Codificación en C

Traducción de lenguaje algorítmico a C

Jordi Riera i Baburés

PID_00154303

Índice

Introducción

A lo largo de la asignatura se ha estado utilizando el *lenguaje algorítmico* para describir algoritmos. Este lenguaje es fácilmente comprensible para nosotros, pero no por el ordenador.

Con el fin de poder ejecutar los algoritmos en el ordenador, se tendrán que describir en un lenguaje que éste pueda entender. En este módulo veremos como codificar algoritmos en lenguaje C. De hecho, si hablamos con propiedad, el ordenador todavía no puede entender los programas codificados en lenguaje C. Es necesario todo un proceso de compilación y enlazado con el fin de traducir los programas escritos en C a un lenguaje que el ordenador sea capaz de ejecutar. Si se dispone de las herramientas adecuadas, sin embargo, este proceso es automático y no requiere intervención, por lo que, a efectos de la asignatura, se puede considerar que el lenguaje C es comprendido por el ordenador.

Hablando del lenguaje C, se puede decir que es un lenguaje de alto nivel, imperativo y estructurado, eficiente en la ejecución de los programas y muy utilizado, especialmente en la ingeniería y los campos científicos. Claro está que hay otros lenguajes de alto nivel igualmente válidos y con una estructuración mayor, o incluso orientados a objetos. Sería el caso del propio C++, para citar a uno. El lenguaje C, sin embargo, tiene el inestimable valor añadido de ser un lenguaje sencillo, próximo al lenguaje algorítmico. Además, es fácil disponer de excelentes compiladores de C gratuitos, para casi cualquier plataforma sobre la que se quiera trabajar y con entornos de programación claros y funcionales.

El entorno de programación en lenguaje C se trata en el "Anexo de software" de la asignatura.

Este módulo didáctico, de todas maneras, no pretende en ningún caso ser ni un manual, ni tampoco una guía de referencia del lenguaje C. Al contrario, el enfoque del módulo se dirige a cubrir sólo aquella pequeña parte del C necesaria para poder codificar la notación algorítmica vista a la asignatura.

En otras asignaturas a lo largo de la carrera se completará el conocimiento del lenguaje C.

En este sentido, la organización del presente módulo refleja la de los módulos didácticos anteriores de la asignatura. Así, el estudio de cada uno de los diferentes apartados de este módulo se hará a continuación de la lectura del módulo didáctico homónimo de la asignatura. Por lo tanto, el presente módulo se irá estudiando en paralelo con el del resto de módulos, y el conocimiento del lenguaje C se irá completando a medida que se completa también el conocimiento del lenguaje algorítmico.

Objetivos

Los objetivos de este módulo son básicamente los siguientes:

1. Conocer la codificación en lenguaje C de las construcciones del lenguaje algorítmico.
2. Saber traducir a lenguaje C cualquier algoritmo expresado en lenguaje algorítmico.

1. Introducción a la programación en C

En este primer apartado veremos la estructura básica de un programa descrito en lenguaje C. no entraremos todavía en detalles sobre la traducción de la notación algorítmica a lenguaje C.

Ved el módulo "Introducción a la programación" si tenéis dudas sobre alguno de los conceptos usados en este apartado.

1.1. Nuestro primer programa en C

Nuestro primer programa en lenguaje C será un breve pero emotivo saludo universal. El programa podría ser similar a:

```
/* Programa hola.c - Saludo universal
   Jordi Riera, 2001 */
#include <stdio.h>
#include <stdlib.h>

int main()
{
      printf("Hola mundo!\n");      /* Escribe "Hola mundo!" por
                                          pantalla */

      system("PAUSE");        /* El programa espera una tecla */
      return 0;
}
```

Podéis probar el programa usando el entorno de programación. Consultad el "Anexo del software" para conocer los detalles.

Este sencillo trozo de código nos sirve para ilustrar algunas características importantes del lenguaje C, que comentaremos acto seguido.

La primera línea del programa es un comentario. En C, los comentarios están delimitados entre los caracteres "/*" y "*/". Tal como se ve en el ejemplo, pueden ocupar varios líneas. Evidentemente, los comentarios son siempre opcionales, pero recordad que un programa bien documentado será mucho más inteligible. Usaremos el primer comentario para indicar el nombre del programa, una breve descripción de lo que hace, al autor, etc.

Algunos entornos de programación que trabajan en C++ y C aceptan comentarios de una sola línea delimitados por los caracteres "//" y el final de la línea. Pero eso no es C estándar.

En nuestro programa de saludo, después del comentario inicial, nos encontramos dos líneas que incluyen dos ficheros de declaraciones estándares del lenguaje C. Estos ficheros de declaraciones se llaman ficheros de cabecera o ficheros de *include*. Son proporcionados por el fabricante del compilador y contienen declaraciones de funciones, variables y constantes que el programador puede usar a partir de la inclusión del fichero con la directiva **#include**. El fichero *stdio.h* contiene, entre de otros, la declaración de la función *printf* que utilizaremos más tarde. y el fichero *stdlib.h* la declaración de la función *system*.

Las directivas se han de escribir siempre…

... en la primera posición de una línea, ya que sino el compilador no las reconocería. Todas las directivas empiezan siempre por "#".

Acto seguido viene el programa principal. Los programas en C siempre tienen que declarar la función *main*, que lo primero que se ejecutará y que, opcionalmente, podrá llamar a otras funciones declaradas. La función *main* retorna un

entero, por convenio un 0 para denotar una finalización correcta, y no necesita parámetros.

El cuerpo de la función está delimitado por los caracteres "{" y "}" que llamaremos llaves. ¡En él encontramos una sentencia *printf* que imprime "Hola mundo!" por pantalla, una llamada al sistema operativo para que el programa espere una tecla, y una sentencia **return** que sale de la función *main* y acaba la ejecución del programa.

Las llaves "{" y "}" sirven para delimitar un bloque de código.

El código del programa de saludo universal muestra todavía una importante característica más del lenguaje C:

El lenguaje C distingue entre mayúsculas y minúsculas.

Así, tenemos que utilizar la función *printf* para escribir por pantalla. No podemos utilizar ni la función *PRINTF*, ni la *Printf*, ni la *printF*, que no estén declaradas en el fichero *stdio.h*. De la misma manera, tenemos que ir con cuidado con el uso del nombre correcto de todo objeto declarado en el programa.

1.2. Un programa más completo en lenguaje C

Acto seguido se muestra la traducción a lenguaje C del algoritmo *media* expresado en lenguaje algorítmico en el apartado 1.2 del módulo "Introducción a la programación". El algoritmo calcula la media de cuatro números:

```
/* Programa mediana.c - Media de cuatro números
   Jordi Riera, 2001
*/
#include <stdio.h>
#include <stdlib.h>

int main() {
/* Declaración de variables */
   int n, suma, y;
   float resultado;

   suma = 0;
   i = 1;
   while (i<=4) {
         scanf("%d", &n); /* Lee un numero decimal por el teclado */
         suma = suma + n; /*Acumula la suma de los números leídos */
         y = y + 1;
   }
   resultada = (float) suma / 4.0;
   printf( "%f", resultado);     /* Imprime la media por pantalla */

   system("PAUSE"); /* Hacemos que el programa espere una tecla */
   return 0;
}
```

No os preocupéis por el funcionamiento del algoritmo o por los detalles de cómo se ha hecho la traducción. Todos estos aspectos quedarán mucho más claros

en el siguiente apartado. Aquí se ha incluido el programa *media* para mostrar un código más completo que el programa de saludo universal.

2. Introducción a la algorítmica en C

En este apartado se verán los objetos elementales, así como las estructuras algorítmicas, acciones y funciones del lenguaje C, necesarias para traducir a programas en C los algoritmos del módulo "Introducción al algorítmica".

2.1. Objetos elementales del lenguaje C

Tal como habéis visto en el lenguaje algorítmico, un objeto tiene tres atributos: nombre o identificador, tipo y valor. Los nombres o identificadores de los objetos siguen las mismas normas que en el lenguaje algorítmico: tienen que empezar por una letra, y después pueden seguir letras, números o el carácter "_". Tampoco puede utilizar ninguna palabra reservada del lenguaje C (como **while**) para identificar un objeto. La tabla siguiente contiene nombres correctos, incorrectos y un breve comentario.

Nombre o identificador	¿Es correcto?	Comentario
ventas	Si	Es recomendable escribir en minúsculas.
_articulo	No	No empieza por un carácter alfabético.
aux2	Si	Poco recomendable. Es mejor usar nombres descriptivos.
1_articulo	No	No empieza por un carácter alfabético.
numero articulos	No	Los espacios en blanco no son permitidos.
numeroArticulos	Si	Es un nombre descriptivo.
piezaCamión	No	No se pueden utilizar caracteres especiales ni acentos (la "ñ" se considera un carácter especial, recordad que los ordenadores provienen del mundo anglosajón).
const	No	Es una palabra reservada del lenguaje C.

A continuación vamos a ver la traducción de los tipos elementales del lenguaje algorítmico al lenguaje C, así como los valores que éstos pueden tomar.

2.1.1. Tipos elementales en C

El lenguaje algorítmico define cuatro tipos elementales: booleano, entero, caracter y real. Acto seguido veremos cómo se traducen al lenguaje C.

Tipo booleano

El tipo booleano no existe como tal en el lenguaje C, pero lo podemos definir fácilmente usando la declaración siguiente:

```
typedef enum {FALSE, TRUE} bool;
```

Hay que ir con cuidado porque el orden de los elementos (FALSE y TRUE) es importante, como veremos más adelante.

A partir de este momento podemos utilizar el nuevo tipo *bool* para declarar objetos de tipo booleano.

La siguiente tabla resume las características del tipo booleano y su traducción a lenguaje C.

Característica	Lenguaje algorítmico	Lenguaje C
Identificador	booleano	bool
Rango de valores	cierto, falso	TRUE, FALSE
Operadores internos	no, y, o, =, ≠, <, ≤, >, ≥	`!, &&, ||, ==, !=, <, <=, >, >=`

Tipo caracter

En C los caracteres van delimitados por comillas simples como en el lenguaje algorítmico. La tabla siguiente resume las características del tipo carácter y su traducción a C.

Característica	Lenguaje algorítmico	Lenguaje C
Identificador	caracter	char
Rango de valores	conjunto finito de valores	conjunto finito de valores definidos por el código ASCII
Operadores externos	=, ≠, <, ≤, >, ≥	`==, !=, <, <=, >, >=`
Sintaxis de valores	'a', 'W', '1'	`'a', 'W', '1'`

Tipo entero

La siguiente tabla resume las características del tipo entero y su traducción a C.

Característica	Lenguaje algorítmico	Lenguaje C
Identificador	entero	int
Rango de valores	ENTEROMIN a ENTEROMAX	-2147483648, 2147483647
Operadores internos	- (cambio de signo), + -, *, div, mod	- (cambio de signo), +, -, *, /, %
Operadores externos	=, ≠, <, ≤, >, ≥	==, !=, <, <=, >, >=
Sintaxis de valores	3, 465434, -2134567	3, 465434, -2134567

El rango de los enteros puede variar según el compilador y el ordenador donde se ejecute el programa.

Tipo real

La siguiente tabla resume las características del tipo real y su traducción a C.

Característica	Lenguaje algorítmico	Lenguaje C
Identificador	real	float
Rango de valores	REALMIN a REALMAX (no todos)	±3.4E±38 (7 dígitos significativos)
Operadores internos	- (cambio de signo), + -, * /	- (cambio de signo), + -, * /
Operadores externos	=, ≠, <, ≤, >, ≥	==, !=, <, <=, >, >=
Sintaxis de valores	0.6, 5.0E-8, -49.2E+0.8, 1.0E5, 4.0	0.6, 5.0E-8, -49.2E+0.8, 1.0E5, 4.0

El lenguaje C permite reales con más dígitos significativos: los tipos *double* y *long double*.

2.1.2. Declaración de objetos en C

Antes de utilizar un objeto en un programa, hace falta haberlo declarado previamente. En el caso del lenguaje C, sin embargo, las declaraciones no van delimitadas por palabras reservadas como en el lenguaje algorítmico. Simplemente, se colocan las declaraciones de constantes y variables al inicio del cuerpo de la función correspondiente (función *main* si hacemos referencia al programa principal), antes de cualquier otra instrucción.

Para declarar una constante, se utiliza la construcción:

const *tipo nombre = valor;*

Notad que la palabra reservada **const** es obligatoria ante cada declaración de constante. No sirve para definir todo un bloque de declaraciones de constantes como en el lenguaje algorítmico, sino sólo para declarar una.

Este tipo de constante es conocida en tiempo de ejecución. Es decir, la asignación del *valor* a la constante *nombre* se ganará durante la ejecución del programa, y no durante la compilación.

A veces es conveniente definir constantes con valor conocido ya durante la compilación. Se puede hacer con la directiva **#define**:

Las constantes conocidas en tiempo de compilación serán útiles para la parametrización de tipos de datos estructurados, como las tablas. Se comentarán en el apartado 4.1.4. "Acceso directo a una tabla" del presente módulo.

#define *NOMBRE valor*

Observamos que no se indica el tipo de la constante que se está definiendo. De hecho, el compilador únicamente sustituye el *nombre* por su *valor* cada vez que lo encuentra en el código posterior a la declaración. Asimismo, es habitual usar un nombre escrito todo en mayúsculas al declarar una constante. De esta manera el código del programa es más legible, ya que se pueden distinguir las constantes de las variables con un solo vistazo.

Para declarar una variable, se usa la sintaxis siguiente:

tipo nombre;

Así se declara una variable de tipo *tipo* y nombre *nombre.* También se pueden declarar diversas variables del mismo tipo en una sola declaración, separando sus nombres por comas (","). Por ejemplo, para declarar tres variables i, *j*, *k* de tipo entero:

El lenguaje C…

... permite indicar un valor inicial por la variable en la propia declaración, pero evitaremos esta construcción que no es permitida en el lenguaje algorítmico.

```
int i, j, k;
```

Finalmente, remarcar que aunque el lenguaje C no usa palabras reservadas para delimitar las declaraciones de variables y constantes, nosotros colocaremos los comentarios apropiados para hacer el programa más legible.

2.1.3. Expresiones en C

Las expresiones en lenguaje C siguen las mismas reglas de construcción y evaluación que en el lenguaje algorítmico. En caso de duda, sin embargo, siempre es aconsejable poner un paréntesis de más para indicar claramente la orden de evaluación de la expresión.

2.1.4. Definición de tipos. Tipos enumerados en C

Tal como sucede con las declaraciones de variables y constantes, las declaraciones de tipo en C tampoco van delimitadas por palabras reservadas. Se tienen que colocar entre las declaraciones de las constantes y las de las variables, indicando con un comentario que se procede a hacer declaraciones de tipo.

El constructor de tipos enumerados en C sigue la siguiente sintaxis:

typedef enum *{$valor_1$, $valor_2$.., $valor_n$} nombre;*

Recordad que hemos visto un ejemplo de esta sintaxis en la definición del tipo booleano.

Como en el lenguaje algorítmico, el orden en que se enumeran los valores sirve para establecer las relaciones más pequeño a más grande y por lo tanto la orden en que se enumeran los valores es importante. En C se puede asignar un entero a cada valor, para establecer estas relaciones, pero nosotros no utilizaremos esta característica.

2.1.5. Funciones de conversión de tipos

Tal como pasa con el lenguaje algorítmico, a veces se necesitan funciones de conversión de tipo para pasar de un tipo de número (o código) a otro. Aunque el lenguaje C a menudo puede hacer estas conversiones automáticamente, lo mejor es indicar claramente en el programa el tipo de conversión deseada.

La siguiente tabla muestra la traducción de las funciones de conversión de tipo del lenguaje algorítmico a C.

Lenguaje algorítmico	Lenguaje C	Ejemplo
realAEntero	(**int**)	(**int**) -3. 99
enteroAReal	(**float**)	(**float**) 3
caracterACodigo	(**int**)	(**int**) 'A'
codigoACaracter	(**char**)	(**char**) 65

2.2. Especificación de algoritmos

La especificación de algoritmos, acciones y funciones en lenguaje C se hará con la utilización de comentarios, tal como se hacía en la notación algorítmica. En este caso, sin embargo, recordad usar la sintaxis de C para los comentarios.

```
/* Pre: precondición */
algoritmo
/* Post: postcondición */
```

2.3. Estructuras algorítmicas

En este apartado se muestran los equivalentes en C de las estructuras básicas del lenguaje algorítmico.

2.3.1. Estructura general de un programa en lenguaje C

La estructura de un programa en lenguaje C es la siguiente:

```
/* Programa estructura.c - Estructura básica de un programa en C
    Jordi Riera, 2001
*/
/* Inclusión de ficheros de cabecera */
#include <fichero1>
#include <fichero2>
...
/* Declaración de constantes en tiempo de compilación */
#define NOMBRE_DEFINE1 valor_define1
#define NOMBRE_DEFINE2 valor_define2
...
/* Declaración de tipos */
typedef enum {FALSE, TRUE} bool;        /* tipo booleano */
typedef definicion_de_tipo1 nombre_tipo1;
typedef definicion_de_tipo2 nombre_tipo2;
…
int main() {
   /* Definición de constantes */
   const tipo_de_constante1 nombre_constante1 = valor_constante1;
   const tipo_de_constante2 nombre_constante2 = valor_constante2;
   ...
   /* Declaración de variables */
   tipo_de_variable1 nombre_variable1;
   tipo_de_variable2 nombre_variable2;

   /* Cuerpo del programa */
   acción1
   acción2
   ...
}
```

Observad que las declaraciones de tipos se han hecho fuera de la función *main* del programa. Eso es porque habitualmente es deseable que los tipos de datos estén disponibles en todo el programa y para todas las funciones que en él se puedan declarar. De todas maneras, también se pueden hacer declaraciones de tipo locales, situándolas entre las declaraciones de las constantes y las de las variables.

2.3.2. Acciones elementales: la asignación

La acción de asignación del lenguaje algorítmico, ":=" se traduce en el operador de asignación en lenguaje C, "=". La distinción entre acción y operador es muy importante, ya que implica que el lenguaje C permite hacer asignaciones en cualquier punto del código donde una expresión sea válida. Por ejemplo, es un error muy común hacer una asignación, en vez de una comparación en la condición de una sentencia condicional, por usar el operador de asignación "=" en lugar del operador relacional de igualdad "==", lo cual tiene graves repercusiones de cara al buen funcionamiento del programa.

2.3.3. Composición de acciones

Veremos ahora como combinar acciones elementales para poder construir programas más complejos.

Composición secuencial

En C, la composición secuencial se hace como en el lenguaje algorítmico, escribiendo una acción a continuación de la otra y separando cada acción de la siguiente con un ";".

Si queréis agrupar un conjunto de acciones en un bloque, las podéis colocar entre las llaves "{" y "}".

Composición alternativa

La sintaxis de la composición alternativa de acciones en lenguaje C es la siguiente:

```
if (expresión) {
    acciónA
} else {
    acciónB
}
```

Notad que el paréntesis entorno a la *expresión* es obligatorio y forma parte de la sintaxis. Las acciones a ejecutar condicionalmente (*acciónA* y acciónB) son bloques de acciones delimitados por llaves (composición secuencial). Si sólo hubiera que ejecutar una acción simple, se podrían eliminar las llaves correspondientes.

Es recomendable utilizar siempre las llaves, aunque sólo haya que colocar una acción. ¡Quizás más adelante habrá que añadir otras nuevas!

Al igual que en el lenguaje algorítmico, se puede eliminar la parte correspondiente al **else** si no deseemos no ejecutar ninguna acción en caso de que la condición sea falsa.

Composición iterativa

La sintaxis de la composición iterativa en lenguaje C es la siguiente:

```
while (expresión) {
    acción
}
```

Notad que el paréntesis entorno a la *expresión* es obligatorio. También, como en la composición alternativa, se pueden eliminar las llaves si sólo hay una sentencia en el cuerpo del **while**, pero es más recomendable ponerlas siempre

El lenguaje C dispone también de una variante de la composición alternativa que nos será especialmente útil cuando conozcamos exactamente el número de repeticiones que hace falta hacer. Su sintaxis es la siguiente:

```
for (índice = valor_inicial; índice <= valor_final; índice++) {
    acción
}
```

Notad la diferencia...

En el lenguaje algorítmico no aparece la condición de finalización (sólo aparece "hasta valor_final"); en cambio, en C aparece la condición completa "indice <= valor_final".

Fijaos que aparentemente en esta sintaxis no se especifica qué *incremento* se hará en cada paso. De hecho, se está indicando que se hará un *incremento* unitario con la operación *índice*++. Si queréis un decremento unitario (incremento = -1) tenéis que sustituir *índice*++ por *índice*--. Asimismo, para incrementos no unitarios, positivos y negativos, usaréis las operaciones *índice* += *incremento* e *índice* -= *incremento* respectivamente.

Notad que...

... ++, --, += y -= son operadores del lenguaje C. De hecho, C tiene muchos otros operadores todavía. Nosotros no los utilizaremos, ya que nos concentramos en la manera más sencilla y sistemática de traducir la notación algorítmica al lenguaje C. Podéis, sin embargo, encontrar más información en el apartado "Construcciones avanzadas del lenguaje C" de este mismo módulo..

Podríamos hacer el equivalente del for con un while

```
indice = valor_inicial;
while (indice <= valor_final)
{
   acción;
   indice++;
}
```

Usando la construcción **for** se puede codificar el programa que calcula el factorial de un número entero.

```
/* Programa fact.c - Calcula el factorial de n
   Jordi Riera, 2001
*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
   /* Declaración de variables */
   int i, n, fact;

   scanf("%d", &n);  /* Leemos uno numero entero por teclado */
   /* Pre: n=N y N>=0 y N<=16 */
   fact = 1;
   for (i=1; i<=n; i++) {
         fact = fact * i;
   }
   /* Post: fact es el factorial de N */
   printf("%d\n", fact); /* Escribimos el resultado por pantalla */

   system("PAUSE");  /* Hacemos que el programa espere una tecla */
   return 0;
}
```

Teneis otra solución al algoritmo fact, usando while en lugar de for, en el apartado 3.3.3. “Composición iterativa”, del módulo “Introducción a la algorítmica” de la asignatura.

Observad que hemos añadido una nueva condición en la precondición, concretamente N<=16. Esta condición no viene impuesta por una limitación del algoritmo, sino por su codificación en un programa en C. La limitación está en el entero mayor que se puede representar en el ordenador. Observad qué pasa si introducís por teclado el número 17, por ejemplo.

2.4. Acciones y funciones en C

El lenguaje C no dispone de acciones en el sentido de la notación algorítmica. En C, todo son funciones. Recordad que incluso el algoritmo principal se escribe dentro de la función *main*.

Para una definición precisa de lo que son las acciones y las funciones, consultad el apartado 4. “Acciones y funciones” del módulo “Introducción a la algorítmica” de la asignatura.

De todas maneras, en lenguaje C también se puede distinguir entre acciones y funciones. Se hará según la manera de declarar parámetros y la forma de hacer las invocaciones. En una función sólo se declararán parámetros de entrada, y se invocará dentro de una expresión. En una acción se podrán declararán parámetros de todo tipo, y se invocará allí donde una acción sea válida.

Considerando estas restricciones, la sintaxis para la declaración de una acción en C es:

void *nombre*(param$_1$, *param$_2$*.., *param$_n$*)
{
   ... cuerpo de la acción
}

void...

... es un tipo especial del lenguaje C que indica que la función no devuelve ningún valor.

Y la correspondiente invocación de la acción es:

```
nombre(obj1, obj2.., objn);
```

En el caso de una función, la sintaxis para su declaración en C es:

```
tipo nombre(param1, param2.., paramn)
{
    ... cuerpo de la función
    return expresión;
}
```

Y la correspondiente invocación de la función es:

```
nombre(obj1, obj2.., objn)
```

La invocación sólo será válida dentro de una expresión donde se espera un valor del tipo que devuelve la función.

Tal como sucede en el lenguaje algorítmico, las acciones y funciones en C pueden declarar su entorno local, con definiciones de constantes, tipo y variables propias. La sintaxis es la misma que se ha utilizado en la función *main*.

También, al igual que las variables, en C hay que declarar las acciones y funciones antes de su uso. Una práctica muy conveniente es *predeclarar* todas las funciones y acciones al principio del fichero (después de los includes, defines y typedefs) con lo cual podemos declararlas después allí dónde queramos y nos podremos despreocupar a la hora de utilizarlas de si ya las hemos declarado o no.

La predeclaración de la acción/función es muy parecida a la primera línea de la declaración, sin embargo, en lugar de los parámetros se pueden poner sólo sus tipos:

```
tipo nombre(tipus1, tipus2.., tipon);     /* función */
void nombre2(tipus1, tipus2.., tipon);      /* acción */
```

Igualmente es más recomendable, por claridad, poner todo el parámetro como en la declaración:

```
tipo nombre(param1, param2.., paramn);      /* función */
void nom2(param1, param2.., paramn);        /* acción */
```

2.4.1. Parámetros en C

El paso de parámetros en lenguaje C sigue las directrices explicadas en el apartado 4.2. del módulo "Introducción al algorítmica". Es decir, hay que especificar su tipo en la cabecera de la acción o función, y el orden de los parámetros actuales en la invocación tiene que ser el mismo que el de los parámetros formales en la declaración de la función.

Cuando llamamos a la función, el lenguaje C siempre hace una copia de cada uno de los parámetros actuales y esta copia es la que se asocia a los parámetros formales. Por lo tanto, las modificaciones que hacemos al cuerpo de la función no afectan a los parámetros actuales. Podríamos decir que es como si los parámetros fueran de entrada. Este mecanismo de paso de parámetros, en el que pasamos una copia del objeto, se conoce como paso de parámetros *por valor*.

La forma de simular parámetros de entrada/salida consiste en pasar la dirección de memoria del parámetro actual en lugar del parámetro en sí.
De esta forma después, desde la acción, podremos acceder a memoria y modificar el parámetro actual. Este otro mecanismo de paso de parámetros, en el que pasamos la dirección de memoria del objeto, es el que se llama *paso por referencia*.

Por último, en C los parámetros de salida no existen propiamente. En realidad serán parámetros de entrada/salida donde su valor de entrada se ignora o no interesa.

A partir de ahora, cuando hablemos de parámetros de entrada/salida nos referiremos también a los parámetros de salida.

IMPORTANTE: Para trabajar con direcciones de memoria en C utilizamos un nuevo tipo de datos al que llamamos **apuntador**. Un apuntador no es más que una dirección de memoria.
Mirad el apartado 6.4 de este manual para más información sobre los apuntadores que os será útil para entender bien lo que viene a continuación.

La sintaxis de los parámetros en la cabecera de una acción o función es:

```
para un parámetro de entrada:          tipo nombre
para un parámetro de salida:           tipo *nombre
para un parámetro de entrada/salida:   tipo *nombre
```

Notad como en la declaración de parámetros de entrada/salida el nombre va precedido por un "*". Eso sirve para indicar que el parámetro es un apuntador, es decir, una dirección de memoria que contiene un objeto del tipo declarado.

De hecho...

... & y * son operadores unarios de C que permiten, respectivamente, obtener la dirección de memoria donde se guarda un objeto y acceder al objeto contenido en una dirección de memoria.

Dentro del cuerpo de las acciones/funciones, el operador "*" se tiene que utilizar para acceder a los objetos que se corresponden con los parámetros de entrada/salida, ya sea para actualizarlos o para obtener su valor.

Como podéis ver hay dos usos bien diferenciados de este operador "*":

- Declaración de la cabecera de una acción con parámetros de entrada/salida.
- En el cuerpo de la acción para acceder a los objetos correspondientes a los parámetros de entrada/salida.

Por regla general en la llamada a una acción todos los parámetros reales de entrada/salida deben ir precedidos por el operador unario "**&**". Este operador "&", ante una variable, permite construir un apuntador a la mencionada variable (recordad la llamada a *scanf* para leer un número del teclado en el programa que calcula el factorial, por ejemplo). Pero hay una excepción: cuando se invoca una acción B desde otra acción A y alguno de los parámetros de entrada/salida de la llamada a B ya era un parámetro de entrada/salida en A. Vedlo en el siguiente ejemplo:

```
/* Programa Ejemplo.c.- Sirve para ilustrar el uso de los
   parámetros de entrada/salida */
#include <stdio.h>
#include <stdlib.h>

/* Predeclaración de funciones */
void accion_A(int *param_ent_sal);
                    /* Declaramos un parámetro de entrada/salida */
void accion_B(int *param_ent_sal);
                    /* Declaramos un parámetro de entrada/salida */
void accion_C(int param_ent);
                    /* Declarem un paràmetre d'entrada           */

void accion_A (int *param_ent_sal)
{
  int param_local;

  param_local=*param_ent_sal;
                    /*param_local toma el valor del objeto apuntado
                          por param_ent_sal */
  accion_B(param_ent_sal);
                    /* Atención: No ponemos & delante de
                          param_ent_sal porque ya tenemos la
                          dirección del objeto */
  accion_C(*param_ent_sal);
                    /* Hemos de pasar el objeto como parámetro de
                          entrada */

  accion_B(&param_local);
                          /* Pasamos la variable como a parámetro
                                de entrada/salida */
  accion_C(param_local);  /* Pasamos la variable como parámetro de
                                entrada */
}
```

```
void accion_B(int *numero)
{
   *numero=*numero + 1;    /* Dentro del cuerpo de la acción los
                              parámetros de entrada/salida deben
                              ir precedidos por "*" */
}

void accion_C(int numero)
{
   printf("%d",numero);
}

int main()
{
   int numero;
   scanf("%d",&numero);  /* Debemos pasar la dirección de la
                            variable */
   accion_A(&numero); /* la acción A espera un parámetro de
                         entrada/salida por tanto ponemos el
                         & delante de la variable */
   system("PAUSE");
   return 0;
}
```

2.4.2. Acciones y funciones de entrada y salida de datos en C

La tabla siguiente muestra los equivalentes del lenguaje C de las funciones predefinidas en el lenguaje algorítmico (suponemos que tenemos declaradas las variables *e*, *r* y *c* como entero, real y carácter respectivamente):

Lenguaje algorítmico	Lenguaje C
e = leerEntero();	`scanf("%d", &e);`
r = leerReal();	`scanf("%f", &r);`
c = leerCaracter();	`scanf("%c", &c);`
escribirEntero(e);	`printf("%d ", e);`
escribirReal(r);	`printf("%f ", r);`
escribirCaracter(c);	`printf("%c ", c);`

scanf y printf...

... son funciones de entrada y salida de datos del lenguaje C, que permiten controlar muchos detalles que el lenguaje algorítmico no contempla. Usaremos las versiones simplificadas de la tabla adjunta con el fin de concentrar esfuerzos en el diseño de los algoritmos y dejar la codificación como un trabajo casi mecánico de traducción a C.

Notad el último espacio en el primer parámetro del *printf*. Si no estuviera, la salida sería ilegible, ya que se escriben todos los objetos uno a continuación del otro, sin ninguna separación.

Por ejemplo:

```
...
printf("%d", 1234);
printf("%d", 567);
...
```

La ejecución del programa escribe los caracteres `"1234567"` por el dispositivo de salida. Observad cómo no se distingue dónde acaba el primer número y empie-

za el segundo... En cambio, si en el primer parámetro del *printf* ponemos `"%d "` en vez de `"%d"`, la salida sería `"1234 567"`, que es la que nos interesa.

2.5. Ejemplos de codificación en C

En este apartado veremos codificados en lenguaje C algunos de los ejercicios de auto evaluación del módulo "Introducción a la algorítmica".

2.5.1. Algoritmo *raizEntera*

La codificación del algoritmo que calcula la raíz entera de un número entero sería la siguiente:

```
/* Programa raizEntera.c - Calcula la raíz entera de un entero
   Jordi Riera, 2001 */
#include <stdio.h>
#include <stdlib.h>

int main()
{
       /* Declaración de variables */
       int n, raiz;

       scanf("%d", &n);     /* Leemos un número entero por teclado */
       /* Pre: n=N y N>=0 */
       raiz = 0;
       while ((raiz + 1)*(raiz + 1) <= n) {
              raiz = raiz + 1;
       }
       /* Post: raiz*raiz <= N < (raiz + 1)*(raiz + 1) */
       printf("%d\n", raiz); /* Escribimos el resultado por pantalla */

       system("PAUSE");     /* El programa espera una tecla */
       return 0;
}
```

Consultad la solución al algoritmo *raizEntera*, correspondiente al ejercicio de auto evaluación número 12 del módulo "Introducción a la algorítmica" de la asignatura.

2.5.2. Algoritmo *raizEcuacionSegundoGrado*

En este apartado veremos la codificación del algoritmo que calcula las raíces reales de una ecuación de segundo grado, dados los coeficientes de la misma. Codificaremos la acción *raizEcuacion* para encontrar las raíces, mientras que en el algoritmo principal haremos la entrada y salida de datos y la invocación a la acción para resolver el problema.

```
/* Programa raizEcuacionSegundoGrado.c
   Calcula las raíces reales de una ecuación de segundo grado, dados los
    coeficientes
   Jordi Riera, 2001 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>              /* Necesaria para la función sqrt() */

/* Definición de tipo */
typedef enum {FALSE, TRUE} bool;

/* Predeclaraciones de funciones */
void raizEcuacion(float a, float b, float c, bool *tieneSolucion, float *x1,
   float *x2);

int main()
{
   /* Declaración de variables */
   float a, b, c, x1, x2;
   bool tiene;
```

Consultad las soluciones al algoritmo *raizEcuacionSegundoGrado* y a la acción *raizEcuacion*, correspondientes a los ejercicios de auto evaluación números 19 i 20, del módulo "Introducción a la algorítmica" de la asignatura.

```
    scanf("%f", &a);           /* Entrada de los coeficientes por teclado */
    scanf("%f", &b);
    scanf("%f", &c);
    /* Pre: a=A y b=B y c=C y A!=0 */
    raizEcuacion(a, b, c, &tiene, &x1, &x2);
    /* Post: ((tiene=cert) y (A*x1*x1 + B*x1 + C = 0) y
        (A*x2*x2 + B*x2 + C = 0)) o (tiene=falso) */
    if (tiene) {
           printf("%f, x1);
           printf("%f, x2);
    } else printf("N\n");

    system("PAUSE");    /* Hacemos que el aprograma espere una tecla */
    return 0;
}

void raizEcuacion(float a, float b, float c, bool *tieneSolucion, float *x1,
    float *x2)
{
    float discr;

/* Pre: a=A y b=B y c=C y A!=0 */
    discr = b*b - 4.0*a*c;
    if (discr>=0.0) {
           *x1 = (-b + sqrt(discr)) / (2.0 * a);
           *x2 = (-b - sqrt(discr)) / (2.0 * a);
           *tieneSolucion = TRUE;
    } else *tieneSolucion = FALSE;
/* Post: ((tieneSolucion=cierto) y (A*x1*x1 + B*x1 + C = 0) y
       A*x2*x2 + B*x2 + C = 0)) o (tieneSolucion=falso) */
}
```

El algoritmo *raizEcuacionSegundoGrado* nos ilustra algunas posibilidades o características del lenguaje C, que se comentan a continuación. En primer lugar, observad la inclusión del fichero *math.h*; en él se encuentran las declaraciones de algunas funciones matemáticas, entre las cuales se encuentra la de la raíz cuadrada de un número real (*sqrt*).

Se pueden declarar también constantes globales…

... pero hay que evitar el uso de variables globales, ya que rompen la metodología de diseño descendente seguida en la asignatura y hace los programas menos legibles.

Notad acto seguido la declaración del tipo *bool*. Ésta no se hace dentro del programa principal (*main*), sino al inicio del programa, fuera de cualquier acción o función. Eso permite que la definición sea global al programa y, por lo tanto, nos sirva para cualquier acción o función posterior.

Observad a continuación la predeclaración de la acción *raizEcuacion*. Esta declaración es necesaria porque si no el compilador al encontrar la invocación de la acción no sabría todavía qué tipo de parámetros formales tiene ésta. Lo mejor es que la predeclaración coincida exactamente con la posterior declaración.

Hay que hacer las predeclaraciones de todas las funciones y acciones utilizadas en el programa. Las predeclaraciones de funciones de las librerías de C se hacen dentro de los ficheros que incluimos con **#include**.

Hay que destacar también que la acción *raizEcuacion* tiene tres parámetros de salida. Observad como en la declaración de los parámetros formales hay que preceder el nombre con un "*", y también hay que hacerlo en el uso de estos parámetros en el cuerpo de la acción. De manera similar, en la invocación de la acción los parámetros actuales tienen que ir precedidos por un "&".

Finalmente, notad que a la hora de escribir el caracter "N" por pantalla no se ha utilizado la acción *printf("%c ", 'N')* sino *printf("N\n"*). Se ha usado una variación del *printf* que permite escribir directamente una cadena de caracteres por pantalla, colocando como parámetro del *printf* los caracteres a escribir (entre comillas). En este caso, se escribirá el caracter 'N' y el caracter '\n', que se corresponde con un salto de línea.

La acción printf("*cadena_caracteres*");

... permite sacar fácilmente por pantalla mensajes completos que indiquen al usuario del programa lo que éste está haciendo o esperando.

3. Tratamiento secuencial en lenguaje C

En el módulo "Tratamiento secuencial" de la asignatura se introduce una metodología sistemática para plantear soluciones a problemas que se pueden modelar con una secuencia de elementos: son los esquemas de búsqueda y recorrido. Esta metodología no introduce nuevos elementos del lenguaje algorítmico y, por lo tanto, los algoritmos que lo utilizan se pueden codificar en lenguaje C con lo que se ha visto hasta ahora en este módulo.

Tened en cuenta que...

... se verán algoritmos que tratan secuencias generadas u obtenidas de la entrada estándar (teclado). El lenguaje C dispone de funciones para tratar directamente secuencias guardadas en ficheros, pero el estudio de éstas queda fuera del alcance de la asignatura.

En este apartado se mostrará, a modo de ejemplo, la codificación en lenguaje C de algunos de los algoritmos vistos en el módulo "Tratamiento secuencial" de los materiales.

3.1. Ejemplo de esquema de recorrido de una secuencia en C

Como ejemplo de esquema de recorrido en una secuencia, codificaremos en C el algoritmo *sumaDigitos*, que escribe en pantalla la suma de los dígitos de un número previamente obtenido por el teclado. Recordad que la secuencia de dígitos se genera en el propio algoritmo.

Consultad cómo se obtiene la solución al algoritmo *sumaDigitos* en el apartado 2.5.2. "Suma de las cifras de un número", del módulo "Tratamiento secuencial" de la asignatura.

```
/* Programa sumaDigitos.c
   Escribe en pantalla la suma de los dígitos de un número introducido por el teclado
   Jordi Riera, 2001
*/
#include <stdio.h>
int main()
{
   /* Declaración de variables */
   int n, suma, d;
   scanf("%d", &n);
   /* Pre: n=N y N>=0 */
   d = n % 10;       /* preparar secuencia */
   suma = 0;         /* inicio tratamiento */
   while (!(n==0)) {            /* fin secuencia */
         suma = suma + d;       /* tratar elemento */
         n = n / 10;            /* avanzar secuencia */
         d = n % 10;
   }
   printf("%d\n", suma);   /* tratamiento final */
   /* Post: La suma de los dígitos de N se ha escrito en pantalla*/

   system("PAUSE");  /* Hacemos que el programa espere una tecla */
   return 0;
}
```

3.2. Ejemplo de esquema de búsqueda en una secuencia en C

Como ejemplo de esquema de búsqueda en una secuencia, codificaremos en C el algoritmo *buscarFibonacci9*, que escribe en pantalla uno de los mil primeros números de Fibonacci que además acaba en 9, si es que existe; si no existe se escribe un -1. Recordad que la secuencia de números de Fibonacci se genera en el propio algoritmo.

Consultad cómo se obtiene la solución al algoritmo *buscarFibonacci9* en el apartado 3.5.1. "Números de Fibonacci", del módulo "Tratamiento secuencial" de la asignatura.

```
/* Programa buscarFibonacci9.c
   Escribe en pantalla uno de los mil primeros números de
   Fibonacci, acabado en 9, si es que existe; si no existe, escribe
   un -1
 Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de tipos */
typedef enum {FALSE, TRUE} bool;

int main()
{
   /* Declaración de variables */
   int actual, siguiente, siguiente2;
   int i;
   bool encontrado;
   /* Pre: Cierto */
   actual = 0;       /* preparar secuencia */
   siguiente = 1;
   i = 1;
   encontrado = FALSE;
   /* inicio tratamiento - está vacío */
   while ((i<=1000) && !encontrado) {  /* fin secuencia - i>1000 */
        encontrado = (actual % 10) == 9; /* actualizar encontrado */
      if (!encontrado) {   /* tratar elemento - está vacío */
         siguiente2 = actual + siguiente;/* avanzar secuencia */
         actual = siguiente;
         siguiente = siguiente2;
         i = i + 1;
      }
   }
   if (encontrado) { /* tratamiento final */
      printf("%d\n", actual);
   } else {
      printf("%d\n", -1); /*Notad: printf("-1\n") sería más simple*/
   }
   /* Post: En pantalla se ha escrito uno de los mil primeros
         números de Fibonacci, acabado en 9, si es que existe; si
         no existe, se ha escrito un -1 */

   system("PAUSE"); /* Hacemos que el programa espere una tecla */
   return 0;
}
```

3.3. Ejemplo de esquema de recorrido aplicado a la entrada en C

Como ejemplo de esquema de recorrido aplicado a la entrada, codificaremos en C el algoritmo *cuentaLetraA*, que escribe en el dispositivo de salida cuántas veces aparece la letra "a" en una frase acabada con un punto que se introduce por la entrada.

Consultad cómo se obtiene la solución al algoritmo *cuentaLetraA* en el apartado 4.1.3. "Ejemplo", del módulo "Tratamiento secuencial" de la asignatura.

```
/* Programa cuentaLetraA.c
   Calcula el número de veces que se encuentra la letra "a" en la
   entrada
   Jordi Riera, 2001
*/
#include <stdio.h>
int main()
{
   /* Declaración de variables */
   char car;
   int n;
   /* Pre: En la entrada se lee una secuencia de caracteres que no
         tiene ningún punto, seguida de un punto */
   scanf("%c", &car);      /* preparar secuencia - leer() */
   n = 0;                  /* inicio tratamiento */
   while (! (car=='.')) {  /* fin secuencia - marca = '.' */
      if (car=='a') n = n + 1; /* tratar elemento */
         scanf("%c", &car); /* avanzar secuencia - leer() */
   }
   printf("%d\n", n); /* tratamiento final */
   /* Post: Se ha escrito en pantalla el número de veces que la
         letra "a" está en la entrada */
   system("PAUSE"); /* Hacemos que el programa espere una tecla */
```

```
   return 0;
}
```

3.4. Ejemplo de esquema de búsqueda aplicado a la entrada en C

Como ejemplo de esquema de busca aplicado a la entrada, codificaremos en C el algoritmo *saleLetraA*, que escribe en el dispositivo de salida "S" si en la entrada hay alguna letra "a" y escribe "N" en caso contrario.

Consultad cómo se obtiene la solución al algoritmo *saleLetraA* en el apartado 4.2.3. "Ejemplo", del módulo "Tratamiento secuencial" de la asignatura.

```
/* Programa saleLetraA.c
   Lee una secuencia de caracteres que no tiene ningún punto,
   seguida de un punto. Escribe "S" si en la entrada había alguna
   "a" y escribe "N" en caso contrario.
   Jordi Riera, 2001
*/
#include <stdio.h>
typedef enum {FALSE, TRUE} bool;       /* Definición de tipos */

int main()
{
   char car;   /* Declaración de variables */
   bool encontrado;
   /* Pre: En la entrada hay una secuencia de caracteres que no
         tiene ningún punto, seguida de un punto */
   scanf("%c", &car);      /* preparar secuencia - leer() */
   encontrado = FALSE;

   /* inicio tratamiento - está vacío */
   while (!(car=='.') && !encontrado) {/* fin secuencia – marca='.'
   */
      encontrado = car == 'a';   /* actualizar encontrado */
      if (!encontrado) {
               /* tratar elemento - está vacío */
         scanf("%c", &car);      /* avanzar secuencia - leer() */
      }
   }
   if (encontrado) printf("S\n");      /* tratamiento final */
   else printf("N\n");
   /* Post: Por el dispositivo de salida se escribe "S" si en la
         entrada hay alguna letra "a", y "N" en caso contrario */

   system("PAUSE"); /* Hacemos que el programa espere una tecla */
   return 0;
}
```

3.5. Combinación de esquemas en C

Finalizaremos este apartado sobre el tratamiento secuencial codificando en C un algoritmo que combine un esquema de busca con uno de recorrido. Se ha escogido como ejemplo el algoritmo *mediaSuspensos* porque en él se trata con una secuencia de números reales leída del dispositivo de entrada estándar (hasta ahora se habían tratado sólo secuencias de caracteres).

Consultad cómo se obtiene la solución al algoritmo *mediaSuspensos* en el apartado 5.2.1. "Media de los suspensos", del módulo "Tratamiento secuencial" de la asignatura.

El algoritmo *mediaSuspensos* lee del dispositivo de entrada una secuencia de reales entre 0.0 y 10.0 ordenada de forma descendente, seguida por un -1, y escribe en el dispositivo de salida la media de los números entre 0.0 y 5.0, o bien 0.0 si no había ningún número que cumpliera esta condición.

```c
/* Programa mediaSuspensos.c
   calcula la media de los suspensos en una serie descendente de
   números reales
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de tipos */
typedef enum {FALSE, TRUE} bool;

int main() {
   /* Declaración de variables */
   float x, sum, result;
   int n;
   bool encontrado;

   /* Pre: En la entrada se lee una secuencia de reales entre 0.0 y
         10.0 ordenada de forma descendente, seguida por un -1.0 */
   /* Esquema de búsqueda */
   scanf("%f", &x);  /* preparar secuencia - leer() */
   encontrado = FALSE;
   /* inicio tratamiento - está vacío */
   while (!(x==-1.0) && !encontrado) { /* fin secuencia - marca=
   -1.0 */
      encontrado = x<5.0;  /* actualizar encontrado */
      if (!encontrado) {
               /* tratar elemento - está vacío */
         scanf("%f", &x);  /* avanzar secuencia - leer() */
      }
   }
   /* tratamiento final - es el esquema de recorrido */
   /* Esquema de recorrido sobre el resto de la secuencia */
   /* preparar secuencia - ya está a punto */
   n = 0.0; /* inicio tratamiento */
   sum = 0.0;
   while (!(x==-1.0)) { /* fin secuencia - marca=-1.0 */
      sum = sum + x;       /* tratar elemento */
      n = n + 1;
      scanf("%f", &x);     /* avanzar secuencia - leer() */
   }
   if (n>0) result = sum/(float) n;    /* tratamiento final */
   else n = 0.0;
   printf("%f\n", result);
   /* Post: En la salida se ha escrito la media aritmética de los
         números x tal que 0.0 <= x < 5.0, o bien 0.0 si no había
         ningún número que cumpliera la condición */

   system("PAUSE"); /* Hacemos que el programa espere una tecla */
   return 0;
}
```

Notad que los números reales de la secuencia de entrada se pueden entrar separándolos por espacios o saltos de línea, gracias al uso de la acción *scanf*.

4. Tipos estructurados de datos en lenguaje C

En este apartado se mostrará cómo se codifican en lenguaje C los nuevos tipos de datos introducidos en el módulo "Tipos estructurados de datos" de la asignatura. Se estudiará la codificación en lenguaje C de tablas y tuplas, así como las particularidades respecto de la traducción directa desde el lenguaje algorítmico.

4.1. Tablas en C

Igual que en el lenguaje algorítmico, las tablas son un tipo de datos estructurado, homogéneo y de acceso directo. A continuación se verá su declaración y como utilizarlas en lenguaje C.

4.1.1. Declaración de tablas, matrices y tablas de tablas en C

La declaración de una variable tipo tabla en C obedece la siguiente sintaxis:

tipo nombre[tamaño];

Consultad la sintaxis de declaración de una tabla y de una tabla de tablas en lenguaje algorítmico en los apartados 2.1. "Declaración" y 2.3. "Otros tipos de tablas", del módulo "Tratamiento secuencial" de la asignatura.

La sintaxis para declarar tablas de tablas es:

tipo nombre[tamaño$_1$][tamaño$_2$] ...[tamaño$_n$];

En C...

"tipo nombre[tamaño$_1$, tamaño$_2$.., tamaño$_n$]; es una declaración válida equivalente a *"tipo nombre*[tamaño$_1$];. El operador "," permite evaluar diversas expresiones y devuelve el resultado de la primera de ellas.

El lenguaje C no permite declarar matrices de más de una dimensión usando la sintaxis del lenguaje algorítmico (separando los índice por ","). En su lugar, hay que usar la sintaxis anterior, encerrando los índices entre corchetes ("*[indice]*"). La funcionalidad, sin embargo, es la misma.

También podemos declarar nuevos tipos con los constructores:

typedef *tipo nombre[tamaño];*
typedef *tipo nombre[tamaño$_1$][tamaño$_2$] ...[tamaño$_n$];*

4.1.2. Funciones elementales de acceso

Tal y como sucede en el lenguaje algorítmico, las funciones elementales de acceso a las tablas, matrices y tablas de tablas en lenguaje C nos permiten

acceder a uno solo de sus elementos para consultarlo o asignarle un valor. La sintaxis es la misma que la vista en el módulo "Tipos estructurados de datos" de la asignatura.

Hay, sin embargo, una diferencia en la numeración de los elementos de una tabla en el lenguaje algorítmico y los de una tabla codificada en C. Así, en el lenguaje algorítmico el primer elemento es el correspondiente al índice 1; en cambio, en lenguaje C:

el índice 0 identifica el primer elemento de la tabla.

Esta diferencia es pequeña, pero muy importante, y hay que tenerla bien presente en la codificación de los algoritmos a lenguaje C. Habitualmente no comportará más quebraderos de cabeza que retroceder una posición en los índices de acceso y decrementar en una unidad los límites de los procesos iterativos.

Como ejemplo de los cambios que habrá que hacer, codificaremos en C el algoritmo *frecLetras*, que escribe en el dispositivo de salida la frecuencia de aparición de los caracteres 'A' hasta 'Z' en un texto acabado en un punto introducido por el dispositivo de entrada.

Consultad el algoritmo *frecLetras* en el apartado 2.2.2. "Lectura y escritura", del módulo "Tipos estructurados de datos" de la asignatura.

```
/* Programa frecLetras.c
   Frecuencia de aparición de los caracteres "A" hasta "Z"
   Jordi Riera, 2001
*/
#include <stdio.h>

int main()
{
   /* Declaración de variables */
   float frec[26];
   int ndc, i, codigoBase;
   char c;

   /* Pre: Por el canal de entrada leeremos una secuencia de
         caracteres en mayúsculas, de como mínimo uno, acabada en
         un punto */
   scanf("%c", &c);        /* preparar secuencia - leer() */
   ndc = 0;                /* inicio tratamiento */
   for (i=0; i<26; i++) {  /* Notad que el índice i va de 0 a 25 */
      frec[i] = 0;
   }
   codigoBase = (int) 'A';
   while (c!='.') { /* fin secuencia - marca = '.' */
      i = (int) c - codigoBase; /* tratar elemento - no hay el +1 */
      frec[i] = frec[i] + 1.0;
      ndc = ndc + 1;
      scanf("%c", &c);     /* avanzar secuencia - leer() */
   }
   for (i=0; i<26; i++) { /* tratamiento final - Notad i entre 0
   y 25*/
      frec[i] = frec[i] / (float) ndc;
      printf("%c", (char) codigoBase + i); /* Notad que no hay
                                              el -1 */
      printf(":");
      printf("%f\n", frec[i]);
   }
   /* Post: Se ha escrito por el canal de salida la frecuencia de
      aparición de los caracteres "A" hasta "Z" del texto
      introducido por el canal de entrada */

   system("PAUSE"); /* Hacemos que el programa espere una tecla */
```

```
   return 0;
}
```

Hay que destacar que el índice *i* va de 0 a 25 al hacer el recorrido de la tabla, en lugar de ir de 1 a 26 como en el lenguaje algorítmico. Por lo tanto, en el **for** se ha modificado la inicialización de *i*, a 0 en vez de a 1, y la comparación con la posición final, que utiliza el operador "<" en ningún vez del operador "<=". Por el mismo motivo, al calcular la posición de la tabla que hay que incrementar, se ha eliminado el "+1", de tal manera que al carácter "a" le corresponde la posición 0 en vez de la 1, y así sucesivamente.

4.1.3. Acceso secuencial a una tabla

El acceso secuencial a las tablas permite aplicar los esquemas de recorrido y búsqueda vistos en el apartado 2.4. "Acceso secuencial a una tabla" del módulo "Tipos estructurados de datos" de la asignatura. La codificación de estos esquemas en lenguaje C es directo. Sólo hace falta tener presente la diferente numeración en los índices de las tablas y hacer la inicialización a 0 en vez de a 1 como en el lenguaje algorítmico. También hay que retocar el límite del recorrido o de la búsqueda en el caso de secuencias no marcadas, que requerirá un índice una unidad menor en C que en lenguaje algorítmico.

Como ejemplo, codificaremos el algoritmo *infMedia,* que a partir de una secuencia de enteros con las calificaciones de como mucho 50 estudiantes calcula la nota media y el número de estudiantes que tienen una nota inferior a ésta. Codificaremos la variante que usa una variable entera para indicar cuántas posiciones significativas hay en la tabla.

Consultad el algoritmo *calificaciones* en el apartado 2.4.1. “Esquema de recorrido aplicado a tablas” (página 22), del módulo “Tipos estructurados de datos” de la asignatura.

```
/* Programa infMedia.c
   Calcula la nota media y el número de estudiantes que tienen una
   nota inferior
   Jordi Riera, 2001
*/
#include <stdio.h>

int main()
{
   /* Declaración de constantes */
   const int M = 50;

   /* Declaración de variables */
   int t[M];
   int cuantos; /* Indicará cuántos elementos hay en la tabla */
   int i, n;
   int suma, infMedia;
   float media;

   /* Pre: Por el canal de entrada se lee una secuencia no vacía
      de como mucho 50 enteros entre 0 y 10, seguida de -1 */
   /* Recorrido sobre la secuencia de entrada */
   scanf("%d", &n);        /* preparar secuencia - leer() */
   cuantos = 0;            /* inicio tratamiento */
   suma = 0;
   while (!(n==-1)) {      /* fin secuencia - marca = -1 */
      t[cuantos] = n; /* tratar elemento */
      suma = suma + n; /* acceder a n es más eficiente que acceder
      a t[cuantos] */
      cuantos = cuantos + 1; /* cuantos se actualiza después de
                                tratar el elemento actual */
      scanf("%d", &n); /* avanzar secuencia - leer() */
   }
   media = (float) suma / (float) cuantos;  /* tratamiento final */
```

```
   /* Recorrido sobre la tabla */
   /* preparar secuencia - i=0 en el for */
   infMedia = 0;     /* inicio tratamiento */
   for (i=0; i<cuantos; i++) { /* fin secuencia - límite=cuantos */
      if ((float) t[i] < media) /* tratar elemento */
         infMedia = infMedia + 1 ;
                        /* avanzar secuencia - i++ en el for */
   }
   printf("%f ", media); /* tratamiento final */
   printf("%d\n", infMedia);
   /* Post: Se ha escrito por el canal de salida la media de los
      elementos de la secuencia y el número de elementos con
      un valor inferior a ésta */

   system("PAUSE"); /* Hacemos que el programa espere una tecla */
   return 0;
}
```

Notad como, a diferencia del lenguaje algorítmico, la variable *cuantos* se actualiza después de haber guardado la calificación en la tabla, y no antes. Así, la primera calificación se guarda en el índice 0, la segunda el índice 1, y la última en el índice *cuantos*-1.

Se puede observar que la utilización de una instrucción **for** hace que diversas partes del recorrido de la tabla se hagan en esta sentencia. En concreto, se inicializa la variable con la que se recorrerá la tabla (*i=0*), se incrementa este índice para avanzar en la tabla (*i*++) y se comprueba si se ha llegado al final del recorrido (*i<cuantos*).

4.1.4. Acceso directo a una tabla

El acceso directo es una de las características más importantes de las tablas. Aunque este tipo de acceso no incorpora ningún elemento nuevo del lenguaje C, en este apartado se mostrará un algoritmo que aprovecha las ventajas. Se trata del algoritmo *busquedaDicotomica*, que dada una tabla ordenada en forma creciente, responde si un entero dado está o no en la tabla, haciendo una búsqueda que descarta en cada paso la mitad de los elementos que quedan en la tabla.

Consultad el algoritmo *busquedaDicotomica* en el apartado 2.5.1. "Importancia de los datos ordenados. Búsqueda dicotómica", del módulo "Tipos estructurados de datos" de la asignatura.

```
/* Programa busquedaDicotomica.c
   Comprueba si un entero está en una tabla ordenada, realizando
   una búsqueda dicotómica
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de constantes conocidas en tiempo de compilación */
#define N 18

int main()
{
   /* Declaración de variables */
   int t[N] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20,
   27, 30, 50};
   int dado;
   int inicio, fin, medio;

   /* Pre: Por el canal de entrada se lee un entero */
   scanf("%d", &dado);
   inicio = 1; /* Definimos los límites iniciales de la tabla */
   fin = N-1;
   while (inicio != fin) { /* La búsqueda acaba cuando los límites
                              coinciden */
      medio = (inicio + fin) / 2; /* Posición del elemento central
                                     de la tabla */
      if (t[medio] < dado) { /* Establecemos los nuevos límites
                                de la tabla */
```

```
            inicio = medio + 1;
         } else {
            fin = medio;
         }
      }
      if (t[inicio]==dado) printf("S\n"); /* Comprobar si se ha
                                          encontrado el elemento... */
      else printf("N\n");
      /* Post: Se escribe "S" por el canal de salida si el entero está
         en la tabla y "N" en caso contrario */

      system("PAUSE"); /* Hacemos que el programa espere una tecla */
      return 0;
   }
```

Recordad que la directiva #define...

... se tiene que escribir en la primera posición de una línea, ya que si no el compilador no la reconocería. Todas las directivas empiezan siempre por "#".

En primer lugar, notad la declaración de la constante *N* con la directiva **#define**. Observamos que no se indica el tipo de la constante que se está definiendo. De hecho, el compilador únicamente sustituye el nombre, N, por su valor, *18*, cada vez que lo encuentra en el código posterior a la declaración. La definición de esta constante no se puede hacer con *const int N=1*8, ya que el valor *de* N es asignado en tiempo de ejecución del programa y, en cambio, el compilador lo necesita en tiempo de compilación para poder saber la medida de la tabla *t* y hacer la inicialización correspondiente. En el algoritmo *infMedia*, por el contrario, la declaración de la constante *M* sí que se puede hacer con *const int M=5*0, porque su valor se utiliza para declarar una tabla que no hay que inicializar. En caso de duda, siempre es mejor usar la directiva **#define** para declarar constantes.

La inicialización de la tabla *t*, precisamente, corresponde a una construcción del lenguaje C que no habíamos visto todavía. La tabla *t* se inicializa en tiempo de compilación. Fijaos cómo asignamos un valor por cada posición de la tabla, separando los valores por comas, dentro de un bloque con llaves.

En el apartado "Construcciones avanzadas del lenguaje C" de este mismo módulo podéis encontrar más información sobre la inicialización de tablas en tiempo de ejecución.

4.1.5. Tablas como parámetros de acciones y funciones

En el lenguaje algorítmico, las tablas se pueden pasar como parámetros de acciones y funciones, o incluso ser devueltas por una función, sin ninguna restricción. En lenguaje C, en cambio, sólo se permite pasar una tabla referencia. Es decir, siempre se pasa la tabla original, y los cambios hechos en la tabla por la acción o función se reflejarán en ella. Asimismo, tampoco se puede devolver una tabla entera en una función.

La sintaxis para la declaración de los parámetros en la cabecera de una acción o función cuando *tipo* es una tabla debe ser:

para un parámetro de entrada:	**const** *tipo nombre*
para un parámetro de salida:	*tipo nombre*
para un parámetro de entrada/salida:	*tipo nombre*

Fijaos que al declarar una tabla como parámetro de entrada, añadimos la palabra reservada **const** a la declaración. De esta manera indicamos al compilador que no haremos cambios en el contenido de la tabla.

El lenguaje C...

... no permite trabajar con tablas como parámetros de entrada, modificando su valor sin que se vea afectada la tabla original en la llamada.

Notad también que al declarar una tabla como parámetro de entrada/salida o de salida, no hemos de preceder el nombre de la tabla con "*", como hacíamos con el resto de tipos. Tampoco usaremos "*" ante el nombre en el cuerpo de la acción o función, ni "&" en la invocación.
Eso es debido a que el nombre de la tabla es ya por si una dirección, más concretamente es la dirección del primer elemento de la tabla, y los corchetes ("[*índice*]") nos servirán para acceder a los diferentes elementos que contiene.

Como ejemplo del paso de tablas como parámetros, codificaremos el algoritmo *cuadrados*, que calcula los cuadrados de los elementos de una tabla, usando llamadas a acciones y funciones y pasando tablas como parámetros.

Consultad el algoritmo *cuadrados* en el apartado 2.2.2. "Lectura y escritura", del módulo "Tipos estructurados de datos" de la asignatura.

```
/* Programa cuadrados.c
   Calcula los cuadrados de una tabla de enteros
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de constantes en tiempo de compilación */
#define M 10

/* Definición de tipos */
typedef int t[M];          /* Declaración del tipo t como tabla de
                                enteros */
/* Predeclaraciones de funciones */
void rellenarTabla(t p);
void escribirTabla(const t p);
void cuadradosTabla(const t p, t nuevap);

int main()
{
   /* Declaración de variables */
   t tablaI, tablaF;
   /* Pre: Se leerá por el dispositivo de entrada una secuencia de
      M enteros */
   rellenarTabla(tablaI);
   cuadradosTabla(tablaI, tablaF);
   escribirTabla(tablaF);
   /* Post: Se ha escrito por el dispositivo de salida el cuadrado
      de cada elemento de la secuencia leída, en el mismo orden */
      system("PAUSE"); /* Hacemos que el programa espere una tecla*/
   return 0;
}

void rellenarTabla(t p) /* p es un parámetro de entrada/salida */
{
   /* Declaración de variables */
   int i;
   /* Pre: Se leerá por el dispositivo de entrada una secuencia de
         M enteros */
   for(i=0; i<M; i++) scanf("%d", &p[i]);
   /* Post: La tabla p contiene la secuencia leída, en el mismo
         orden */
}

void escribirTabla(const t p) /* p es un parámetro de entrada */
{
   /* Declaración de variables */
   int i;
   /* Pre: La tabla p contiene M enteros */
   for(i=0; i<M; i++) printf("%d ", p[i]);
   /* Post: Se ha escrito en el dispositivo de salida el contenido
         de la tabla p, en el mismo orden */
```

```
}

void cuadradosTabla(const t p, t nuevap) /* p es de entrada, nuevap
   de salida */
{
   /* Declaración de variables */ int i;
   /* Pre: La tabla p contiene M enteros */
   for(i=0; i<M; i++) {
         nuevap[i] = p[i] * p[i];
   }
   /* Post: La tabla nuevap contiene los cuadrados de los elementos
         de p, en el mismo orden */
}
```

La constante *M* y el tipo *t* se declaran fuera de cualquier función, a fin de que estén disponibles en todo el programa.

Fijaos en la declaración de la función *cuadradosTabla*. El parámetro de entrada *p* va precedido por la palabra reservada **const**, indicando que no haremos cambios en la tabla original. Además, se ha añadido un segundo parámetro con el fin de poder devolver una tabla con los cuadrados. Este segundo parámetro tiene el mismo tipo y nombre, *nuevap*, que la variable que devuelve el algoritmo *cuadrados* en lenguaje algorítmico. Eso no es ninguna coincidencia. Como el lenguaje C no nos permite devolver una tabla en una función, añadiremos un nuevo parámetro de salida para hacerlo y, para más claridad, usaremos el mismo nombre que la variable que devolvemos en lenguaje algorítmico.

4.2. Tuplas en C

Igual que en el lenguaje algorítmico, las tuplas son un tipo de datos estructurado, heterogéneo y de acceso directo. A continuación se verá su declaración y cómo utilizarlas en lenguaje C.

4.2.1. Declaración de tuplas en C

La declaración de una variable tipo tupla en C obedece la siguiente sintaxis:

struct {
   tipo1 $campo_1$*;*
   tipo2 $campo_2$*;*
   ...
} *nombre;*

En ella se declara una variable tipo tupla, de nombre *nombre*, y campos $campo_1$, $campo_2$ … Se utiliza una sintaxis muy similar en el constructor de tipo:

typedef struct {
   tipo1 $campo_1$*;*

tipo2 campo$_2$;

...

} *nombre;*

4.2.2. Funciones elementales de acceso

Las funciones elementales de acceso en tuplas y campos de tuplas son equivalentes a las vistas en el lenguaje algorítmico. La sintaxis es la misma que la vista en el módulo "Tipos estructurados de datos" de la asignatura.

Como ejemplo, codificaremos en C el algoritmo *puntoMedio*, que escribe por el dispositivo de salida el punto medio entre dos puntos entrados por el dispositivo de entrada.

Consultad el algoritmo *puntoMedio* en el apartado 3.2.2. "Lectura y escritura por los canales de entrada/salida estándar", del módulo "Tipos estructurados de datos" de la asignatura.

```
/* Programa puntoMedio.c
   Calcula el punto medio entre dos puntos dados por el dispositivo
   de entrada
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Declaración de tipos */
typedef struct {
   float x, y;
} coordenadas;

/* Predeclaración de funciones */
coordenadas obtenerPunto();
void escribirPunto(coordenadas p);
coordenadas calculoPuntoMedio(coordenadas p1, coordenadas p2);

int main()
{
   /* Declaración de variables */
   coordenadas a, b;

   /* Pre: Se leen cuatro reales del canal estándar X1, Y1, X2, Y2
      que representan las coordenadas de dos puntos A y B,
      respectivamente */
   a = obtenerPunto();
   b = obtenerPunto();
   escribirPunto(calculoPuntoMedio(a, b));
   /* Post: Se escriben las coordenadas del punto medio entre A y
      B, que son (X1+X2)/2, (Y1+Y2)/2 */

   system("PAUSE"); /* Hacemos que el programa espere una tecla */
   return 0;
}

coordenadas obtenerPunto()
{
   /* Declaración de variables */
   coordenadas p;
   /* Pre: Se leen del canal estándar dos reales X e Y */
   scanf("%f", &p.x);
   scanf("%f", &p.y);
   return p;
   /* Post: p.x = X y p.y = Y */
}

void escribirPunto(coordenadas p)
{
   /* Pre: p.x = X y p.y = Y */
   printf("%f", p.x);
   printf(",");
   printf("%f ", p.y);
   /* Post: Se ha escrito en el canal estándar "X,Y " */
```

```
}

coordenadas calculoPuntoMedio(coordenadas p1, coordenadas p2)
{
   /* Declaración de variables */
   coordenadas p;
   /* Pre: p1.x = X1 y p1.y = Y1 y p2.x = X2 y p2.y = Y2 */
   p.x = (p1.x + p2.x) / 2.0;
   p.y = (p1.y + p2.y) / 2.0;
   return p;
   /* Post: p.x = (X1+X2)/2 y p.y = (Y1+Y2)/2 */
}
```

Retornar una tupla en C...

... o pasarla por valor a una función puede ser poco eficiente. Hay que pensar que una tupla puede contener diversos campos (algunos pueden ser incluso otras tuplas y/o tablas). Por lo tanto, pasar tuplas por valor o devolverlas en una función implica copiar una gran cantidad de información. Lo más eficiente es pasar las tuplas por referencia.

Es importante destacar que, a diferencia de lo que pasaba con las tablas, las tuplas se pueden pasar como parámetro, tanto por valor como por referencia, e incluso pueden ser devueltas por una función. Es decir, que siguen la sintaxis de los tipos elementales del lenguaje C, y que no se aplican las restricciones y consideraciones que hemos visto para las tablas.

Para acceder a un campo de una tupla pasada por referencia, sin embargo, habrá que poner un paréntesis en torno al nombre de la tupla, antes del operador de acceso al campo ".". Eso es debido a la orden de precedencia de los operadores involucrados. La sintaxis es:

(*nombre).campo

El lenguaje C dispone también de un operador que ahorra el uso del paréntesis al acceder a un campo de una tupla pasada por referencia. Es totalmente equivalente al anterior, y su sintaxis, más simple, queda como sigue:

nombre->campo

4.2.3. Tablas y tuplas

Evidentemente, también podemos tener en C tablas de tuplas.

El lenguaje C, igual que el algorítmico, permite que los campos de una tupla puedan ser de cualquier tipo. En concreto, pueden ser también tipos estructurados de datos, en concreto, otras tuplas o tablas.

Así, el tipo *palabra* visto en el apartado del mismo nombre, en el módulo "Tipos estructurados de datos" de la asignatura se definiría como:

```
typedef struct {
   char letras[40];
```

```
int l;
} palabra;
```

Notad que en lenguaje C, guardar en *palabra.l* el número de caracteres de la *palabra*, o guardar el índice del primer carácter vacío de *palabra.letras* es exactamente lo mismo, ya que los índice de las tablas en C empiezan en la posición 0.

Finalmente, con el fin de ver un ejemplo complejo de combinación de tablas y tuplas, se codificará en C la función *costeProducto*, que dada una estructura de almacén y composición del producto nos calcula su precio de coste. Notad las declaraciones de tipo, donde tenemos tablas de tuplas, tuplas que contienen otras tuplas, etc.

Consultad el ejercicio de auto evaluación número 9, del módulo "Tipos estructurados de datos" de la asignatura para a una especificación detallada de los tipos de datos involucrados y de la manera de calcular el precio de coste de un producto.

```
/* Declaraciones de constantes en tiempo de compilación */
#define MAXL 40

/* Declaración de tipos */
typedef struct {     /* tipo palabra */
   char letras[MAXL]; /* la tupla contiene una tabla de
                         caracteres */
   int l;
} palabra;

typedef struct {     /* tipo materia */
   palabra referencia; /* la tupla materia contiene una tupla
                          palabra */
   int precioCoste, stock;
} materia;

typedef struct { /* tipo producto */
   palabra nombre; /* la tupla producto contiene una tupla
                      palabra */
   int costeFabricacion;
   struct { /* la tupla producto contiene la tupla
               composicion */
         struct { /* elemento de la tabla materias (tupla) */
               int codigoMat;
               int cantidad;
         } materias[10]; /* la tupla composicion tiene la tabla
                            materias */
         int num;
   } composicion;
} producto;

typedef struct { /* tipo almacen */
   materia materias[200]; /* la tupla almacen contiene una tabla*/
   int numMat;
} almacen;

typedef producto productos[15];  /* tipo productos */
                                 /* tabla de tuplas producto */

int costeProducto(producto p, almacen m)
{
   /* Declaración de variables */
   int c, j;
   /* Pre: p=P y m=M */
   c = p.costeFabricacion;
   for(j=0; j<p.composicion.num; j++) {
      c = c +
      (m.materias[p.composicion.materias[j].codigoMat].precioCoste
      * p.composicion.materias[j].cantidad);
   }
   return c;
   /* Post: Retorna el coste calculado como P.costeFabricacion, más
         la suma del coste de las P.composicion.num materias de
```

```
            P.composicion.materias multiplicadas por la cantidad
            de cada una de éstas */
    }
```

Observad que el programa no está completo, ya que falta la función *main*. En ella se debería inicializar la tabla *productos* y el *almacén*, hacer la llamada a la función *costeProducto* (para el producto que nos interese) y escribir el resultado por el dispositivo de salida.

5. Introducción a la metodología de diseño descendente en C

En el módulo "Introducción a la metodología de diseño descendente" de la asignatura se ha puesto de manifiesto que para poder abordar problemas complejos, del mundo real, hay que aplicar una metodología concreta, sistemática, consistente en la abstracción de datos y/o código y en el planteamiento y resolución de subproblemas cada vez más sencillos.

La metodología de diseño descendente no añade nuevos elementos al lenguaje algorítmico, sino que muestra una manera más efectiva de usar los elementos disponibles. Por lo tanto, la codificación en lenguaje C de algoritmos resueltos en base a la metodología del diseño descendente no implica, en realidad, ningún cambio en la forma de trabajar que venimos aplicando, consistente en traducir directamente las construcciones del lenguaje algorítmico a los suyos equivalentes en C.

En realidad, todo el trabajo está en el diseño del algoritmo, aplicando diseño descendente y la experiencia adquirida hasta el momento, mientras que la codificación en lenguaje C resta como un trabajo puramente mecánico.

5.1. Ejemplo: algoritmo *analisisCorrecciones*

Como ejemplo de codificación en lenguaje C de un algoritmo resuelto aplicando diseño descendente, se muestra a continuación la codificación completa del algoritmo *analisisCorrecciones*, que sirve para analizar los errores de un texto presentado por un autor de materiales.

Consultad el apartado 1.1. "Análisis de problemas complejos", del módulo "Introducción a la metodología del diseño descendente" de la asignatura para una especificación detallada del problema a resolver con el algoritmo *analisisCorrecciones*. La aplicación de la metodología del diseño descendente al problema se hace a lo largo del resto del módulo.

```
/* Programa analisisCorrecciones.c
   Analiza el porcentaje de frases, la media de palabras por frase
   y el máximo de palabras por frase que se deben corregir en un
   texto. Si no hay que corregir nada, no dice nada.
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Declaración de tipos */
typedef enum {FALSE, TRUE} bool;

typedef struct {
   bool final;
   bool vacío;
   bool sedebeCorregir;
} tPalabra; /* Definido en el 3.er nivel */

typedef struct {
   int nPalabrasC;
   bool final;
} tFrase; /* Definido en el 2.º nivel */

typedef struct {
   int nFrases;
   int nFrasesCor;
   int nPalabrasParaCorregir;
   int maximoPalabrasParaCorregir;
} tDatosAnalisis; /* Definido en el 1.er nivel */
```

```
/* Predeclaraciones de funciones */
void obtenerFrase(tFrase f);/* Funciones definidas en el 1.er nivel
   */
bool ultimaFrase(tFrase f);
void inicioResultadosAnalisis(tDatosAnalisis *a);
void actualizaResultadosAnalisis(tFrase f, tDatosAnalisis *a);
void escribirResultadosAnalisis(tDatosAnalisis a);
int palabrasParaCorregir(tFrase f);/* Definida en el 2.º nivel -
   asociada a tFrase*/
void obtenerPalabra(tPalabra m);/* Definidas en el 2.º nivel */
bool ultimaPalabra(tPalabra m);
bool palabraVacia(tPalabra m);
bool sedebeCorregirPalabra(tPalabra m);
   /* 1.er nivel de diseño descendente. Recorrido sobre una
      secuencia de frases. Abstracción de los tipos tFrase y
      tDatosAnalisis. */

int main()
{
   /* Declaración de variables */
   tDatosAnalisis resultadosAnalisisTexto;
   tFrase frase;

   /* Pre: En la entrada tenemos una secuencia de frases, T, que
      puede ser vacía. Cada frase de T contiene información para
      saber si se debe corregir o no */
   obtenerFrase(&frase); /* preparar secuencia */
   inicioResultadosAnalisis(&resultadosAnalisisTexto);
                                    /* inicio tratamiento */
   while (!ultimaFrase(frase)) { /* fin secuencia */
      actualizaResultadosAnalisis(frase, resultadosAnalisisTexto);
               /* tratar elemento */
      obtenerFrase(&frase); /* avanzar secuencia */
   }
   escribirResultadosAnalisis(resultadosAnalisisTexto);
            /* tratamiento final */
   /* Post: En el caso de que no sea necesario corregir T, no se
      escribe nada. En el caso de que T sea un texto que se deba
      corregir, en la salida tiene que generarse una secuencia R
      formada por tres elementos: R1 R2 R3.
      R1 representa la media de palabras que se deben corregir
      por frase de T,
      R2 representa el porcentaje de frases que se deben corregir
      en T, y
      R3 representa el máximo de palabras que hay que corregir
      por frase de T. */

   system("PAUSE"); /* Hacemos que el programa espere una tecla */
   return 0;
}

/* 2.º nivel de diseño descendente. Funciones asociadas al tipo
   tFrase. Abstracción del tipo tPalabra. */
void obtenerFrase(tFrase *f)
{
   /* Declaración de variables */
   tPalabra m;
   int nPalabras;

   /* Pre: En la entrada tenemos un texto T representado como una
      secuencia de frases t. En la parte derecha de t, DT,
      hay una secuencia de frases o la frase vacía */
   /* Recorrido sobre una secuencia de palabras */
   nPalabras = 0; /* inicio tratamiento */
   f->nPalabrasC = 0; /* f->campo equivale a (*f).campo */
   obtenerPalabra(&m); /* preparar secuencia */
   while (!ultimaPalabra(m)) { /* fin secuencia */
      if (sedebeCorregirPalabra(m)) {/* tratar elemento */
         f->nPalabrasC = f->nPalabrasC + 1;
      }
      nPalabras = nPalabras + 1;
      obtenerPalabra(&m); /* avanzar secuencia */
   }
   if ( !PalabraVacia(m) ) {/* tratamiento final */
      if (sedebeCorregirPalabra(m)) {
         f->nPalabrasC = f->nPalabrasC + 1;
      }
      nPalabras = nPalabras + 1;
   }
   printf("nPalabras %d\n", nPalabras);
   f->final = nPalabras==0;
   /* Post: En f tenemos representada la primera frase de DT. La
```

```
      frase representada está en la parte izquierda de t */
}

bool ultimaFrase(tFrase f)
{
   /* Pre: f = F */
   return f.final;
   /* Post: ultimaFrase(F) es cierto si F es una frase vacía. Debe
      retornar falso en caso contrario */
}

int palabrasParaCorregir(tFrase f)
{
   /* Pre: f = F */
   return f.nPalabrasC;
   /* Post: palabrasParaCorregir(F) da el número de Palabras que
      se deben corregir en la frase F */
}

/* 2.° nivel de diseño descendente. Funciones asociadas al tipo
   tDatosAnalisis. */
/* Notad las siguientes abreviaciones:
   nFrases - número de frases del texto
   nFrasesC - número de frases que se deben corregir del texto
   nPalabrasC - número de palabras que se deben corregir del texto
   mx - máximo de palabras que hay que corregir por frase */
void inicioResultadosAnalisis(tDatosAnalisis *en)
{
   /* Pre: Cierto */
   (*a).nFrases = 0;
   (*a).nFrasesCor = 0;
   (*a).nPalabrasParaCorregir = 0;
   (*a).maximoPalabrasParaCorregir = 0;
   /* Post: En a tenemos nFrases, nFrasesC, nPalabrasC y mx de un
      texto vacío */
}

void actualizaResultadosAnalisis(tFrase f, tDatosAnalisis *a)
{
   /* Declaración de variables */
   int nPalabrasParaCorFrase;

   /* Pre: a = A y f = F */
   a->nFrases = a->nFrases + 1; /* a->campo equivale a (*a).campo*/
   nPalabrasParaCorFrase = palabrasParaCorregir(f);
   if (nPalabrasParaCorFrase>0) {
      a->nFrasesCor = a->nFrasesCor + 1;
      a->nPalabrasParaCorregir = a->nPalabrasParaCorregir +
         nPalabrasParaCorFrase;
   }
   if (nPalabrasParaCorFrase > a->maximoPalabrasParaCorregir) {
      a->maximoPalabrasParaCorregir = nPalabrasParaCorFrase;
   }
   /* Post: En a tenemos nFrases, nFrasesC, nPalabrasC y mx del
      texto resultante de añadir el análisis de la frase F
      al texto ya analizado en A */
}

void escribirResultadosAnalisis(tDatosAnalisis a)
{
   /* Pre: a = A */
   if (a.nFrasesCor > 0) {
      printf("%f ", (float)a.nFrasesCor / (float)a.nFrases * 100.0);
      printf("%f ", (float)a.nPalabrasParaCorregir /
                           (float)a.nFrasesCor);
      printf("%d ",a.maximoPalabrasParaCorregir);
   }
   /* Post: Si los datos de A son de un texto que se debe corregir,
      por el canal de salida se escribe una secuencia R formada
      por tres elementos: R1 R2 R3.
      R1 representa la media de palabras que se deben corregir
      por frase presente en el texto analizado en A,
      R2 representa el porcentaje de frases que se deben corregir
      en el texto analizado en A, y
      R3 representa el máximo de palabras que se deben corregir
      por frase existente en el texto analizado en A.
      En el caso de que A tenga los datos de un texto vacío
      no se escribe nada */
}

/* 3.er nivel de diseño descendente. Funciones asociadas al tipo
tPalabra. */
```

```
void obtenerPalabra(tPalabra *m)
{
   /* Declaración de variables */
   char c;
   int caracteresLeidos;
   /* Pre: En la entrada tenemos la secuencia de palabras, t, del
      texto T, de la que sólo podemos obtener la parte derecha
      DT, y ésta no ha llegado al final del texto */
   /* Búsqueda sobre una secuencia de caracteres. Buscamos '*'. */
   caracteresLeidos = 0; /* inicio tratamiento */
   scanf("%c", &c); /* preparar secuencia */
   while ( (c!='*')&&(c!=' ')&&(c!='.') ) {/* fin secuencia
                                          marca=' ' o '.' */
      caracteresLeidos = caracteresLeidos + 1;/* tratar elemento */
      scanf("%c", &c); /* avanzar secuencia */
   }
   m->sedebeCorregir = c=='*';/* tratamiento final */
   /* Recorrido sobre una secuencia de caracteres */
   /* inicio tratamiento - está vacío */
   /* preparar secuencia, no es necesario*/
   while ( (c!=' ')&&(c!='.') ) {/* fin secuencia marca=' ' o '.'*/
      caracteresLeidos = caracteresLeidos + 1; /* tratar elemento */
      scanf("%c", &c); /* avanzar secuencia */
   }
   m->vacio = caracteresLeidos==0; /* tratamiento final */
   m->final = c=='.';
   if ( (m->final) && !(m->vacio) ) scanf("%c", &c);
   /* Post: m representa la primera palabra que hay en DT.
      La palabra obtenida estará en la parte izquierda de la
      secuencia t */
}

bool ultimaPalabra(tPalabra m)
{
   /* Pre: m = M */
   return m.final || m.vacio;
   /* Post: ultimaPalabra(M) es cierto si M representa la última
      palabra de una frase */
}

bool palabraVacia(tPalabra m)
{
   /* Pre: m = M */
   return p.vacia;
   /* Post: palabraVacia(M) es cierto si M representa una palabra
      vacía */
}

bool sedebeCorregirpalabra(tPalabra m)
{
   /* Pre: m = M */
   return m.sedebeCorregir;
   /* Post: sedebeCorregirPalabra(M) es cierto si la palabra M
      se debe corregir. sedebeCorregirPalabra(M) es falso
      si no se debe corregir M */
}
```

Notad cómo las definiciones de nuevos tipos de datos se hacen en orden inverso al de los niveles de diseño descendente en el cual han sido introducidas. Hay que hacerlo así porque los tipos de datos que se definen en niveles inferiores pueden ser utilizados por los tipos de datos de niveles superiores con el fin de hacer una buena abstracción de datos. En este algoritmo en concreto, sin embargo, los nuevos tipos de datos asociados a cada nivel únicamente están basados en los tipos elementales del lenguaje C, por lo que no habría sido necesario hacer la declaración de tipos en orden inverso. A pesar de todo, es mejor habituarse a seguir siempre la misma rutina.

Otra buena costumbre de programación consiste en colocar las predeclaraciones de las acciones y funciones según el orden de su definición en cada nivel de diseño descendente. De manera similar, las acciones y funciones se tienen que

implementar organizándolas por niveles y según el tipo de datos con el cual se asocian. Todo contribuye a hacer el código más legible.

6. Construcciones avanzadas del lenguaje C

En este apartado se verán algunas construcciones del lenguaje C que no tienen equivalente en el lenguaje algorítmico, pero que permiten una codificación más simple, más eficiente y a veces más general. Tal y como ya se ha comentado con anterioridad, no se hará, ni mucho menos, un manual completo del lenguaje C, tampoco en este apartado. Sólo se comentarán aquellas construcciones sencillas de C que simplifican la codificación de nuestros algoritmos.

Este apartado es, evidentemente, opcional y en ninguno caso necesario para codificar los algoritmos desarrollados en lenguaje algorítmico.

6.1. Operadores aritméticos avanzados

Los operadores aritméticos avanzados de C permiten hacer una operación entre una variable y una expresión, guardando el resultado en la misma variable. La tabla siguiente muestra las características:

Operador	Ejemplo	Equivalencia en C
+=	*variable* += expresión;	*variable* = *variable* + expresión;
-=	*variable* -= expresión;	*variable* = *variable* - expresión;
*=	*variable* *= expresión;	*variable* = *variable* * expresión;
/=	*variable* /= expresión;	*variable* = *variable* / expresión;
%=	*variable* %= expresión;	*variable* = *variable* % expresión;
++	*variable*++ ++*variable*	*variable* = *variable* + 1;
--	*variable*-- --*variable*	*variable* = *variable* - 1;

Los operadores aritméticos avanzados se podrían usar donde la equivalencia en C fuese válida. Los operadores de autoincremento "++" y de autodecremento "--" merecen un comentario adicional. Se pueden usar en medio de una expresión y se pueden situar delante o detrás del nombre. Si se sitúan detrás del nombre, la expresión se evalúa primero y después se aplica el operador. Si se colocan ante el nombre, primero se evalúa el operador y después la expresión. Veamos un ejemplo de ello:

```
...
j = 0;
t[j++] = 1;    /* t[0] = 1 y j = 1 */
j = 2;
t[--j] = 2;    /* t[1] = 2 y j = 1 */
...
```

6.2. Funciones de entrada/salida

La sintaxis de las funciones de entrada/salida *scanf* y *printf* es mucho más completa de lo que se ha mostrado en las simples equivalencias entre las funciones de lectura y escritura del lenguaje algorítmico y el C.

6.2.1. Función de salida de datos *printf*

La invocación de la función de salida de datos *printf* tiene la siguiente sintaxis:

printf("*cadena_de_formato*", *dato*$_1$, *dato*$_2$.., *dato*$_n$);

Observemos que el primer argumento de la invocación se corresponde a una cadena de formato, gracias a la cual se indica el tipo de los datos que siguen en los argumentos segundo y posteriores. La función *printf* acepta un número variable de parámetros, donde el único parámetro obligatorio es el primero.

La cadena de formato está constituida por una cadena de caracteres que se escriben literalmente en la salida y unos códigos de control que especifican el formato de los datos que serán sustituidos en el lugar preciso ocupado para el código de control. Así, por ejemplo:

```
printf("El valor de pi es: %5.2f aproximadamente\n", 3.14);
```

Escribe por pantalla, a partir de la posición donde se encuentre el cursor:

```
El valor de pi es:  3.14 aproximadamente
```

Un código de control empieza siempre por el carácter "%" y su formato es:

%[anchura][.precisión]tipo

En C, una cadena de caracteres...

... es equivalente a una tabla de caracteres, pero se puede manipular más fácilmente con las funciones proporcionadas en las librerías del lenguaje.

Donde *anchura* es opcional e indica el número de caracteres que ocupará la salida (incluido el punto decimal y los decimales); *precisión* es también opcional y especifica el número de dígitos decimales a mostrar; y finalmente, *tipo*, es un carácter que indica el tipo de dato a mostrar. Los tipos posibles y el carácter que los identifica son: carácter ("c"), entero ("d"), real ("f") y cadena de caracteres ("s").

Para escribir algunos caracteres especiales se requieren códigos compuestos. Es el caso del salto de línea (`"\n"`), el tanto por ciento (`"%%"`), la barra invertida (`"\\"`), el tabulador (`"\t"`), la comilla simple (`"\'"`) y la comilla doble (`"\""`).

Como ejemplo final, el código:

```
printf("El %5.2f%% de las %s tiene %d trabajadores o menos.\n",
         85.32, "tiendas", 10);
```

Escribe por pantalla:

```
El 85.32% de las tiendas té 10 trabajadores o menos.
```

6.2.2. Función de entrada de datos *scanf*

La invocación de la función de entrada de datos *scanf* tiene la siguiente sintaxis:

scanf("*formato*", *dato*$_1$, *dato*$_2$.., *dato*$_n$);

Observamos que el primer argumento de la invocación se corresponde a un formato, gracias al cual se indica el tipo de los datos que se leerán sobre los argumentos segundo y posteriores.

El formato está constituido por unos códigos de control que especifican el formato de los datos que serán leídos del dispositivo de entrada y guardadas en las direcciones de memoria indicadas por los argumentos que siguen. Un código de control empieza siempre por el carácter "%" y su formato es:

%[*anchura*]*tipo*

Donde *anchura* es opcional e indica el número máximo de caracteres que se tienen que leer y *tipo*, es un carácter que indica el tipo de dato a leer. Los tipos posibles y el carácter que los identifica son: carácter ("c"), entero ("d"), real ("f") y cadena de caracteres ("s").

Como ejemplo, el código siguiente:

```
scanf("%d %f %20s", &i, &r, t);
```

Lee un entero y lo guarda en la variable *i*; lee un real y lo guarda en la variable *r* y, finalmente, lee una cadena de como máximo 20 caracteres y la guarda en la tabla de caracteres *t*. Observemos el uso del operador "&" para obtener la dirección de memoria donde se guardan las variables *i* y *r*. en Cambio, no se utiliza con la variable *t*, ya que se trata de una tabla y, como hemos visto el nombre de una tabla apunta a su primer elemento. Lo mismo pasaría si ya tuviéramos la dirección de una variable, es decir, un apuntador.

6.3. Inicialización de variables

El lenguaje C permite inicializar las variables en el momento de su declaración. Por eso, sólo hay que utilizar el operador asignación y un valor apropiado. Por ejemplo:

```
int i = 1, j = 0;
```

El valor asignado puede también ser una expresión, siempre y cuando los valores de las variables involucradas sean conocidos (hayan sido inicializados previamente). En general, sin embargo, hay que evitar este tipo de inicializaciones que hacen el código menos legible.

Si la variable es una tabla, la inicialización se hace entre llaves, separando cada elemento por comas. Si los elementos de la tabla son tablas o tuplas, la inicialización de éstas también se hace entre llaves. En este caso, el lenguaje C acepta también una inicialización con un número suficiente de elementos separados por comas, asignándolos por orden. En el siguiente código, las dos inicializaciones son equivalentes:

```
int m[2][3] = { {0, 1, 2}, {3, 4, 5} };
int t[2][3] = {0, 1, 2, 3, 4, 5};
```

Si la variable es una tupla, la inicialización se hace entre llaves, separando cada campo por comas. Si los elementos de la tupla son tablas u otros tuplas, la inicialización de éstas también se hace entre llaves. Así, por ejemplo:

```
struct {
   char c;
   int d;
   float f[2];
   struct {
      int i;
      float j;
   } dato;
} reg = {'c', 10, {2.3, 5.2}, {3, 6.2} };
```

6.4. Apuntadores y matrices

6.4.1. Concepto de apuntador

El valor de cada variable está almacenado en un lugar determinado de la memoria, accesible mediante una dirección. El ordenador mantiene una tabla de direcciones que relacionan los nombres de las variables con sus direcciones de memoria. Trabajando con estos nombres normalmente no hace falta que el programador se preocupe de la dirección de memoria que ocupan, pero en ciertas ocasiones es más útil trabajar con las direcciones que con los propios nombres de las variables.

El lenguaje C dispone de un tipo especial de variables que permiten contener estas direcciones, son los llamados *apuntadores*.

Así, un apuntador es una variable que permite contener la dirección de otra variable.

La forma general para declarar una variable apuntador es:

tipo *nombre;

donde *tipo* puede ser cualquier tipo válido de C (también denominado tipo base) al que apuntará la variable *nombre*. A partir de esta declaración la variable *nombre* podrá contener la dirección de cualquier variable del tipo base. Se dice que un apuntador apunta a una variable cuando el valor del apuntador es la dirección de esta variable.

6.4.2. Operadores dirección e indirección

Como ya hemos visto el lenguaje C dispone de un operador "&" que permite determinar la dirección de una variable y un operador "*" que permite acceder al valor contenido en la zona de memoria a la que hace referencia el apuntador.

Como ejemplo vemos estas declaraciones y sentencias:

```
int i,j;    /* i, j son variables enteras */
int *p;     /* p es un apuntador a un int */

p = &i;        /* p contiene la dirección de i */
*p = 5;     /* i toma el valor 5 */
p = &j;     /* p contiene ahora la dirección de j */
*p = 3;     /* j toma el valor 3 */
```

Las constantes y las expresiones no tienen dirección por lo que no se puede aplicar el operador "&". Tampoco se puede cambiar la dirección de memoria de una variable.
Los valores posibles para un apuntador son todas las direcciones válidas de memoria o el valor NULL que es una constante simbólica predefinida equivalente a 0 y que se utiliza para indicar que el apuntador no apunta a ningún sitio.

6.4.3. Aritmética de apuntadores

Las operaciones permitidas con variables de tipo apuntador son:

Las binarías "+" y "-" y las unarias "++" (autoincremento) y "--" (autodecremento)

Estas operaciones actúan de una forma un poco especial porque tienen en cuenta el tipo de dato al que se refiere el apuntador. Así, por ejemplo, si suman un 1 al valor de un apuntador, tendremos la dirección del siguiente objeto almacenado a la memoria. Lo mismo pasa con el decremento.

6.4.4. Relación con las tablas y las matrices

Los programas que trabajan con tablas y matrices a menudo realizan operaciones muy similares, pero sobre matrices que tienen índices de medidas diferentes. Las dos posibles formas de atacar el problema consisten en: o bien codificar una función específica para cada matriz de medida diferente; o bien declarar todas las matrices con la medida máxima y codificar una función que trate matrices de medidas variables, indicándole por parámetros las medidas reales de la matriz con la cual hay que trabajar.

Las dos opciones tienen inconvenientes. En la primera, se necesitan múltiples funciones que realizan el mismo tratamiento; simplemente cambian los límites de los índices. En la segunda opción, hay una única función, pero todas las matrices se declaran con unas dimensiones posiblemente superiores a las necesarias, malgastando espacio de memoria.

La solución, en C, consiste en trabajar con apuntadores. Efectivamente, el nombre de una matriz es, en realidad, un apuntador a la dirección donde se encuentra el primer elemento de esta. Así, partiendo de esta posición y usando la aritmética de apuntadores, que permite hacer operaciones aritméticas con los apuntadores, se puede recorrer la matriz. Por ejemplo:

```
/* Programa apuntadores.c
   Demuestra el uso de apuntadores para recorrer matrices
   Jordi Riera, 2001
*/
#include <stdio.h>

int main()
{
   int i, j;
   int m[2][3] = { {0, 1, 2}, {3, 4, 5} };   /* Inicialización de
                                                una matriz */
   int *p;

   /* Recorrido usando acceso directo al elemento de la tabla */
   for(i=0; i<2; i++) {
      for(j=0; j<3; j++) printf("%d ", m[i][j]);
      printf("\n");
   }
   /* Recorrido con apuntadores */
   p = &m[0][0]; /* p apunta a la dirección del primer
   elemento */
   for(i=0; i<2; i++) {
      for(j=0; j<3; j++) printf("%d ", *p++); /* *p es el elemento
                                        entero apuntado por p */
      printf("\n");
   }

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}
```

Observemos cómo en el programa se declara un apuntador al tipo de elementos de la matriz que se quiere recorrer. El apuntador se inicializa a la dirección del primer elemento de la matriz y a continuación avanza secuencialmente gracias al operador de autoincremento.

Un ejemplo...

... de codificación de funciones generales para tratamiento de matrices lo podéis encontrar en el programa *productoMatrices*, del ejercicio de autoevaluación número 8 de este módulo.

Usando las ideas introducidas en el programa *apuntadores*, es fácil pensar en codificar funciones generales que trabajen sobre matrices de varias dimensiones y de medidas diferentes. Habrá que pasar como parámetros el apuntador al primer elemento de la matriz, y las medidas de cada uno de los índices de la matriz actual. Así se podrá declarar cada matriz en su medida justa, sin ocupar más memoria de la necesaria, y sólo habrá una sola función por cada tipo de tratamiento, independientemente de la medida de los índices de la matriz.

Se podría pensar que este tipo de solución sólo es válida si la función realiza un tratamiento secuencial sobre la matriz. Por el contrario, la aritmética de apuntadores permite acceder a cualquier elemento de la matriz, si sabemos las medidas máximas de cada índice. Lo veremos en un ejemplo.

```
/* Programa apuntadorDirecto.c
   Demuestra el uso de apuntadores para acceder directamente a una
   posición de la matriz
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de constante en tiempo de compilación */
#define N 2
#define M 3

int main()
{
   int m[N][M] = { {0, 1, 2}, {3, 4, 5} };
   int *p;
   p = &m[0][0]; /* p apunta al primer elemento de la matriz */
   printf("%d\n", m[1][2]); /* Acceso directo al elemento m[1][2]
                               con matriz */
   printf("%d\n", *(p + 1*M + 2)); /* Acceso directo a m[1][2] con
                                      apuntador */
   system("PAUSE");
   return 0;
}
```

El programa *apuntadorDirecto* demuestra que es posible el acceso directo a un elemento de una matriz a través de un apuntador. Podríamos pensar que este método es más complicado, ya que hacen falta una multiplicación y dos sumas para calcular la dirección de memoria donde se guarda el elemento. En realidad, esta misma operación se realiza en el acceso directo a través de la matriz. Simplemente, que en este último caso la sintaxis no lo pone de manifiesto tanto claramente.

Notad, finalmente, que las operaciones aritméticas con apuntadores nos retornan un apuntador. Para acceder efectivamente al dato guardado en la dirección de memoria, hay que utilizar el operador "*".

6.5. Cadenas de caracteres

En el lenguaje C, las cadenas de caracteres tienen un elemento más para indicar el final de la cadena. Este elemento especial es el código "\0". Todas las funciones que trabajan con cadenas de caracteres suponen que éstas acaban con el carácter "\0". Hay que tenerlo presente al hacer entradas por teclado usando la función *scanf*, ya que habrá que guardar este carácter adicional. De forma similar, si se quiere utilizar la función *printf* para escribir una cadena de caracteres, habrá que asegurar que ésta acaba con el carácter "\0".

El tipo *tPalabra*...

... no guarda este carácter especial. Por lo tanto, no se puede leer o escribir el tipo *tPalabra* usando las funciones *scanf* y *printf* con cadenas de caracteres, sino que habrá que hacer el acceso carácter a carácter.

Las cadenas de caracteres que se manipulan en los programas codificados en C se guardan en tablas de caracteres. Por lo tanto, el acceso a estas cadenas también se puede hacer a través de apuntadores. De hecho, C incorpora múltiples funciones para tratar con cadenas de caracteres, pero éstas nos desviarían de los objetivos de la asignatura.

Finalmente, remarcar que las cadenas de caracteres se pueden inicializar fácilmente colocando los caracteres deseados entre comillas dobles. Veamos el siguiente ejemplo:

```
char p[] = "Hola";
char q[] = {'H', 'o', 'l', 'a', '\0'};
char r[10] = "Hola";
```

Las variables *p* y *q* son tablas de 5 posiciones. Las dos inicializaciones son completamente equivalentes. La variable *r* es una tabla de 10 posiciones. El contenido de las 5 primeras posiciones es idéntico al de las tablas de las variables *p* y *q*, mientras que las 5 posiciones restantes de *q* no están definidas.

Observemos que si al declarar una tabla no especificamos su medida, ésta se ajusta a la inicialización que se haga. Y, a la inversa, si la inicialización en la declaración no cubre todos los elementos de la tabla, los elementos restantes se quedan sin inicializar.

Resumen

En este módulo hemos aprendido como codificar en lenguaje C algoritmos descritos en la notación algorítmica estudiada en la asignatura. De hecho, la codificación se realiza mediante una traducción casi directa de las construcciones del lenguaje algorítmico en C. Así, la codificación se reduce a una sencilla y sistemática tarea de conversión entre lenguajes muy similares.

La sintaxis del lenguaje C se parece mucho a la del lenguaje algorítmico. Asimismo, la traducción de las construcciones del lenguaje algorítmico al inglés nos acerca mucho a las construcciones del lenguaje C. A pesar de eso, ya en un pequeño programa en C aparecen diferencias respecto del lenguaje algorítmico.

A modo de resumen del módulo, nos centraremos en las diferencias entre el lenguaje algorítmico y el lenguaje C. Así, en el lenguaje C:

- Se distingue entre las letras mayúsculas y minúsculas. Hace falta ir con cuidado y usar el nombre correcto de los objetos, funciones y acciones.

- Se necesita acceder a ficheros de cabecera con la directiva **#include** para tener acceso a funciones básicas del lenguaje (entrada/salida de datos, funciones matemáticas, etc.). Las directivas se deben escribir siempre en la primera posición de una línea de programa, normalmente en la cabecera del programa, antes de la función *main*.

- Se utilizan los caracteres "/*" y "*/" para delimitar los comentarios. En los programas usaremos comentarios:

  - en la cabecera del programa, indicando el nombre, la utilidad y el autor del programa;

  - para delimitar las secciones del programa: declaraciones de constantes, tipo y variables;

  - para especificar las precondiciones y postcondiciones del programa y sus acciones y funciones;

  - para aclarar los esquemas aplicados en la resolución de un subproblema, así como para hacer más legibles aquellas partes del código que puedan resultar menos evidentes.

- Se utiliza la función *main* para codificar el algoritmo principal y, desde ella se pueden invocar el resto de funciones y acciones del programa.

- Se pueden utilizar los tipos elementales de objetos del lenguaje algorítmico, con las mismas características. Hay que recordar, sin embargo, que para poder utilizar el tipo **bool** en lenguaje C, hay que declararlo previamente con:

```
typedef enum {FALSE, TRUE} bool;
```

- Se declararán objetos de manera similar al lenguaje algorítmico. La sintaxis, sin embargo, varía ligeramente. En el caso de objetos constantes tenemos la posibilidad de declarar su valor en tiempo de compilación con la directiva **#define**, lo cual es útil para dimensionar tablas y matrices. La definición de nuevos tipos en C sigue también una sintaxis ligeramente diferente de la del lenguaje algorítmico.

- Las funciones de conversión de tipo no son funciones propiamente dichas, sino que se indica directamente en la expresión el tipo de variable deseado.

- el operador de asignación "=" sustituye la acción elemental de asignación ":=" del lenguaje algorítmico. Un conjunto de acciones se pueden agrupar en un bloque si se colocan entre las claves "{" y "}".

- Las estructuras elementales del lenguaje algorítmico (**si**..., **mientras**..., **por**...) tienen un equivalente directo (**if**..., **while**..., **for**...). En el caso del **for** la sintaxis varía pero la funcionalidad es la misma.

- No hay acciones, sino que todo son funciones. Una acción se codificará como una función que devuelve el tipo especial **void**. La sintaxis para la declaración de acciones y funciones en C varía ligeramente respecto del lenguaje algorítmico. Básicamente, se indica primero el tipo de objeto devuelto y después el nombre de la función y sus parámetros.

- Los parámetros de salida y de entrada/salida de una función requieren el uso del carácter "*" precediendo el nombre formal del parámetro. En la invocación, hay que pasar la dirección de memoria del parámetro, precediendo normalmente el nombre del parámetro actual con un "&". Las tablas no siguen esta regla, ya que siempre se pasan por referencia. En el caso de las tuplas, para poder acceder a un campo dentro de la función, hay que rodear el nombre de la tupla ("*" incluido) entre paréntesis. Alternativamente, podemos usar el operador "->". Por ejemplo, es equivalente:

```
(*nombre).campo;
nombre->campo;
```

- Las funciones de entrada y salida de datos se hacen con las funciones *scanf* y *printf*. El primer parámetro indica el formato y el tipo de los valores a leer o escribir. Habitualmente traduciremos directamente las funciones de entrada salida del lenguaje algorítmico en los suyos equivalentes en C, aunque la sintaxis de las funciones *scanf* y *printf* es mucho más versátil.

- Permite hacer tratamiento secuencial, simplemente codificando en C los esquemas vistos en lenguaje algorítmico.

- Las declaraciones de tablas y tablas de tablas siguen una sintaxis muy similar a la del lenguaje algorítmico, pero simplemente hay que indicar el tipo de los elementos, sin especificar que se trata de una tabla. Eso ya se deduce al poner la medida entre corchetes "[" y "]". En cambio, las matrices n-dimensionales no existen en C. Hay que declararlas como tablas de tablas, con lo que se consigue la misma funcionalidad. El acceso a los elementos de una tabla para asignación, consulta, lectura y escritura, sigue los mismos mecanismos vistos en el lenguaje algorítmico.

- Los índice de las tablas empiezan en la posición 0, y no en la 1 como en el lenguaje algorítmico. Eso implica cambios en:

  - La inicialización de la variable índice con la cual se recorre la tabla, que tiene que ser a 0 en lugar de a 1.

  - El límite del recorrido de un índice sobre la tabla, que ha de ser una posición inferior a la del lenguaje algorítmico. Habitualmente, lo que se hace es comparar igualmente con la medida de la tabla, pero usando el operador "<" en lugar de hacerlo con el operador "<=" como en el lenguaje algorítmico.

  - El tratamiento del elemento actual cambia ligeramente. Hay que tener presente que la posición del elemento actual en C es una unidad inferior a la que tiene en la notación algorítmica. Típicamente, desaparece un posible +1 que hubiera en la expresión del lenguaje algorítmico, o bien, si no estaba, habrá que añadir un -1 en la expresión equivalente en C. Con frecuencia, en un recorrido o búsqueda secuencial sobre la tabla, será suficiente con cambiar el orden de las acciones correspondientes al acceso a la tabla y el incremento del índice.

- El paso de tablas como parámetros de acciones y funciones se hace siempre por referencia, y no es posible devolver una tabla en una función. Así en C, hay que tener presente que:

  - Una tabla pasada como parámetro de entrada se puede declarar como constante, para indicar al compilador que no se harán cambios. La notación algorítmica permite cambiar valores en la tabla dentro de la función sin que se vea afectada la tabla original en la llamada, pero esta característica normalmente no se utiliza. Si hubiera que usarla, tendría que declararse una tabla local a la función y hacer una copia de la tabla pasada como parámetro.

- Una tabla pasada como parámetro de entrada/salida o salida se puede usar normalmente. Sigue la notación de los tipos elementales, por lo que su nombre no tiene que ir precedido del carácter "*". Tampoco hay que preceder el nombre del parámetro actual por "&" en la invocación.

- No podemos tener una tabla como valor de retorno en una función. Podemos obtener la misma funcionalidad, declarando un nuevo parámetro de salida, el último en la función. Para hacer más legible el programa, usaremos el mismo nombre que la variable local devuelta en el lenguaje algorítmico.

• Las tuplas se declaran con una sintaxis muy similar a la de la notación algorítmica. El acceso a los campos de una tupla para asignación, consulta, lectura y escritura, sigue los mecanismos ya vistos en el lenguaje algorítmico. Al ser pasadas como a parámetros de acciones y funciones, las tuplas siguen las mismas reglas que los tipos elementales del lenguaje C (nombre precedido con "*" y paso de la dirección de memoria "&" en caso de parámetros de de entrada/salida). Hay que recordar, sin embargo, el acceso a un campo de una tupla pasada por referencia ("(**nombre*).*campo*" o bien "*nombre*->*campo*), ya comentado en este resumen. Asimismo el lenguaje C permite usar tuplas como valor de retorno de una función, aunque por razones de eficiencia se tendría que evitar.

• la metodología de diseño descendente aplicada al lenguaje algorítmico supone sólo tener que hacer las predeclaraciones de las acciones y funciones necesarias en cada nivel. Éstas se hacen antes de la función *main*, usando la misma sintaxis que en la cabecera de la definición de la acción o función correspondiente.

Hay que recordar que el enfoque del presente módulo se dirige a cubrir sólo aquella pequeña parte del C necesaria para poder codificar la notación algorítmica vista a la asignatura. A pesar de eso, se han comentado también algunos aspectos más avanzados del C que permiten obtener codificaciones más sencillas, a menudo más eficientes y a veces también más generales. En concreto, se han comentado brevemente:

- los operadores aritméticos avanzados, que permiten un código más compacto y eficiente;

- la sintaxis extendida de las funciones de entrada/salida *scanf* y *printf*, que permiten dar formato a la entrada/salida de los programas;

- la inicialización de variables en la declaración, tanto para los tipos elementales como para las tablas y tuplas;

- los apuntadores y su relación con las matrices, que permiten un código más eficiente y general;

- las cadenas de caracteres y su inicialización.

Ejercicios de autoevaluación

1. Codificad el algoritmo *potencia*, resuelto en el ejercicio de autoevaluación número 11 del módulo "Introducción al algorítmica" de la asignatura. Consultad el enunciado y la solución del ejercicio en las páginas 58 y 61, respectivamente.

2. Codificad el algoritmo *sumaCuadrados*, resuelto en el ejercicio de autoevaluación número 13 del módulo "Introducción al algorítmica" de la asignatura. Consultad el enunciado y la solución del ejercicio en las páginas 58 y 61, respectivamente.

3. Codificad el algoritmo *hitoAprobados*, resuelto en el apartado 4.3.3. "Número de aprobados" del módulo "Tratamiento secuencial" de la asignatura. Consultad el enunciado y la solución del ejercicio a la página 38 del módulo.

4. Codificad el algoritmo *longPrimPal*, resuelto en el apartado 5.2.2. "Longitud de la primera palabra" del módulo "Tratamiento secuencial" de la asignatura. Consultad el enunciado y la solución del ejercicio a partir de la página 42 del módulo.

5. Codificad el algoritmo *valMax*, resuelto en el ejercicio de autoevaluación número 2 del módulo "Tratamiento secuencial" de la asignatura. Consultad el enunciado y la solución del ejercicio en las páginas 47 y 48, respectivamente.

6. Codificad el algoritmo *esPerfecto*, resuelto en el ejercicio de autoevaluación número 4 del módulo "Tratamiento secuencial" de la asignatura. Consultad el enunciado y la solución del ejercicio en las páginas 47 y 49, respectivamente.

7. Codificad el algoritmo *esCapicua*, parcialmente resuelto en el apartado 2.4.3. "Ejemplos de los esquemas de recorrido y busca" del módulo "Tipos estructurados de datos" de la asignatura. Consultad el enunciado y la solución del ejercicio a partir de la página 30 del módulo. Observad que el algoritmo no está completo y que habrá que completarlo para poder hacer la entrada de datos y la salida de los resultados.

8. Codificad el algoritmo *productoMatrices*, resuelto en el apartado 2.4.3. "Ejemplos de los esquemas de recorrido y búsqueda" del módulo "Tipos estructurados de datos" de la asignatura. Consultad el enunciado y la solución del ejercicio a partir de la página 31 del módulo. Observad que hará falta declarar y definir la acción *rellenarTabla*. Pensad que habrá que adaptarla para tratar con matrices de diferente medida.

Consultad la acción *rellenarTabla* del algoritmo *cuadrados* en el apartado 2.2.2. “Lectura i escritura”, del módulo “Tipos estructurados de datos” de la asignatura.

9. Codificad el algoritmo *seleccionDirecta*, resuelto en el apartado 2.5.2. "Ordenación por selección" del módulo "Tipos estructurados de datos" de la asignatura. Consultad el enunciado y la solución del ejercicio a partir de la página 34 del módulo. Observad que hará falta declarar y definir la función *rellenarTabla*, y mostrar la tabla ordenada en el dispositivo de salida.

10. Codificad el algoritmo *codifica5En5*, resuelto en el ejercicio de autoevaluación número 3 del módulo "Tipos estructurados de datos" de la asignatura. Consultad el enunciado y la solución del ejercicio en las páginas 51 y 56, respectivamente.

Consultad la acción *invertirVector* en el apartado 2.4.3. “Ejemplos de los esquemas de recorrido y búsqueda”, del módulo “Tipos estructurados de datos” de la asignatura.

11. Codificad el algoritmo *transposicionDeTexto*, resuelto en el ejercicio de autoevaluación número 5 del módulo "Tipos estructurados de datos" de la asignatura. Consultad el enunciado y la solución del ejercicio en las páginas 51 y 58, respectivamente.

12. Codificad las funciones de tratamiento del tipo *palabra*, *leerPalabra*, *escribirPalabra*, *palabrasIguales*, resueltas en el ejercicio de autoevaluación número 7 del módulo "Tipos estructurados de datos" de la asignatura. Consultad el enunciado y la solución del ejercicio en las páginas 52 y 61, respectivamente.

13. Codificad la acción *escribirPuntos*, resuelta en el ejercicio de autoevaluación número 8 del módulo "Tipos estructurados de datos" de la asignatura. Consultad el enunciado y la solución del ejercicio en las páginas 53 y 63, respectivamente. Si queréis probar la acción en el ordenador, podéis simplificar el tipo *tEquipo* y dejar sólo los campos *nombre*, *partidosGanados* y *partidosEmpatados*. De esta manera, las inicializaciones en el programa principal serán más sencillas.

14. Codificad el algoritmo *contarLA*, resuelto en el ejercicio de autoevaluación número 1 del módulo "Introducción a la metodología del diseño descendente" de la asignatura. Consultad el enunciado y la solución (planteamiento general 2) del ejercicio en las páginas 35 y 39, respectivamente.

15. Codificad el algoritmo *cuentaCapicuasFrase*, resuelto en el ejercicio de autoevaluación número 3 del módulo "Introducción a la metodología del diseño descendente" de la asignatura. Consultad el enunciado y la solución del ejercicio en las páginas 36 y 43, respectivamente.

Solucionario

1. Codificación del programa potencia:

```
/* Programa potencia.c
   Calcula la potencia de un entero elevado a un entero.
   Jordi Riera, 2001
*/
#include <stdio.h>

int main()
{
   /* Declaración de variables */
   int n, exp, pot;
   scanf("%d", &n);
   scanf("%d", &exp); /* preparar secuencia */
   /* Pre: n=N y exp=EXP y EXP>=0 */
   pot = 1; /* inicio tratamiento */
   while (exp>0) { /* fin secuencia - exp=0 */
      pot = pot * n; /* tratar elemento */
      exp = exp - 1; /* avanzar secuencia */
   }
   /* Post: pot = N elevado a EXP */
   printf("%d\n", pot); /* tratamiento final */

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}
```

2. Codificación del programa sumaCuadrados:

```
/* Programa sumaCuadrados.c
   Lee n y suma los cuadrados de los n primeros naturales.
   Jordi Riera, 2001
*/
#include <stdio.h>

int main()
{
   /* Declaración de variables */
   int n, s, i;
   scanf("%d", &n);
   /* Pre: n=N */
   i = 1; /* preparar secuencia */
   s = 0; /* inicio tratamiento */
   while (i<=n) { /* fin secuencia - i>n */
      s = s + i * i; /* tratar elemento */
      i = i + 1; /* avanzar secuencia */
   }
   /* Post: s es la suma de cuadrados entre 1 y N */
   printf("%d\n", s); /* tratamiento final */

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}
```

3. Codificación del programa hitoAprobados:

```
/* Programa hitoAprobados.c
   Comprueba si el número de aprobados no es inferior a un hito
   dado.
   Jordi Riera, 2001
*/
#include <stdio.h>

int main()
{
   /* Definición de tipos */
   typedef enum {FALSE, TRUE} bool;
```

```
   /* Declaración de variables */
   bool encontrado;
   char car;
   int hito, n;
   /* Pre: En la entrada se lee un entero HITO Y una secuencia de
      caracteres del conjunto {"A", "B", "C", "c", "D" } seguida
      por una "Z" */
   scanf("%d", &hito);
   scanf("%c", &car); /* preparar secuencia */
   encontrado = FALSE;
   n = 0; /* inicio tratamiento */
   while (!(car=='Z') && !encontrado) {
            /* fin secuencia - marca="Z" */
      encontrado = (car=='c') || (car=='D') || (n>=hito);
            /* actualizar encontrado */
      if (!encontrado) {
         n = n + 1; /* tratar elemento */
         scanf("%c", &car); /* avanzar secuencia */
      }
   }
   if (n>=hito) { /* tratamiento final */
      printf("N\n");
   } else {
      printf("S\n");
   }
   /* Post: por el dispositivo de salida se escribe "N" si el
      número de aprobados no es inferior al HITO, y "S" en caso
      contrario */

   system("PAUSE"); /* El programa espera una tecla */
   return 0
}
```

4. Codificación del programa longPrimPal:

```
/* Programa lonPrimPal.c
   calcula la longitud de la primera palabra de un texto
   Jordi Riera, 2001
*/
#include <stdio.h>

int main()
{
   /* Definición de tipos */
   typedef enum {FALSE, TRUE} bool;

   /* Declaración de variables */
   char car;
   int n;
   bool encontrado;

   /* Pre: En la entrada se leerá una secuencia de caracteres que
      no contiene ningún punto, seguida por un punto */
   /* Busca la primera letra mayúscula. Salta los separadores ini
      ciales */
   scanf("%c", &car); /* preparar secuencia - leer() */
   encontrado = FALSE;
   /* inicio tratamiento - está vacío */
   while (!(car=='.') && !encontrado) {
                              /* fin secuencia - marca='.' */
      encontrado = (car<='Z') && (car>='A');
                              /* actualizar encontrado */
      if (!encontrado) {
         /* tratar elemento - está vacío */
         scanf("%c", &car); /* avanzar secuencia - leer() */
      }
   }
   /* tratamiento final - está vacío */
   /* Busca sobre la secuencia el primer carácter separador. Cuenta
   las letras de la palabra actual */
   /* preparar secuencia - no es necesario */
   encontrado = FALSE;
   n = 0; /* inicio tratamiento */
   while (!(car=='.') && !encontrado) {
                    /* fin secuencia - marca='.' */
      encontrado = (car>'Z') || (car<'A');
                     /* actualizar encontrado */
      if (!encontrado) {
         n = n + 1; /* tratar elemento */
```

Notad que ha habido que cambiar los nombres de las variables *long* y *encontr* del algoritmo original en las declaraciones, ya que no se adecuan a los nombres usados en el cuerpo del algoritmo.

```
            scanf("%c", &car); /* avanzar secuencia - leer() */
         }
      }
      printf("%d\n", n); /* tratamiento final */
      /* Post: Escribe la longitud de la primera palabra (secuencia de
         letras mayúsculas seguidas, o 0 si no había ninguna) */

      system("PAUSE"); /* El programa espera una tecla */
      return 0;
   }
```

5. Codificación del programa valMax:

```
   /* Programa valMax.c
      calcula el máximo de una secuencia de reales marcada con 0.0
      Jordi Riera, 2001
   */
   #include <stdio.h>

   int main()
   {
      /* Declaración de variables */
      float x, max;

      /* Pre: En la entrada se leerá una secuencia no vacía de reales
         que no contiene ningún 0.0, seguida de 0.0 */
      scanf("%f", &x); /* preparar secuencia - leer() */
      max = x; /* inicio tratamiento */
      scanf("%f", &x);
      while (x!=0.0) { /* fin secuencia - marca=0.0 */
         if (x>max) { /* tratar elemento */
            max = x;
         }
         scanf("%f", &x); /* avanzar secuencia - leer() */
      }
      printf("%f\n", max); /* tratamiento final */
      /* Post: Por el canal de salida se ha escrito el mayor real de
         los que se han leído por el canal de entrada */

      system("PAUSE"); /* El programa espera una tecla */
      return 0;
   }
```

6. Codificación del programa esPerfecto:

```
   /* Programa esPerfecto.c
      Comprueba si un número es perfecto, es decir, si es igual a la
      suma de sus divisores, excepto él mismo.
      Jordi Riera, 2001
   */
   #include <stdio.h>

   int main()
   {
      /* Declaración de variables */
      int x, i, sum;

      /* Pre: En la entrada hay un número entero mayor que cero, X */
      scanf("%d", &x);
      i = 1; /* preparar secuencia */
      sum = 0; /* inicio tratamiento*/
      while (i<x) { /* fin secuencia */
         if ((x % i)==0) { /* tratar elemento */
            sum = sum + i;
         }
         i = i + 1; /* avanzar secuencia */
      }
      if (sum==x) printf("S\n"); /* tratamiento final */
      else printf("N\n");
      /* Post: Por el canal de salida se ha escrito "S" si la suma de
         los divisores de X menores que X es igual a X, y "N"
         en caso contrario */

      system("PAUSE"); /* El programa espera una tecla */
      return 0;
   }
```

7. Codificación del programa esCapicua:

```
/* Programa esCapicua.c
   Comprueba si una secuencia de enteros es capicúa
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de tipos */
typedef enum {FALSE, TRUE} bool;

int main()
{
   /* Declaración de constantes */
   const int N = 10;

   /* Declaración de variables */
   int t[N];
   bool encontrado;
   int i;

   /* Pre: En la entrada hay una secuencia de N enteros */
   for( i=0; i<N; i++) scanf("%d", &t[i]);
                                  /* Rellenamos el vector t */
   /* Busca sobre la mitad del vector. Se usa acceso directo para
      comparar con la posición simétrica */
   i = 0; /* preparar secuencia */
   encontrado = FALSE;
   /* inicio tratamiento - está vacío */
   while (i<=(N / 2) && !encontrado) {
               /* fin secuencia - mitad del vector */
      if (t[i]!=t[N-i-1]) { /* tratar elemento */
         encontrado = TRUE; /* actualizar encontrado */
      } else {
         i = i + 1; /* avanzar secuencia*/
      }
   }
   if (!encontrado) printf("S\n"); /* tratamiento final */
   else printf("N\n");
   /* Post: Por el canal de salida se ha escrito "S" si la
      secuencia leída por el canal de entrada es capicúa, y se ha
      escrito "N" en caso contrario */

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}
```

8. Codificación del programa productoMatrices:

```
/* Programa productoMatrices.c
   Calcula el producto de dos matrices
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Declaración de constantes */
const int N = 1;
const int M = 2;
const int P = 3;

/* Predeclaración de funciones */
void rellenarMatrizA(int t[N][M]);
void rellenarMatrizB(int t[M][P]);
void escribeMatrizC(int t[N][P]);

int main()
{
   /* Declaración de variables */
   int a[N][M];
   int b[M][P];
   int c[N][P];
   int i, j, k, v;
   /* Pre: En la entrada hay una secuencia de enteros de N*M + M*P
      enteros */
   rellenarMatrizA(a);
   rellenarMatrizB(b);
```

```
   for (i=0; i<N; i++) /* Recorrido por las filas de c */
      for (j=0; j<P; j++) { /* Recorrido por las columnas de c */
         v = 0; /* Recorrido con tratamiento acumulativo */
         for (k=0; k<M; k++) v = v + a[i][k]*b[k][j];
         c[i][j] = v;
      }
   escribeMatrizC(c);
   /* Post: Por el canal de salida se ha escrito la matriz
      resultante del producto A*B, donde A guarda los N*M primeros
      enteros leídos, y B los siguientes M*P */

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}

void rellenarMatrizA(int t[N][M])
{
   /* Declaración de variables */
   int i, j;

   /* Pre: En la entrada hay una secuencia de enteros de N*M */
   for (i=0; i<N; i++)
      for (j=0; j<M; j++)
         scanf("%d", &t[i][j]);
   /* Post: La matriz t guarda la secuencia de la entrada */
}

void rellenarMatrizB(int t[M][P])
{
   /* Declaración de variables */
   int i, j;
   /* Pre: En la entrada hay una secuencia de enteros de M*P */
   for (i=0; i<M; i++)
      for (j=0; j<P; j++)
         scanf("%d", &t[i][j]);
   /* Post: La matriz t guarda la secuencia de la entrada */
}

void escribeMatrizC(int t[N][P])
{
   /* Declaración de variables */
   int i, j;
   /* Pre: t=T */
   for (i=0; i<N; i++) {
      for (j=0; j<P; j++) printf("%d ", t[i][j]);
      printf("\n");
   }
   /* Post: En la salida se ha escrito el contenido de T */
}
```

Tal como se puede comprobar en el programa, el hecho de tratar con matrices de medidas diferentes obliga a definir funciones específicas para llenar y escribir cada uno de las matrices. El lenguaje C, sin embargo, permite resolver más eficientemente este problema gracias a la utilización de apuntadores. El algoritmo quedaría tal como sigue:

En el apartado "Construcciones avanzadas del lenguaje C" de este mismo módulo podéis encontrar más información sobre cómo utilizar apuntadores para hacer tratamientos eficientes sobre tablas y matrices.

```
/* Programa productoMatricesApuntadores.c
   Calcula el producto de dos matrices
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Declaración de constantes */
const int N = 1;
const int M = 2;
const int P = 3;

/* Predeclaración de funciones */
void rellenarMatriz(int *t, int Nmax, int Mmax);
void escribirMatriz(int *t, int Nmax, int Mmax);

int main()
{
   /* Declaración de variables */
   int a[N][M];
   int b[M][P];
   int c[N][P];
   int i, j, k, v;
```

```
   /* Pre: En la entrada hay una secuencia de N*M + M*P enteros */
   rellenarMatriz(&a[0][0], N, M);
   rellenarMatriz(&b[0][0], M, P);
   for (i=0; i<N; i++) /* Recorrido por las filas de c */
      for (j=0; j<P; j++) { /* Recorrido por las columnas de c */
         v = 0; /* Recorrido con tratamiento acumulativo */
         for (k=0; k<M; k++) v = v + a[i][k]*b[k][j];
         c[i][j] = v;
      }
   escribirMatriz(&c[0][0], N, P);
   /* Post: Por el canal de salida se ha escrito la matriz
      resultante del producto A*B, donde A guarda los N*M primeros
      enteros leídos, y B los siguientes M*P */

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}

void rellenarMatriz(int *m, int Nmax, int Mmax)
{
   /* Declaración de variables */
   int i, j;

   /* Pre: En la entrada hay una secuencia de enteros de N*M */
      for (i=0; i<Nmax; i++)
         for (j=0; j<Mmax; j++) scanf("%d", m++);
   /* Post: La matriz m guarda la secuencia de la entrada */
}

void escribirMatriz(int *m, int Nmax, int Mmax)
{
   /* Declaración de variables */
   int i, j;
   /* Pre: m=M */
   for (i=0; i<Nmax; i++) {
      for (j=0; j<Mmax; j++) printf("%d ", *m++);
      printf("\n");
   }
   /* Post: En la salida se ha escrito el contenido de T */
}
```

9. Codificación del programa seleccionDirecta:

```
/* Programa seleccionDirecta.c
   Ordenación de un vector por selecció directa
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Declaración de constantes */
#define N 10

/* Declaración de tipos */
typedef int vector[N];

/* Predeclaración de funciones */
void rellenarVector(vector v);
void escribirVector(const vector v);

int main()
{
   /* Declaración de variables */
   vector t;
   int i, j, posMin;
   int aux;

   /* Pre: En la entrada hay una secuencia de N enteros */
   rellenarVector(t);
   i = 0; /* Primera posición del vector, 0 */
   while (i<N-1) { /* N-1 pasos sobre el vector */
      posMin = i; /* Recorrido desde i+1 hasta N */
      j = i + 1;
      while (j<N) {
         if (t[j]<t[posMin]) posMin = j;
         j = j + 1;
      }
      aux = t[i]; /* Intercambia posiciones */
      t[i] = t[posMin];
```

```
      t[posMin] = aux;
      i = i + 1;
   }
   escribirVector(t);
   /* Post: Por el canal de salida se han escrito en orden
      creciente los N enteros leídos por el canal de entrada */

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}

void rellenarVector(vector t)
{
   /* Declaración de variables */
   int i;

   /* Pre: En la entrada hay una secuencia de N enteros */
   for (i=0; i<N; i++) scanf("%d", &t[i]);
   /* Post: El vector t guarda, en el mismo orden, la secuencia de
         la entrada */
}

void escribirVector(const vector v)
{
   /* Declaración de variables */
   int i;
   /* Pre: v=V */
   for (i=0; i<N; i++) printf("%d ", v[i]);
   /* Post: En la salida se ha escrito el contenido de V, en el
         mismo orden */
}
```

10. Codificación del programa codifica5En5:

```
/* Programa codifica5En5.c
   Invierte cada grupo de 5 caracteres leídos de la entrada.
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Predeclaración de funciones */
void invTabla(char t[5], int N);

int main()
{
   /* Declaración de variables */
   char t[5];
   int i, lon;
   char c;

   /* Pre: En el canal de entrada hay una frase acabada en "." */
   scanf("%c", &c);
   while (!(c=='.')) {
      lon = 0;
      while (!(c=='.') && (lon<5)) {
         t[lon] = c;
         lon = lon + 1;
         scanf("%c", &c);
      }
      invTabla(t, lon);
      for (i=0; i<lon; i++) printf("%c", t[i]);
   }
   /* Post: En el canal de salida se genera una frase que
      corresponde a la leída por el canal de entrada, de manera que
      los grupos de 5 caracteres de la salida corresponden al mismo
      grupo que en la entrada, pero con el orden invertido. Estas
      agrupaciones no son aleatorias, sino que corresponden a
      agrupaciones de 5 caracteres correlativos desde el inicio de
      la frase */

   system("PAUSE");        /* El programa espera una tecla */
   return 0;
}

void invTabla(char t[5], int N)
{
   /* Declaración de variables */
   int i, aux;
```

```
   /* Pre: t=T contiene N valores enteros */
   i = 0;
   while (i<=(N/2)) {
      aux = t[i];
      t[i] = t[N-i-1];
      t[N-i-1] = aux;
      i = i + 1;
   }
   /* Post: Para i desde 0 hasta N-1, t[i] = t[N-i-1] */
}
```

11. Codificación del programa transposicionDeTexto:

```
/* Programa transposicionDeTexto.c
   Hace una transposición de texto usando una matriz de 10x10.
   Jordi Riera, 2001
*/
#include <stdio.h>

int main()
{
   /* Declaración de variables */
   char t[10][10];
   int fila, col;
   int maxFila, maxColumna;
   char c;

   /* Pre: En el canal de entrada hay una frase acabada en punto y
         de una longitud máxima de 100 caracteres */
   fila = 0;
   col = 0;
   scanf("%c", &c);
   while (c!='.') {
      t[fila][col] = c;
      col = col + 1;
      if (col==10) {
         col = 0;
         fila = fila + 1;
      }
      scanf("%c", &c);
   }
   maxFila = fila + 1;
   if (maxFila>1) {
      while (col<10) t[fila][col++] = '-';
                     /* Se debe rellenar toda la última fila */
      maxColumna = 10;
   } else {
      maxColumna = col;
   }
   for (col=0; col<maxColumna; col++)
      for (fila=0; fila<maxFila; fila++)
         printf("%c", t[fila][col]);
   /* Post: En el canal de salida se ha escrito una frase que
      resulta de aplicar la transposición de texto según una matriz
      de 10 filas y 10 columnas a la frase leída por el canal de
      entrada */

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}
```

12. Codificación del programa pruebaPalabra:

```
/* Programa pruebaPalabra.c
   Implementa y prueba funciones de tratamiento del tipo palabra
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de constantes */
#define MAXCAR 40

/* Definición de tipos */
typedef enum {FALSE, TRUE} bool;
typedef struct {
   char letras[MAXCAR];
   int l;
```

```
} palabra;

/* Predeclaración de funciones */
palabra leerPalabra();
void escribirPalabra(palabra p);
bool palabrasIguales(palabra p1, palabra p2);

int main()
{
   /* Declaración de variables */
   palabra p1, p2;
   /* Pre: En la entrada hay una secuencia de caracteres acabada
         en un punto */
   p1 = leerPalabra();     /* Leemos dos palabras de la entrada */
   p2 = leerPalabra();
   printf("\n");
   escribirPalabra(p2); /* Escribimos las palabras, e indicamos
                              si son iguales */
   printf(" == ");
   escribirPalabra(p1);
   printf(" ? ");
   if (palabrasIguales(p1, p2)) printf("SI\n");
   else printf("No\n");
   /* Post: Las dos primeras palabras leídas se han escrito por el
      dispositivo de salida, y se ha indicado si eran iguales o no*/

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}

palabra leerPalabra()
{
   /* Declaración de variables */
   palabra p;
   char c;

   /* Pre: Hay una secuencia de caracteres en la entrada */
   scanf("%c", &c);
   while (c==' ') scanf("%c", &c);
   p.l = 0;
   while ((c!='.') && (c!=' ')) {
      if (p.l<MAXCAR) {
         p.letras[p.l] = c;
         p.l = p.l + 1;
      }
      scanf("%c", &c);
   }
   return p;
   /* Post: p.l=0 si se lee algún carácter diferente del espacio;
         si no, p.letras contiene los caracteres diferentes de
         espacio que se han leído hasta el primer espacio posterior
         a algún carácter diferente de espacio, y p.l es
         el número de caracteres diferentes de espacio */
}

void escribirPalabra(palabra p)
{
   /* Declaración de variables */
   int i;

   /* Pre: p=P */
   for (i=0; i<p.l; i++) printf("%c", p.letras[i]);
   /* Post: En la salida se han escrito los p.l primeros caracteres
         de p.letras */
}

bool palabrasIguales(palabra p1, palabra p2)
{
   /* Declaración de variables */
   bool iguales;
   int j;

   /* Pre: p1=P! y p2=P2 */
   iguales = p1.l==p2.l;
   if (iguales) {
      j = 1;
      while ((j<p1.l) && (p1.letras[j]==p2.letras[j])) j++;
      iguales = p1.letras[j]==p2.letras[j];
   }
   return iguales;
   /* Post: iguales será cierto si P1.l=P2.l y los P1.l elementos
      de P1.letras son iguales a los P2.l elementos de P2.letras,
```

```
      y falso en caso contrario */
}
```

13. Codificación del programa escribirPuntos:

```
/* Programa escribirPuntosSimplificado.c
   Prueba la acción escribirPuntos
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de constantes */
#define MAXCAR 40
#define MAXJUGADORES 25
#define MAXEQUIPOS 20

/* Definición de tipos */
typedef struct {
   char letras[MAXCAR];
   int l;
} palabra;
typedef struct {
   palabra nombre;
   int numAmarillas, numRojas;
   int partidosJugados;
} tJugador;
typedef struct {
   palabra nombre;
   palabra ciudad;
   int partidosGanados, partidosPerdidos, partidosEmpatados;
   int golesFavor, golesContra;
} tEquipo; /* Tipo tEquipo simplificado para facilitar la
              inicialización */
/* tipo tEquipo completo sería de esta manera:
typedef struct {
   palabra nombre;
   palabra ciudad;
   int partidosGanados, partidosPerdidos, partidosEmpatados;
   int golesFavor, golesContra;
   struct {
      tJugador jugador[MAXJUGADORES];
      int num;
   } jugadores;
} tEquipo;
*/
typedef struct {
   tEquipo equipos[MAXEQUIPOS];
   int num;
} ligaFutbol;

/* Predeclaración de funciones */
void escribirPuntos(const ligaFutbol l);
void escribirPalabra(palabra p);

int main()
{
   /* Declaración de variables */
   ligaFutbol liga = { { /* Tabla de equipos */
   {{"F.C.Barcelona", 13}, {"Barcelona", 9}, 5, 3, 2, 12, 5},
   {{"R.C.D.Español", 14}, {"Barcelona", 9}, 3, 1, 4, 5, 15}}, 2};
   /* Pre: En la entrada hay una secuencia de caracteres acabada
         en un punto */escribirPuntos(liga);
   /* Post: Escribe en pantalla los nombres y puntos de los equipos
         de la liga */

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}

void escribirPuntos(const ligaFutbol l)
{
   /* Declaración de variables */
   int j;

   /* Pre: l=L */
   for (j=0; j<l.num; j++) {
      escribirPalabra(l.equipos[j].nombre);
      printf(": ");
      printf("%d ", l.equipos[j].partidosEmpatados +
```

```
                        3*l.equipos[j].partidosGanados);
      printf("\n");
   }
   /* Post: Se ha escrito en el canal estándar el campo nombre de
   los L.num elementos de la tabla L.equipos junto con
   los puntos de cada uno. */
}

void escribirPalabra(palabra p)
{
   /* Declaración de variables */
   int i;
   /* Pre: p=P */
   for (i=0; i<p.l; i++) printf("%c", p.letras[i]);
   /* Post: En la salida se han escrito los P.l primeros caracteres
         de P.letras */
}
```

14. Codificación del programa contarLA:

```
/* Programa contarLA.c
   Cuenta el número de "LA" que hay en una frase acabada
   en un punto
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de tipos */
typedef enum {FALSE, TRUE} bool;
typedef char tPareja[2]; /* Definido en el 1er nivel */

/* Predeclaración de funciones */
void obtenerPrimeraPareja(tPareja p); /* Funciones definidas en
                                         el 1.er nivel */
void obtenerPareja(tPareja p);
bool últimaPareja(tPareja p);
bool esLA(tPareja p);

/* 1.er nivel de diseño descendente. Recorrido sobre una secuencia
   de parejas. Abstracción del tipo tPareja.
*/
int main()
{
   /* Declaración de variables */
   int nbDeLA;
   tPareja pareja;
   /* Pre: f=F y f es una frase acabada en punto y está disponible
      en el canal de entrada */
   nbDeLA = 0; /* inicio tratamiento */
   obtenerPrimeraPareja(pareja); /* preparar secuencia */
   while ( !ultimaPareja(pareja) ) { /* fin secuencia */
      if (esLA(pareja)) nbDeLA++; /* tratar elemento */
      obtenerPareja(pareja); /* avanzar secuencia */
   }
   printf("%d", nbDeLA); /* tratamiento final */

   /* Post: Se ha escrito por el canal de salida nbDeLA, que indica
      el número de "LA" que hay en F */

   system("PAUSE"); /* El programa espera una tecla */
   return 0;
}

/* 2.º nivel de diseño descendente. Funciones asociadas al tipo
      tPareja. */
void obtenerPrimeraPareja(tPareja p)
{
   /* Pre: s=S y s es una frase acabada en punto y la parte
         izquierda de s está vacía */
   p[0] = ' ';
   scanf("%c", &p[1]);
   /* Post: p contiene la primera pareja de s y la parte izquierda
      de s contiene un carácter */
}

void obtenerPareja(tPareja p)
{
   /* Pre: s=S y s es una frase acabada en punto y la parte
      izquierda de s no está vacía y su parte derecha tampoco;
      además, p=P y P es la anterior pareja obtenida */
```

```
   p[0] = p[1];
   scanf("%c", &p[1]);
   /* Post: p contiene la pareja siguiente de la secuencia de
      parejas y la parte izquierda de s contiene un carácter más */
}

bool ultimaPareja(tPareja p)
{
   /* Pre: p=P */
   return p[1]=='.';
   /* Post: Retorna CIERTO si p es la última pareja de la
         secuencia*/
}

bool esLA(tPareja p)
{
   /* Pre: p=P */
   return (p[0]=='L') && (p[1]=='A');
   /* Post: Retorna CIERTO si p es la pareja "LA" */
}
```

15. Codificación del programa cuentaCapicuasFrase:

```
/* Programa cuentaCapicuasFrase.c
   Cuenta el número de palabras capicúas que hay en una frase
   acabada en un punto
   Jordi Riera, 2001
*/
#include <stdio.h>

/* Definición de constantes */
#define MAXCAR 15

/* Definición de tipos */
typedef enum {FALSE, TRUE} bool;
typedef struct {
   int lon;
   char c[MAXCAR];
   char caracterLeido; /* Contiene el último carácter leído */
   bool final;
} tPalabra; /* Definido en el 1er nivel */

/* Predeclaración de funciones */
void obtenerPrimerPalabra(tPalabra *m); /* Funciones definidas en
                                           el 1.er nivel */
void obtenerPalabra(tPalabra *m);
bool palabraFinal(tPalabra m);
bool palabraCapicua(tPalabra m);
bool separador(char c); /* Funciones definidas en el 2.º nivel */

/* 1.er nivel de diseño descendente. Recorrido sobre una secuencia
   de palabras. Abstracción del tipo tPalabra.
*/
int main()
{
   /* Declaración de variables */
   int nPalabrasC;
   tPalabra palabra;
   /* Pre: En la entrada hay una secuencia de palabras f de la
         frase F. F no está vacía */
   nPalabrasC = 0; /* inicio tratamiento */
   obtenerPrimerPalabra(&palabra); /* preparar secuencia */
   while ( !palabraFinal(palabra) ) { /* fin secuencia */
      if (palabraCapicua(palabra)) nPalabrasC++;
                                    /* tratar elemento */
      obtenerPalabra(&palabra); /* avanzar secuencia */
   }
   if (palabraCapicua(palabra)) nPalabrasC++;
                                    /* tratamiento final */
   printf("%d", nPalabrasC);
   /* Post: Se ha escrito por el canal de salida el número de
         palabras capicúas de F */

   system("PAUSE");
   return 0;
}

/* 2.º nivel de diseño descendente. Funciones asociadas al tipo
      tPalabra. */
void obtenerPrimeraPalabra(tPalabra *m)
```

```
{
   /* Declaración de variables */
   char c;
   /* Pre: En la entrada hay la secuencia de palabras f de la frase
      F. La parte izquierda de f está vacía. La parte derecha,
      DF, no está vacía */
   /* Recorrido sobre una secuencia de caracteres */
   m->lon = 0;
   scanf("%c", &c);
   while ( separador(c) && (c!='.') ) scanf("%c", &c);
                           /* Salta los primeros separadores */
   m->caracterLeido = c;
   obtenerPalabra(m);
   /* Post: m representa la primera palabra de DF. La palabra
         obtenida estará en la parte izquierda de la secuencia f */
}

void obtenerPalabra(tPalabra *m)
{
   /* Declaración de variables */
   char c;
   /* Pre: En la entrada se encuentra la secuencia de palabras f
         de la frase F. La parte derecha, DF, no está vacía */
   m->lon = 0;
   c = m->caracterLeido;
   while ( !separador(c) && (c!='.') ) { /* Obtenemos la palabra */
      m->c[m->lon++] = c;
      scanf("%c", &c);
   }
   while ( separador(c) && (c!='.') ) scanf("%c", &c);
                           /* Salta los primeros separadores */
   m->caracterLeido = c;
   m->final = c=='.';
   /* Post: m representa la primera palabra de DF. La palabra
         obtenida estará en la parte izquierda de la secuencia f */
}

bool palabraFinal(tPalabra m)
{
   /* Pre: m=M */
   return m.final;
   /* Post: Retorna cierto si m es la última palabra de una frase*/
}
bool palabraCapicua(tPalabra m)
{
   /* Declaración de variables */
   int i, j;
   bool encontrado;
   /* Pre: m=M */
   i = 0;
   j = m.lon-1;
   encontrado = FALSE;
   while (!encontrado && (i<j)) {
      if (m.c[i]!=m.c[j]) {
         encontrado=TRUE;
      } else {
         i++;
         j--;
      }
   }
   return !encontrado;
   /* Post: Retorna cierto sólo si m es un palabra capicúa */
}

bool separador(char c)
{
   /* Pre c=C */
   return (c==' ') || (c==',') || (c==';');
   /* Post: Retorna cierto si C es un separador - espacio, coma o
         punto y coma */
}
```

Glosario

compilación y enlazado
Procesos por los cuales un programa escrito en un lenguaje de alto nivel es traducido a un código máquina que el ordenador es capaz de ejecutar.

fichero de cabecera (o *include*)
Fichero que contiene definiciones de nuevos tipos y declaraciones de constantes, acciones y funciones que después podrán ser utilizadas en el programa.

tiempo de compilación (de un programa)
Periodo en el cual el programa es compilado y traducido a lenguaje máquina.

tiempo de ejecución (de un programa)
Periodo en el cual el programa se está ejecutando en el ordenador.

Bibliografía

Kernighan, B.W.; Ritchie, D.M. (1991). *El lenguaje de Programación C*. 2ª Edición. México: Prentice Hall Hispanoamaricana.