t t1, t2;
t1 = clock();
```

```
(proceso a estudiar su tiempo)
t2 = clock();
printf("Intervalo transcurrido: %ld", t2 – t1);
```

El valor que imprimirá este código será proporcional al tiempo invertido en la ejecución del proceso del que estudiamos su ejecución. Si esa ejecución es muy rápida posiblemente el resultado sea cero.

Veamos ahora dos programas de ordenación. El primero mediante la técnica de la burbuja. Como se ve, en esta ocasión trabajamos con un vector de cien mil valores para ordenar:

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

#define TAM 100000
#define RANGO 10000

int cambiar(long*, long*);

void main(void)
{
      long numeros[TAM];
      long i, j;
      time_t t1, t2;
      randomize();
      for(i = 0 ; i < TAM ; i++)
            numeros[i] = random(RANGO);
// Vamos a ordenar esos numeros ...
// Método de la burbuja ...
      t1 = clock();
      for( i = 0 ; i < TAM ; i++)
            for(j = i ; j < TAM ; j++)
            if(numeros[i] > numeros[j])
                  cambiar(numeros + i, numeros + j);
      t2 = clock();
      printf("t2 - t1 = %ld.\n", t2 - t1);
}

int cambiar(long *a, long *b)
{
      *a ^= *b;
      *b ^= *a;
      *a ^= *b;
}
```

Si lo ejecuta en su ordenador, le aparecerá por pantalla (quizá tarde unos segundos: depende de lo rápido que sea su ordenador) una número. Si es cero, porque la ordenación haya resultado muy rápida, simplemente aumente el valor de TAM y vuelva a compilar y ejecutar el programa.

Ahora escriba este otro programa, que ordena mediante la función *qsort*. Es el que hemos visto antes, algo modificado para hacer la comparación de tiempos:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define TAM 100000
#define RANGO 10000
int ordenar(void*,void*);

void main(void)
{
      long numeros[TAM];
      long i, j;
      time_t t1, t2;
      randomize();
      for(i = 0 ; i < TAM ; i++)
            numeros[i] = random(RANGO);

// Vamos a ordenar esos numeros ...
// Mediante la función qsort ...
      t1 = clock();
      qsort((void*)numeros, TAM, sizeof(long), ordenar);
      t2 = clock();
      printf("t2 - t1 = %ld.", t2 - t1);
}

int ordenar(void *a, void *b)
{
      if(*(long*)a > * (long*)b)
            return 1;
      else if(*(long*)a < *(long*)b)
            return -1;
      return 0;
}
```

Si ejecuta ahora este programa, obtendrá igualmente la ordenación de los elementos del vector. Pero ahora el valor que saldrá por pantalla es del orden de 500 veces más bajo.

El algoritmo de ordenación de la burbuja es muy cómodo de implementar, y es eficaz para la ordenación de unos pocos centenares de enteros. Pero cuando hay que ordenar grandes cantidades, no es suficiente con que el procedimiento sea teóricamente válido: además debe ser eficiente.

Programar no es sólo poder en un lenguaje una serie de instrucciones. Además de saber lenguajes de programación es conveniente conocer de qué algoritmos se disponen para la solución de nuestros problemas. O echar mano de soluciones ya adoptadas, como en este caso, la implementación de la función *qsort*.

Creación de MACROS

La directiva ***#define***, que ya hemos visto, permite la definición de macros.

Una macro es un bloque de sentencias a las que se les ha asignado un nombre o identificador.

Una macro no es una función. Es código que se inserta allí donde aparece su identificador.

Veamos un ejemplo sencillo:

```
#include <stdio.h>
// Definición de la macro ...
#define cuadrado(x) x * x
void main(void)
{
      short a;
      unsigned long b;
      printf("Intoduzca el valor de a ... ");
      scanf("%hd",&a);
      printf("El cuadrado de %hd es %lu", a, cuadrado(a));
}
```

cuadrado NO es una función, aunque su invocación tenga la misma apariencia. En el código no aparece ni un prototipo con ese nombre, ni

su definición. Es una macro: el código "*x * x*" aparecerá allí donde en nuestro programa se ponga *cuadrado(x)*.

#define es una directiva del compilador. Antes de compilar, se busca en todo el texto todas las veces donde venga escrita la cadena "*cuadrado(expresión)*". Y en todas ellas sustituye esa cadena por la segunda parte de la directiva *define*: en este caso, lo sustituye por la cadena "*expresión * expresión*". En nuestro ejemplo hemos calculado el cuadrado de la variable *a*; en general, se puede calcular el cuadrado de cualquier expresión.

En definitiva una macro es un bloque de código que se va a insertar, previamente a la compilación, en todas aquellas partes de nuestro programa donde se encuentre su identificador.

Una macro puede hacer uso de otra macro. Por ejemplo:

```
#define cuadrado(x) x * x
#define circulo(r) 3.141596 * cuadrado(r)
```

La macro *circulo* calcula la superficie de una circunferencia de radio *r*. Para realizar el cálculo, hace uso de la macro *cuadrado*, que calcula el cuadrado del radio. La definición de una macro debe preceder siempre a su uso. No se podría definir la macro *circulo* como se ha hecho si, previamente a su definición, no estuviera recogida la definición de la macro *cuadrado*.

Las macros pueden llegar a ser muy extensas. Vamos a rehacer el código del programa del tema anterior sobre la criba de Erastóthenes, usando macros en lugar de funciones.

Ejemplo de MACRO: la Criba de Erastóthenes

El código mediante funciones que resuelve la criba de Erastóthenes ha quedado resuelto al final de un tema anterior. El propósito ahora es rehacer toda la aplicación sin hacer uso de funciones: definiendo macros.

Para poder ver la diferencia entre utilizar macros y utilizar funciones convendrá presentar de nuevo las dos funciones que se habían definido para la aplicación, y poder comparar cómo se rehacen mediante una directiva de procesador.

Las dos funciones eran la función Criba y la función TablaPrimos. La primera era:

```
long Criba(char* num, long rango)
{
      long i, j;
// En principio marcamos todos los elementos como PRIMOS
      for(i = 0 ; i < rango; i++)
            num[i] = 'p';
      for(i = 2 ; i < sqrt(rango) ; i++)
            if(num[i] == 'p')
                  for(j = 2 * i ; j < rango ; j += i)
                        num[j] = 'c';
      for( i = 1 , j = 0 ; i < rango ; i++)
            if(num[i] == 'p') j++;
      return j;
}
```

Ahora, con la macro, que hemos llamado ***__Criba***, queda de la siguiente forma:

```
#define __Criba(_num, _pr)                                  \
{                                                           \
long _i, _j;                                                \
for(_i = 0 ; _i < MAX; _i++) _num[_i] = 'p';                \
for(_i = 2 ; _i < sqrt(MAX) ; _i++)                         \
      if(_num[_i] == 'p')                                   \
            for(_j = 2 * _i ; _j < MAX ; _j += _i)          \
                  _num[_j] = 'c';                           \
for(_i = 1 , _j = 0 ; _i < MAX ; _i++)                      \
      if(_num[_i] == 'p') _j++;                             \
_pr = _j;                                                   \
}
```

Las barras invertidas al final de cada línea indican que aunque hay un salto de línea en el editor, el texto continúa en la línea siguiente. Deben ponerse tal cual, sin espacio en blanco alguno posteriormente a ellas.

Las diferencias principales con respecto al código de la función se basan en las siguientes peculiaridades de las macros:

1. El nombre: mucha gente habitúa a preceder al nombre de las macros uno o varios caracteres subrayado; nosotros hemos utilizado dos. Es cuestión de criterio personal, y es muy conveniente usar un criterio de creación de identificadores especial para las macros. Si se decide que las macros comienzan con dos caracteres subrayado, y tenemos la disciplina de trabajo de no crear jamás, en código normal, un identificador con ese inicio, entonces es imposible que la macro pueda generar confusión. Dentro de las macros, es habitual también darle un formato especial a los nombres de las variables que se definan en ellas: en el ejemplo se han tomado todas las variables, y todos los parámetros de la macro con un carácter subrayado al principio. Es muy importante dar nombres especiales a esas variables: hay que tener en cuenta que el código se inserta tal cual en la función que invoca a la macro: en nuestro ejemplo, si las variables de la macro se hubieran llamado *i* y *j* en lugar de *_i* y *_j*, tendríamos un error de compilación, porque esas variables, con el identificador *i* y con el identificador *j* ya están creadas en la función principal que utiliza las macros.
2. Hay que considerar que la macro lo que hace es insertar el código en el lugar donde se coloca el identificador: si necesitamos crear variables, habrá que definir la macro como un bloque (comenzar y terminar con llaves) para no tener que arrastrar luego todas esas variables en el ámbito de la función que llama a la macro. Si la función que convertimos en macro devolvía un valor, ahora habrá que ver la manera de que ese valor quede recogido al final de la macro: lo habitual será que se pase como parámetro la variable donde iba a quedar almacenado el valor que devolvía la función. En nuestro caso hemos pasado como parámetro la variable *pr*. La variable que en la función se llamaba rango ha quedado eliminada porque si trabajamos con macros podemos hacer uso de la que define el valor de MAX, cosa que evitábamos al redactar la función: no queríamos que la función tuviera un valor dependiente de una

macro definida en la aplicación donde se definía la función, para permitir que la función fuese lo más transportable posible, y no dependiente de un valor ajeno a su propia definición.

La segunda función era:

```
void TablaPrimos(char* num, long* primos, long rango)
{
      long i, j;
      for(i = 1 , j = 0 ; i < rango ; i++)
            if(num[i] == 'p')
            {
                  *(primos + j) = i;
                  j++;
            }
      *(primos + j) = 0;
}
```

Y ahora, con la macro, que hemos llamado ***__TablaPrimos***, queda de la siguiente forma:

```
#define __TablaPrimos(_num, _primos)                    \
{                                                       \
long _i, _j;                                            \
for(_i = 1 , _j = 0 ; _i < MAX ; _i++)                  \
      if(_num[_i] == 'p')                               \
      { *(_primos + _j) = _i;                           \
      _j++;}                                            \
*(_primos + _j) = 0;                                    \
}
```

Donde de nuevo hemos eliminado el uso del tercer parámetro que se recogía en una variable llamada *rango*. Hemos mantenido el criterio para la asignación de nombres. Hemos recibido como parámetros los dos punteros: en la macro se llaman con una carácter subrayado previo: cuando se sustituye el código de la macro en el programa, antes de la compilación, el nombre que se recoge es el que se haya consignado entre paréntesis en la llamada de la macro: y ahí los nombres van sin esos caracteres subrayado.

Para un usuario que no haya definido la macro, el que un bloque de código sea macro o sea función es algo que no ha de saber. El modo de invocación es el mismo (cambiando los parámetros). Muchas de las

funciones estándares de C hacen uso de macros; otras no son realmente tales, sino que son macros.

La ventaja de la macro es que el código queda insertado en la aplicación antes de la compilación, de forma que su uso no exige la llamada a una función, el apile en memoria de las variables que se deben guardar mientras salimos de un ámbito para meternos en el ámbito de la nueva función en ejecución, etc. El uso de macros reduce los tiempos de ejecución de las aplicaciones notablemente. Una macro consume, de media, un 20 % menos del tiempo total que tardaría, en hacer lo mismo, el código definido en forma de función.

Funciones con un número variable de argumentos

Hasta el momento hemos visto funciones que tienen definido un número de parámetros concreto. Y son, por tanto, funciones que al ser invocadas se les debe pasar un número concreto y determinado de parámetros.

Sin embargo no todas las funciones que hemos utilizado son realmente así de rígidas. Por ejemplo, la función *printf*, tantas veces invocada en nuestros programas, no tiene un número prefijado de parámetros:

```
printf(“Aquí solo hay un parametro.”);   // Un parámetro

printf(“Aquí hay %ld parámetros.”, 2);   // Dos parámetros

printf(“Y ahora %ld%c”, 3, ‘.’);         // Tres parámetros
```

Vamos a ver en este epígrafe cómo lograr definir una función en la que el número de parámetros sea variable, en función de las necesidades que tenga el usuario en cada momento.

Existen una serie de macros que permiten definir una función como si tuviera una lista variable de parámetros. Esas macros, que ahora veremos, están definidas en la biblioteca **stdarg.h**.

El prototipo de las funciones con un número variable de parámetros es el siguiente:

tipo nombre_funcion(tipo_1,[..., tipo_N], ...);

Primero se recogen todos los parámetros de la función que son fijos, es decir, aquellos que siempre deberán aparecer como parámetros en la llamada a la función. En el caso de la función *printf*, siempre debe aparecer, al principio, una cadena de caracteres, que viene recogida entre comillas. Si falta esta cadena en la función *printf* tendremos error en tiempo de compilación.

Y después de los parámetros fijos y obligatorios (como veremos más adelante, toda función que admita un número variable de parámetros, al menos deberá tener un parámetro fijo) vienen tres puntos suspensivos. Esos puntos deben ir al final de la lista de argumentos conocidos, e indican que la función puede tener más argumentos, de los que no sabemos ni cuántos ni de qué tipo de dato.

La función que tiene un número indeterminado de parámetros, deberá averiguar cuáles recibe en cada llamada. La lista de parámetros deberá ser recogida por la función, que deberá deducir de esa lista cuáles son los parámetros recibidos. Para almacenar y operar con esta lista de argumentos, está definido, en la biblioteca **stdarg.h**, un nuevo tipo de dato de C, llamado ***va_list*** (podríamos decir que es el tipo de “lista de argumentos”). En esa biblioteca viene definido el tipo de dato y las tres macros empleadas para operar sobre objetos de tipo lista de argumentos. Este tipo de dato tendrá una forma similar a una cadena de caracteres.

Toda función con un número de argumentos variable deberá tener declarada, en su cuerpo, una variable de tipo de dato ***va_list***.

```
tipo nombre_funcion (tipo_1,[..., tipo_N], ...)
{
      va_list argumentos /* lista de argumentos */
```

Lo primero que habrá que hacer con esta variable de tipo ***va_list*** será inicializarla con la lista de argumentos variables recibida en la llamada a la función.

Para inicializar esa variable se emplea una de las tres macros definidas en la biblioteca **stdarg.h**: la macro ***va_start***, que tiene la siguiente sintaxis:

void va_start(va_list ap, lastfix);

Donde ***ap*** es la variable que hemos creado como de tipo de dato ***va_list***, y donde ***lastfix*** es el último argumento fijo de la lista de argumentos.

La macro ***va_start*** asigna a la variable *ap* la dirección del primer argumento variable que ha recibido la función. Necesita, para esta operación, recibir el nombre del último parámetro fijo que recibe la función como argumento. Se guarda en la variable *ap* la dirección del comienzo de la lista de argumentos variables. Esto obliga a que en cualquier función con número de argumentos variable exista al menos un argumento de los llamados aquí fijos, con nombre en la definición de la función. En caso contrario, sería imposible obtener la dirección del comienzo para una lista de los restantes argumentos.

Ya tenemos localizada la cadena de argumentos variables. Ahora será necesario recorrerla para extraer todos los argumentos que ha recibido la función en su actual llamada. Para eso está definida una segunda macro de **stdarg.h**: la macro ***va_arg***. Esta rutina o macro extrae el siguiente argumento de la lista.

Su sintaxis es:

tipo va_arg(va_list ap, tipo);

Donde el primer argumento es, de nuevo, nuestra lista de argumentos, llamada ***ap***, que ya ha quedado inicializada con la macro ***va_start***. Y ***tipo*** es el tipo de dato del próximo parámetro que se espera encontrar.

Esa información es necesaria: por eso, en la función *printf*, indicamos en el primer parámetro (la cadena que ha de ser impresa) los especificadores de formato.

La rutina va extrayendo uno tras otro los argumentos de la lista variable de argumentos. Para cada nuevo argumento se invoca de nuevo a la macro. La macro extrae de la lista *ap* el siguiente parámetro (que será del tipo indicado) y avanza el puntero al siguiente parámetro de la lista. La macro devuelve el valor extraído de la lista. Para extraer todos los elementos de la lista habrá que invocar a la macro ***va_arg*** tantas veces como sea necesario. De alguna manera la función deberá detectar que ha terminado ya de leer en la lista de variables. Por ejemplo, en la función *printf*, se invocará a la macro ***va_arg*** tantas veces como veces haya aparecido en la primera cadena un especificador de formato: un carácter % no precedido del carácter '\'.

Si se ejecuta la macro ***va_arg*** menos veces que parámetros se hayan pasado en la actual invocación, la ejecución no sufre error alguno: simplemente dejarán de leerse esos argumentos. Si se ejecuta más veces que parámetros variables se hayan pasado, entones el resultado puede ser imprevisible.

Si, después de la cadena de texto que se desea imprimir, la función *printf* recoge más expresiones (argumentos en la llamada) que caracteres '%' ha consignado en la cadena (primer argumento de la llamada), no pasará percance alguno: simplemente habrá argumentos que no se imprimirán y ni tan siquiera serán extraídos de la lista de parámetros. Pero si hay más caracteres '%' que variables e nuestra lista variable de argumentos, entonces la función *printf* ejecutará la macro ***va_arg*** en busca de argumentos no existentes. En ese caso, el resultado será completamente imprevisible.

Y cuando ya se haya recorrido completa la lista de argumentos, entonces deberemos ejecutar una tercera rutina que restablece la pila de llamada a funciones. Esta macro es necesaria para permitir la

finalización correcta de la función y que pueda volver el control de programa a la sentencia inmediatamente posterior a la de la llamada de la función de argumentos variables.

Su sintaxis es:

void va_end(va_list ap);

Veamos un ejemplo. Hagamos un programa que calcule la suma de una serie de variables ***double*** que se reciben. El primer parámetro de la función indicará cuántos valores intervienen en la suma; los demás parámetros serán esos valores. La función devolverá la suma de todos ellos:

```
#include <stdio.h>
#include <stdarg.h>

double sum(long, ...);

void main(void)
{
      double S;
      S = sum(7, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0);
      printf("%f",S);
}

double sum(long v,...)
{
      double suma = 0;
      long i;
      va_list sumandos;
      va_start(sumandos, v);
      for(i = 0 ; i < v ; i++)
            suma += va_arg(sumandos,double);
      va_end(sumandos);
      return suma;
}
```

La función *sumaf* recibe un único parámetro fijo, que es el que indica cuántas variables más va a recibir la función. Así cuando alguien quiere sumar varios valores, lo que hace es invocar a la función *sumaf* indicándole en un primer parámetro el número de sumandos y, a continuación, esos sumandos que se ha indicado.

Después de inicializar la variable sumandos de tipo ***va_list*** mediante la macro ***va_start***, se van sumando todos los argumentos recibidos en la variable *suma*. Cada nuevo sumando se obtiene de la cadena *sumandos* gracias a una nueva invocación de la macro ***va_arg***. Tendré tantas sumas como indique el parámetro fijo recibido en la variable *v*.

Al final, y antes de la sentencia ***return***, ejecutamos la macro que restaura la pila de direcciones de memoria (***va_end***), de forma que al finalizar la ejecución de la función el programa logrará transferir el control a la siguiente sentencia posterior a la que invocó la función de parámetros variables.

Observación: las funciones con parámetros variables presentan dificultades cuando deben cargar, mediante la macro ***va_arg***, valores de tipo ***char***, ***unsigned char*** y valores ***float***. Hay problemas de promoción de variables y los resultados no son finalmente los esperados.

Argumentos de la línea de órdenes

Ya hemos explicado que en todo programa, la única función ejecutable es la función principal: la función ***main***. Un programa sin función principal puede ser compilable, pero no llegará a generar un programa ejecutable, porque no tiene la función de arranque.

Así, vemos que todas las funciones de C pueden definirse con parámetros: es decir, pueden ejecutarse con unos valores de arranque, que serán diferentes cada vez que esa función sea invocada.

También se puede hacer eso con la función ***main***. En ese caso, quien debe pasar los parámetros de arranque a la función principal será el usuario del programa compilado en el momento en que indique al sistema operativo el inicio de esa ejecución de programa.

En muchos sistemas operativos (UNIX especialmente) es posible, cuando se ejecuta un programa compilado de C, pasar parámetros a la función ***main***. Esos parámetros se pasan en la **línea de comandos** que lanza la ejecución del programa. Para ello, en esa función principal se debe haber incluido los siguientes parámetros:

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

Donde ***argc*** recibe el número de argumentos de la línea de comandos, y ***argv*** es un array de cadenas de caracteres donde se almacenan los argumentos de la línea de comandos.

Los usos más comunes para los argumentos pasados a la función principal son el pasar valores para la impresión, el paso de opciones de programa (muy empleado eso en unix, o en DOS), el paso de nombres de archivos donde acceder a información en disco o donde guardar la información generada por el programa, etc.

La función ***main*** habrá recibido tantos argumentos como diga la variable ***argc***. El primero de esos argumentos es siempre el nombre del programa. Los demás argumentos deben ser valores esperados, de forma que la función principal sepa qué hacer con cada uno de ellos. Alguno de esos argumentos puede ser una cadena de control que indique la naturaleza de los demás parámetros, o al menos de alguno de ellos.

Desde luego, los nombres de las variables ***argc*** y ***argv*** son mera convención: cualquier identificador que se elija servirá de la misma manera.

Y un último comentario. Hasta el momento, siempre que hemos definido la función ***main***, la hemos declarado de tipo ***void***. Realmente esta función puede ser de cualquier tipo, y en tal caso podría disponer de una sentencia ***return*** que devolviese un valor de ese tipo de dato.

Veamos un ejemplo. Hacemos un programa que al ser invocado se le pueda facilitar una serie de datos personales, y los muestre por pantalla.

El programa espera del usuario que introduzca su nombre, su profesión y/o su edad. Si el usuario quiere introducir el nombre, debe precederlo con la cadena “-n”; si quiere introducir la edad, deberá precederla la cadena “-e”; y si quiere introducir la profesión, deberá ir precedida de la cadena “-p”.

```
#include <stdio.h>
#include <string.h>
void main(int argc, char*argv[])
{
	char nombre[30];
	char edad[5];
	char profesion[30];
	nombre[0] = profesion[0] = edad[0] = '\0';
	long i;
	for(i = 0 ; i < argc ; i++)
	{
		if(strcmp(argv[i],"-n") == 0)
		{
			i++;
			if(i < argc) strcpy(nombre, argv[i]);
		}
		else if(strcmp(argv[i],"-p") == 0)
		{
			i++;
			if(i < argc) strcpy(profesion, argv[i]);
		}
		else if(strcmp(argv[i],"-e") == 0)
		{
			i++;
			if(i < argc) strcpy(edad, argv[i]);
		}
	}
	printf("Nombre: %s\n", nombre);
	printf("Edad: %s\n", edad);
	printf("Profesion: %s\n", profesion);
}
```

Si ha entrado la cadena “–n”, entonces la siguiente cadena deberá ser el nombre: y así se almacena. Si se ha entrado la cadena “–p”, entonces la siguiente cadena será la profesión: y así se guarda. Y lo mismo ocurre con la cadena “–e” y la edad.

Al compilar el programa, quedará el ejecutable en algún directorio: en principio en el mismo en donde se haya guardado el documento .cpp. Si

ejecutamos ese programa (supongamos que le hemos llamado “datos”) con la siguiente línea de comando:

```
datos –p estudiante –e 21 –n Isabel
```

Aparecerá por pantalla la siguiente información:

```
Nombre: Isabel
Edad: 21
Profesion: estudiante
```

Ejercicios

66. ***Una vez ha creado un vector que contiene todos los primos menores que un límite superior dado, declare y defina una función que busque los primos enlazados. Se llaman primos enlazados a aquellos primos cuya diferencia es igual a dos, es decir, que son dos impares consecutivos: por ejemplo 11 t 13; 17 y 19, etc.*** (Sin resolver)

67. ***Lea el siguiente código y muestre la salida que ofrecerá por pantalla.***

```
#include <stdio.h>
#include <math.h>
#define CUADRADO(x) x*x
#define SUMA(x,y) CUADRADO(x) + CUADRADO(y)
#define RECTO(x,y) sqrt(SUMA(x,y))

void main(void)
{
      printf("%2.1f",RECTO(3,4));
}
```