Capítulo 5

Estructuras de Datos Estáticas

Índice General

5.1. Necesidad de las Estructuras de Datos. 120
5.1.1. Ampliación de la Noción de Tipo de Datos. 121
5.1.2. Clasificación de las Estructuras de Datos. 122
5.2. Vectores. 122
5.2.1. Necesidad de los Vectores. 123
5.2.2. Declaración de vectores en C. 124
5.2.3. Operaciones elementales de acceso en C. 126
5.2.4. Cadenas de caracteres. 128
5.2.5. Vectores Multidimensionales. Matrices. 129
5.3. Tuplas 131
5.3.1. Necesidad de las tuplas 132
5.3.2. Declaración de tuplas en C. 132
5.3.3. Operaciones elementales de acceso en C. 134
5.3.4. Combinación de distintos tipos 136
5.4. Algoritmos de manipulación de las estructuras de datos 137
5.4.1. Esquema de Recorrido 137
5.4.2. Esquema de Búsqueda 141
5.5. Resumen y Consideraciones de Estilo. 142
5.6. Glosario. 143
5.7. Bibliografía. 144
5.8. Problemas Propuestos. 144

Un informático, que se había quedado solo al cuidado de sus 10 hijos, grita desesperado cuando vuelve su mujer:
"¡¡Es horrible, horrible!! ¡¡No sé cómo ha ocurrido, pero he perdido a uno de los niños!!"
"¿Cómo?... pero, no, hombre están aquí los 10..." "No, no, cuéntalos y verás: 0, 1, 2, ..."

Versión libre de un chiste malo que circula por los procelosos mares de internet.

5.1. Necesidad de las Estructuras de Datos.

En el tema 2 comenzó el desarrollo de contenidos básicos de la asignatura; y fue, precisamente, estudiando los objetos básicos del entorno de la programación. También al principio de ese tema se remarcaba que la Informática es la *ciencia para el proceso automático de la información*: de ahí, que lo primero que se debía conocer y manejar era su entorno de trabajo básico, los objetos que, debidamente manipulados, permiten realizar ese proceso.

Hasta este momento sólo se han utilizado los tipos elementales: Entero, Real, Carácter y Boole. Estos tipos han sido suficientes para resolver los problemas que se han planteado por ahora. Pero, a medida que aumenta la complejidad de los problemas a resolver, se necesita diseñar algoritmos que utilicen otros tipos de datos: un problema complejo suele implicar la necesidad de manejar mayor cantidad de datos y, dependiendo de cómo se representen estos datos, de cómo se estructuren, la obtención del algoritmo será más o menos compleja.

En este tema comienza el estudio de las *estructuras de datos*, es decir, de cómo agrupar datos de forma conveniente para modelizar o representar mejor los datos del mundo real. El objetivo de este estudio es obtener una representación de la información que permita diseñar un algoritmo lo más sencillo y eficiente posible. De hecho, la elección correcta de la representación más idónea para desarrollar el proceso es uno de los puntos claves en el desarrollo del software.

> *Como un ejemplo, se propone el problema de localizar un libro en el fichero de una biblioteca. Dependiendo de cómo se haya organizado la información sobre los libros, la tarea puede ser más o menos difícil.*
>
> *En la biblioteca A, por ejemplo, el criterio de organización es ordenar los libros por color, tamaño y número de páginas. En la biblioteca B, el criterio de organización es por materia, autor y título.*
>
> *¿En qué biblioteca será más sencilla la tarea de localizar la bibliografía de esta asignatura?*

> *Como segundo ejemplo, se considera la representación de un valor numérico; un mismo valor admite una representación mediante el sistema de numeración romano (`III, VI, XXXVII, MCCCLI`...) o el sistema de dígitos árabigos y el trabajo en base 10 (`3, 6, 37, 1351`...).*
>
> *¿Qué representación permite realizar de forma más eficiente sumas, restas, multiplicaciones y divisiones?*

Estos y otros ejemplos de la vida cotidiana permiten ver que una buena elección de la representación de los datos, facilita su posterior manejo. En términos de programación, se puede decir que *una buena estructuración de los datos que se van a manejar, proporciona una mayor sencillez y eficiencia al algoritmo que los manipule*.

Sencillez, en el sentido de que si los datos se parecen más a los datos del mundo real a los que representan, los algoritmos serán más claros y fáciles de leer, modificar y mantener. Y eficiencia, en el sentido de que se podrán tratar de forma más potente y compacta, como se irá viendo a medida que se vayan presentando las distintas estructuras.

5.1.1. Ampliación de la Noción de Tipo de Datos.

En el manejo y, sobre todo, en el diseño de las estructuras de datos es posible distinguir varias etapas[1], entre las que cabe destacar:

Abstracción Es el proceso mediante el cual se reconocen aquellas propiedades que son comunes entre muchos objetos y, además, se identifican, entre dichas propiedades comunes, aquellas más adecuadas al proceso posterior que se realizará sobre esos objetos.

Representación Una vez que se sabe qué propiedades interesa abstraer, es preciso establecer la codificación más adecuada para permitir una interpretación correcta y sencilla de los datos.

Manipulación Y, por último, una vez que la información se ha codificado es preciso establecer las normas que permitan su tratamiento y la obtención de resultados correctos. Es decir, *definir las operaciones* que permitan manipularla.

Es decir, un buen programador debe saber abstraer aquello que sea representativo en la información del mundo real, pero además, condicionado por el hecho de que va a representar dicha información en un computador, tendrá que codificarla y posteriormente ha de saber manipularla.

De este proceso se desprende un concepto, mucho más completo y más potente, que el de estructura de datos; realmente, un programador debe definir un *Tipo de Datos*: no sólo hay que diseñar la estructura bajo la que se representarán los datos, también hay que diseñar las operaciones que permitan su manipulación[2].

Estructura de Datos + Operaciones de manipulación = TIPO de DATOS

Hasta este momento sólo se han utilizado los *tipos predefinidos*, los que suministra directamente el lenguaje de programación. En un tipo predefinido, tanto la estructura como las operaciones de manejo de la estructura están impuestos por el lenguaje: el programador se adapta a lo especificado, sin más libertad que elegir el tipo que considera más conveniente. Como contrapartida no debe preocuparse por cómo se implementa el tipo. Simplemente, define variables que contengan valores de un determinado tipo y usa las operaciones que le proporciona el lenguaje. Así, se han utilizado hasta el momento datos de tipo entero, por ejemplo, sin que nadie se haya preguntado cómo representa un lenguaje los enteros (la estructura ¿son 4 bytes en binario natural? ¿son 5 bytes y hay un bit de signo?) ni cómo se implementan las operaciones (¿cuál es el algoritmo para la suma binaria con acarreo? ¿cuál es el algoritmo para el cociente de la división entera?).

A partir de ahora, se comenzará a definir y diseñar *tipos definidos por el programador*; es decir, primero hay que elegir una estructura adecuada para los datos (normalmente, utilizando recursos que proporcionan los lenguajes de programación) y después hay que definir y diseñar las operaciones de manipulación (normalmente, definir y diseñar los algoritmos adecuados).

En esta categoría, los más sencillos de definir son *tipos estructurados básicos*, que se construyen utilizando estructuras de datos básicas: el lenguaje ya las tiene implementadas y dota al programador de las herramientas de definición y de las operaciones básicas de manipulación. Su estudio será el objetivo de este tema, y se centrará en dos estructuras básicas de la mayoría de los lenguajes de programación: *vectores*, de una o más dimensiones, y *tuplas*.

[1] Hay más etapas, sobre todo cuando este estudio se realiza con la formalización que requieren los denominados Tipos Abstractos de Datos, pero este concepto queda cubierto por la asignatura "Estructura de Datos y de la Información".

[2] Recuérdese la definición del atributo Tipo en el tema 2, "permite clasificar los objetos según su utilidad y las operaciones que se pueden hacer con ellos".

A partir de estas estructuras, se pueden definir otras más complejas que dan lugar a los tipos de datos pila, cola, lista, árbol... En este caso, el programador debe definir completamente tanto la estructura como las operaciones básicas de manejo.

5.1.2. Clasificación de las Estructuras de Datos.

El primer paso en la definición de un tipo es la elección de la estructura de datos más adecuada para los datos que se pretende manipular. Por lo tanto, conviene familiarizarse primero con las propiedades más usuales que presentarán dichas estructuras. Se pueden establecer varias clasificaciones,

- Por los tipos de datos que agrupan:

  Estructura Homogénea: Todos los datos son del mismo tipo.

  Estructura Heterogénea: Se agrupan datos de distintos tipos.

- Por la forma de acceder a los datos:

  Secuencial: Se accede a los datos uno tras otro, de forma que para acceder a un dato que está en una determinada posición, hay que acceder previamente a todos los que le preceden.

  De acceso directo: Se puede acceder directamente a cualquier dato de la estructura, independientemente de dónde o cómo esté almacenado.

- Por el número de elementos que puede contener:

  Estática: El número de elementos de la estructura no puede cambiar en tiempo de ejecución. Se debe definir antes de poder utilizarlo, es fijo y no varía.

  Dinámica: El número de elementos que forma la estructura es variable, puede crecer y decrecer en función de las necesidades del algoritmo en tiempo de ejecución.

Según estas clasificaciones, sobre los tipos de datos que se tratarán en este tema, se puede decir lo siguiente:

- Los *vectores* se basan en estructuras que agrupan datos del mismo tipo (estructura homogénea), con acceso directo y de tamaño fijo. Lo mismo rige para vectores de varias dimensiones (por ejemplo, las *matrices*, vectores de dos dimensiones).
- Las *tuplas*[3] se basan en estructuras que agrupan datos del diferentes tipos tipo (estructura heterogénea), con acceso directo y de tamaño fijo.

5.2. Vectores.

Definición 5.1 (Vector) *Es un conjunto finito y ordenado de elementos homogéneos.*

[3]También conocidas como *registros* o *estructuras* dependiendo del lenguaje de programación o el entorno en el que se definan.

En esta definición cabe puntualizar que,

Finito indica que el vector tiene un número determinado y fijo de elementos,

Homogéneo indica que todos los elementos son del mismo tipo, al que se denominará *tipo base* (TB ó tipoBase, serán abreviaturas habituales),

Ordenado indica que sus elementos están identificados por el orden que ocupan, por un *índice*: hay un primer elemento, un segundo elemento,..., un i-ésimo elemento. Y es, quizás, la principal característica de este tipo de datos.

Es decir, *los vectores agrupan valores del mismo tipo base, en un orden dado*. Y este orden define unívocamente a cada uno de los elementos, lo identifica mediante su posición: es lo que se conoce como *indexación*.

El tipo vector se construye considerando las siguientes operaciones elementales:

Crear cuyo resultado es un vector de tamaño definido, pero cuyos componentes individuales no están definidos,

Extraer que permite consultar el valor de un elemento: dado un vector `v` y un valor de índice `i`, su resultado es el valor almacenado en la posición `i` del vector `v`,

Almacenar que permite asignar un valor a un elemento del vector: dado un vector `v`, un valor de índice `i` y un valor `k` del tipo base, su resultado es un nuevo valor de `v` que se diferencia del anterior en que el valor `k` está ahora almacenado en la posición `i`.

De estas operaciones se desprende que, cuando se trabaja con vectores, no se pueden realizar operaciones con todo el vector, sólo con elementos individuales (a no ser, claro está, que el propio programador enriquezca el tipo, dotándole de nuevas operaciones).

5.2.1. Necesidad de los Vectores.

Hay muchos ejemplos del mundo real cuya información se puede modelizar mediante vectores; por ejemplo, listas de valores del mismo tipo como guías telefónicas, el censo, una lista de alumnos... Eso sin mencionar, todas las aplicaciones matemáticas basadas en el cálculo vectorial en las que la relación entre el problema y el tipo de datos es obvia.

Existe, además, un componente del mundo real informático que puede ser visto como un gran vector: la memoria del ordenador, formada por palabras de memoria (todas del mismo tipo) e indexada mediante las direcciones.

Pero, además, el uso de los vectores también puede simplificar el diseño de los algoritmos, como muestra el siguiente ejemplo:

> *Se quiere diseñar un algoritmo que cuente cuántas vocales distintas aparecen en un texto, es decir, cuántas veces aparece la `a`, la `e`... Se puede pensar en diseñar un algoritmo con 5 contadores distintos o bien, un vector de 5 elementos. Puede no parecer un serio problema, pero . . . ¿qué ocurre si se modifica el problema y lo que se quiere es contar cuántas letras distintas hay?. En este caso, la primera solución llevaría a declarar y manejar 27 (!!!) contadores o bien... un único vector con 27 elementos.*
>
> *Y aún más : ¿qué ocurre si se deben tratar miles o cientos de miles de elementos?.*

El hecho de manejar un único objeto, cuyos valores individuales pueden indexarse, hacen que los vectores se hagan imprescindibles no sólo para representar información del mundo real que se pretende procesar: el ejemplo pone de relieve que esta característica los hace ideales también para realizar determinados tratamientos que de otra forma serían muy ineficientes; así, para iniciar a cero los 27 contadores (o los miles) del anterior ejemplo, se necesitarían 27 líneas de código (o miles): el uso combinado de un vector como variable interna y un bucle que permita recorrer todos sus elementos, hace que el proceso se pueda describir en un par de líneas.

5.2.2. Declaración de vectores en C.

En el lenguaje `C`, como en la mayoría de los lenguajes de programación, se dispone de los elementos básicos para trabajar con vectores, si bien es el propio programador quien debe definir la estructura concreta que se va a utilizar. Se puede asimilar esta definición con la operación de *crear* un vector. Y para ello se debe indicar el tipo base, el tamaño y el nombre con el que se va a identificar.

La sintaxis para definir un vector en `C` es la siguiente:

<**tipoBase**> <**nombre**>**[TAM];**

donde se distinguen las siguientes partes:

- `tipoBase`: Es el tipo de datos de los elementos que agrupa.
- `nombre`: Es el identificador del vector.
- `[TAM]`: Define el número de elementos del vector. Ha de ser una constante entera positiva. Si `TAM` es el tamaño del vector, los índices de los elementos tomarán valores entre `0` y `TAM-1`.

Gráficamente, se puede representar del siguiente modo:

0	1	2	3	4	5	...	TAM-1

Así por ejemplo,

```
int s[10], t[10];
```

define dos vectores, `s` y `t`, cada uno de los cuales contiene 10 enteros, numerados del 0 al 9[4]. Y

```
float recaudacion[7];
```

[4]Hay que hacer notar que algunos lenguajes empiezan a numerar los elementos desde el 1. También se puede utilizar un tipo enumerado o un subrango (en los lenguajes que lo permitan) para definir el tamaño de un vector y para acceder a sus elementos.

define un vector denominado `recaudación` con 7 elementos de tipo `float` (reales) que podría utilizarse, por ejemplo, para almacenar la recaudación de cada uno de los días de la semana de un cine, una tienda, ...

Una opción habitual para definir los vectores es dejar su tamaño en función del valor de una constante que se habrá definido previamente:

```
#define N 256
...
   float vecReal[N];
```

También se pueden definir un nuevo tipo utilizando la orden `typedef`:

typedef <**tipoBase**> <**nombreNuevoTipo**>**[TAM];**

Así, por ejemplo,

```
#define N 1000

typedef int vecInt[N];
```

define un nuevo tipo denominado `vecInt` que representa un vector de 1000 enteros. A partir del momento en que se define este nuevo tipo, se pueden declarar variables de ese tipo al igual que se hace con los tipos ya definidos del lenguaje `C`. Por ejemplo,

```
vecInt v1, v2;
```

define dos variables `v1` y `v2` cada una de las cuales contiene 1000 elementos enteros.

La opción de definir un tipo puede ser cómoda, sobre todo cuando se ha de trabajar con más de un vector; pero, además, se debe tener en cuenta una diferencia fundamental entre ambos tipos de declaraciones. Cuando se hace la declaración,

```
int contador[5];
```

se crea un vector de 5 elementos que son enteros, *se reserva memoria para 5 valores enteros* (en posiciones consecutivas, desde una dirección base). Sin embargo, al definir un nuevo tipo

```
typedef float recaudacion[7];
```

no se reserva memoria. La reserva se producirá al realizar la declaración

```
recaudacion azul, casalta;
```

que provoca que se reserve memoria para dos vectores de tipo base real, de tamaño 7.

5.2.3. Operaciones elementales de acceso en C.

Cuando se trabaja con vectores, no se pueden realizar operaciones con todo el vector, ni con partes de él. Operaciones del estilo

```
int s[10], t[10];
if (s<t)
   ...
```

o

```
s=t+5;
```

o

```
recaudacion=0;
```

no están permitidas. Es obligatorio que se realicen las operaciones *sobre cada uno de los elementos*: se puede comparar el elemento 1 de `s` con el 7 de `t`, o sumar 2 al elemento 4 de `recaudacion`, etc. Incluso en declaraciones como la siguiente

```
int v[10] = {6, 2, 5, 1, 4, 5, 6, 3, 7, 6};
```

se ha de tener en cuenta que la asignación inicial de valores se realiza elemento a elemento: hay un valor por cada elemento y en el mismo orden en que aparecen se realizarán las respectivas asignaciones (el valor 6 al primer elemento, el valor 2 al segundo, ...).

Sí que es posible pasar un vector como parámetro de funciones y procedimientos, si bien algunos lenguajes son más permisivos que otros a la hora de permitir que se utilicen como parámetros de entrada, de salida o de entrada/salida. En el caso concreto del lenguaje `C`, los vectores son *siempre parámetros de entrada/salida*. Como ejemplo,

```
float media(int N, float vecReal[]);
```

que es una cabecera habitual, en la que a una función (que, posiblemente calcule la media de los elementos de un vector) se le pasan como parámetros un entero, `N`, el tamaño del vector y el propio vector, `vecReal`. Nótese la sintaxis: se indica el tipo base, el nombre y los corchetes *sin indicar el tamaño* (esa información se debe pasar explícitamente, tal y como se ha hecho en el ejemplo). Otra posibilidad, más elegante es la siguiente:

```
#define N ...

typedef float vecReal[N];
...
float media(int N1, vecReal a);
```

Se define un tipo, `vecReal`, vector de tipo base real, con un número máximo de elementos, `N`. A la función media se le suministra como parámetros el tamaño del vector[5] y el vector, `a` de tipo `vecReal`.

[5] Puede ser o no igual a `N`; eso se decidirá al efectuar la llamada. Lo que no puede hacerse en el programa es una doble definición para N, primero como constante y después como parámetro formal de una función.

Asignación y Consulta en C.

Sólo se pueden realizar operaciones sobre elementos individuales del vector. Para poder realizar estas operaciones sobre los elementos, se hace uso de la propiedad característica de los vectores: ser un conjunto *ordenado* de datos. Por lo tanto, y tal y como se definieron las operaciones *extraer* y *almacenar*, para hacer referencia a un elemento del vector se debe indicar el nombre del vector y la posición que ocupa dicho elemento. La sintaxis de C para estas operaciones es la siguiente:

<nombre>[<expresión>]

donde <expresión> puede ser cualquier expresión cuyo resultado sea un valor entero positivo, que caiga dentro del rango de definición impuesto por el tamaño del vector. No tiene sentido acceder al elemento `5.4` de un vector, o al elemento `-3`, por poner dos ejemplos, como tampoco lo tiene intentar acceder al elemento `89` en un vector de tamaño `50`.

Ejemplos:

- `t[3]`, hace referencia al cuarto elemento[6] de `t`.
- `s[k*3+j]`, estando definidas k y j como variables enteras, hace referencia al elemento de `s` que está en la posición resultado de la expresión `k*3+j`. Este resultado, tal y como se ha definido `s`, debería estar entre 0 y 9.
- `t[-10]`, es erróneo.

Es conveniente hacer notar que un elemento individual de un vector es, a TODOS los efectos, igual que una variable simple de su mismo tipo base. Por lo tanto, para asignar un valor o consultar su contenido, se procede del mismo modo que con una variable entera, real,... con la restricción, ya comentada, de que no se puede operar con el vector entero:

- `t[5]=7;`

  asigna el valor 7 en el sexto elemento del vector `t`,

- `t=0;`

  es una asignación errónea.

- `a=t[6];`

  consulta el valor valor que hay en el séptimo elemento de `t` y se lo asigna a la variable `a` (suponiendo que está definida como una variable del mismo tipo que el tipo base del vector).

- `a=s[c*2];`

  consulta el valor que hay en elemento de `s` que ocupa la posición obtenida al calcular la expresión `c*2` (siendo `c` una variable de tipo entero y asumiendo que el resultado estará dentro del rango del vector) y se lo asigna a la variable `a` (suponiendo que hay compatibilidad de tipos).

[6] Recordad que C empieza a numerar los elementos de un vector por el 0.

Por ejemplo, las acciones descritas anteriormente quedarían de la siguiente forma:

- *"... se puede comparar el elemento 1 de* `s` *con el 7 de* `t`", podría ser el predicado

  ```
  s[1]<t[7].
  ```

  Esto es, una *consulta* de los valores de esos elementos y su posterior comparación.

- *"... se puede sumar 2 al elemento 4 de* `recaudacion`", se expresa como

  ```
  recaudacion[4]=recaudacion[4]+2.
  ```

  Es decir, se *asigna* un valor a un elemento de un vector.

Y, por supuesto, se pueden leer y escribir los elementos individuales de un vector por la entrada/salida estándar, igual que se hace con una variable:

- `scanf(" %d", &t[3]);`

  almacena en el cuarto elemento de `t` un entero leído de la entrada estándar.

- ```
  for (hoy=0;hoy<7;hoy=hoy+1)
      printf(" %f\n", recaudacion[hoy]);
  ```

  escribe en la salida estándar todos los elementos del vector `recaudacion`.

5.2.4. Cadenas de caracteres.

En el lenguaje `C` se definen las cadenas como vectores de caracteres, que tienen la particularidad de que el último elemento válido del vector (independientemente de su tamaño) es el *carácter nulo*, cuyo valor es 0. Dicho valor no debe confundirse con el carácter ASCII cero (`'0'`), sino que es el valor 0, cuyo carácter se representa en `C` como `'\0'`.

Este valor realiza la función de *centinela*, concepto que se verá nuevamente en este capítulo; es decir, sirve para indicar dónde acaba la secuencia de caracteres que forma la cadena: en ocasiones no se utilizarán todas las posiciones de la cadena y el centinela indicará cuántos elementos de la cadena se están utilizando.

Así, la declaración

```
char c1[40];
```

define una cadena que puede almacenar hasta 39 caracteres, ya que el centinela ocupa siempre un elemento de la misma, el siguiente al último significativo.

Con las cadenas se puede realizar cualquiera de las operaciones vistas para los vectores. Tiene la salvedad de que admite dos tipos de inicialización: por ejemplo,

```
char nombre[40] = { 'M', 'O', 'N', 'C', 'H', 'O' };
```

tiene el mismo efecto que
```

```
char nombre[40] = "MONCHO";
```

y la representación real de la cadena sería

0	1	2	3	4	5	6	...
'M'	'O'	'N'	'C'	'H'	'O'	'\0'	...

Ojo: Al trabajar con cadenas es preciso tener muy clara la diferencia entre `'A'` y `"A"`: las comillas dobles siempre denotan una cadena.

Algo similar ocurre cuando se trata de asignar un valor a una cadena leyendo de la entrada estándar. Además de poder leer los elementos de la cadena uno a uno como en cualquier otro vector, es posible leerlos todos en una única llamada a `scanf`. Para ello debe utilizarse la cadena de control `" %s"`. Esta cadena de control también permite escribir una cadena con una única llamada a `printf`. Por ejemplo:

```
printf(" %s\n", nombre);
```

permite imprimir el valor de la cadena `nombre` por pantalla, mientras que

```
scanff(" %s", nombre);
```

permite leer todos sus carácteres de teclado con una única lectura. Nótese que *no se utiliza ampersand*, `&`, antes del nombre de la cadena que se quiere escribir ya que se trata de un vector[7]

Además de las operaciones que le son propias por tratarse de un vector, existen una serie de funciones y procedimientos en la biblioteca estándar de `C` que permiten trabajar con cadenas. Para poder utilizarlas hay que incluir el fichero de cabecera `string.h`. Entre otras, se encuentran las siguientes:

- `strcat(char v1[], char v2[])`: concatena las cadenas `v1` y `v2`, dejando la cadena resultado en `v1`.
- `strcpy(char v1[], char v2[])`: copia la cadena `v2` en `v1`.
- `strcmp(char v1[], char v2[])`: compara la cadena `v1` y `v2`, devuelve 0 si son iguales, -1 si `v1` es menor (por orden alfabético) que `v2` y 1 si `v1` es mayor (por orden alfabético) que `v2`.
- `strlen(char v1[])`: devuelve el número de caracteres ocupados en la cadena `v1`.

5.2.5. Vectores Multidimensionales. Matrices.

Un vector, como se ha dicho, es un conjunto de elementos del mismo tipo, y éste puede ser cualquiera de los conocidos. Puede ser un tipo básico o un tipo estructurado; es decir, los elementos de un vector pueden ser enteros, reales, ... o puede ser otro vector, de forma que cada elemento del vector sea a su vez otro vector. Así se obtendría un vector de *dos dimensiones* o, como se le conoce más comúnmente, una matriz. La idea gráfica sería la siguiente:

[7] En el próximo tema se explicará completamente esta cuestión.

	0	1	2	3	4	...	M-1
0							
1							
2				*			
3							
...							
N-1							

Esta es una matriz con N filas y M columnas; el elemento marcado con $\star$ se sitúa en la tercera fila y en la cuarta columna. Intuitivamente, parece evidente que para identificarlo serán necesarios dos valores de índice: el primero, el de la fila, indica la dimensión principal (el índice del vector de tamaño N, cuyo tipo base es el vector de tamaño M). El segundo, el de la columna indica la segunda dimensión (el índice del vector de tamaño M).

En el lenguaje `C`, las matrices se definen del siguiente modo:

<tipoBase> <nombre>[TAM$_1$][TAM$_2$];

declaración cuya interpretación es similar a la declaración de un vector de una dimensión: cuál es el tipo base, con qué nombre se identifica a la matriz y cuáles son los dos tamaños de las dos dimensiones. Por supuesto, también es posible definir un nuevo tipo:

typedef <tipoBase> <nombreNuevoTipo>[TAM$_1$][TAM$_2$];

Las consideraciones sobre la reserva de memoria siguen siendo iguales: al declarar una matriz se reserva la memoria suficiente para almacenar TAM$_1 \times$TAM$_2$ elementos del tipo base correspondiente. Cuando se hace la declaración de tipo, no se reserva memoria hasta que no se declare alguna variable de ese nuevo tipo.

Para hacer referencia a un vector se necesita conocer el nombre del vector y la posición que ocupaba. Ahora, al existir dos dimensiones, las operaciones de consulta y asignación de elementos individuales deben constar del nombre de la matriz y de dos valores de índice. La sintaxis es:

<nombre>[<expresionFila>][<expresionColumna>]

donde `<expresionFila>` y `<expresionColumna>` deben evaluarse a un valor entero positivo entre `0` y `TAM`$_1$`-1` y entre `0` y `TAM`$_2$`-1`, respectivamente. Así, si se ha definido

```
int mat[N][M];
```

con `mat[2][3]` se hace referencia al cuarto elemento de la tercera fila (precisamente, el que se había marcado con un asterisco en la anterior representación gráfica de la matriz).

Como ocurre con los vectores de una dimensión, en el lenguaje `C`, las matrices también son *siempre parámetros de entrada/salida*. La sintaxis es similar a la de los vectores, pero *hay que indicar el tamaño de la segunda dimensión*:

```
float mediaMatriz(int N1, int N2, float matReal[][M]);
```

o, por supuesto, se puede optar por hacer una definición de tipo:

```
#define N ...
#define M ...

typedef float matReal[N][M];
...
float mediaMatriz(int N1, int N2, matReal a);
```

Se puede generalizar todo lo dicho para vectores a las matrices y, además, se pueden definir vectores de tantas dimensiones como se quiera. Para ello hay que especificar el tamaño de cada dimensión en la definición, tanto al declarar variables:

<tipoBase> <nombre>[TAM$_1$][TAM$_2$] ... [TAM$_n$];

como al definir nuevos tipos:

typedef <tipoBase> <nombreNuevoTipo>[TAM$_1$][TAM$_2$] ... [TAM$_n$];

5.3. Tuplas

Definición 5.2 (Tupla) *Una tupla es un conjunto finito de pares (*`id`*,*`x`*), donde* `id` *es un identificador de campo y* `x` *es el valor de dicho campo.*

En esta definición no aparece ninguna restricción respecto al tipo base de los distintos valores agrupados en una tupla. Es una estructura de datos finita y hetererogénea: permite agrupar objetos de distinto tipo, distinguiendo entre ellos mediante un identificador. Puesto que cada uno de estos objetos se suele conocer como *campo*, resulta habitual hablar de *identificadores de campo*.

Es decir, a cada uno de los campos de una tupla se le asigna un nombre que lo identifica, de forma que para hacer referencia a un elemento de una tupla *hay que conocer el identificador de la tupla y el del campo*. A diferencia de lo que ocurría en el caso de los vectores, el orden de los elementos dentro de una tupla es indiferente.

El término *tupla*, procede de las matemáticas (así se conoce a cada uno de los elementos del producto cartesiano entre dos o más conjuntos). Hemos optado por él, ante las distintas interpretaciones que pueden realizarse de las traducciones de dos de los términos más utilizados en inglés para referirse a este tipo: *record* o *struct*.

El tipo tupla se construye a partir de las siguientes operaciones elementales[8]:

[8]Formalmente, es preciso definir tantas operaciones extraer y almacenar como diferentes identificadores de campo haya en la tupla... es una de las desventajas formales de trabajar con una estructura heterogénea.

Crear cuyo resultado es una tupla con una estructura de campos definida, pero con los valores de campo no definidos,

Extraer_i que permite consultar el valor de un campo: dada una tupla `t` y un identificador `IDi`, su resultado es un valor, de tipo `TBi`, almacenado en el campo `IDi` de la tupla `t`,

Almacenar_i que permite asignar un valor a un campo: dada una tupla `t` y un identificador `IDi`, y un valor `k`, de tipo `TBi`, su resultado es un nuevo valor de `t` que se diferencia del anterior en que el valor `k` está ahora almacenado en el campo `IDi`.

5.3.1. Necesidad de las tuplas

Los vectores permiten agrupar gran cantidad de datos, pero han de ser todos del mismo tipo. Y, en el mundo real, es frecuente encontrar información compuesta por ítems de distinto tipo: por ejemplo, los datos personales de una cuenta bancaria (DNI, nombre, edad, domicilio, lugar de nacimiento, número de cuenta, ...); incluso el propio DNI está formado por información muy diversa (foto, huella dactilar, nombre, dígitos de control, ...).

Así, las tuplas se utilizan principalmente para mejorar la legibilidad de los algoritmos, ya que permiten agrupar en un único objeto información diversa, y, por lo tanto, permiten definir tipos de datos que se parecen mucho a los datos del problema que se quiere resolver.

También son de utilidad cuando se quiere agrupar pequeñas unidades de información en las que, aun siendo del mismo tipo los distintos campos, puede resultar más significativo manejar un identificador de campo que un índice. Un ejemplo típico de esta categoría es la definición de un objeto `complejo` como una tupla de dos campos, `partereal` y `parteimag`. Ambos valores son de tipo base real, lo que permitiría su definición como vector, pero parece más significativo hacer referencia a `partereal` de `complejo`, que al elemento 0 de un vector.

5.3.2. Declaración de tuplas en C.

Para declarar una tupla en el lenguaje `C`, hay que definir los elementos o campos que la forman, indicando el nombre y el tipo de cada uno de estos elementos. En `C` se hace utilizando la palabra reservada `struct` del siguiente modo:

```
struct <nombre> {
  <tipoBase_1> <nombreCampo_1>;
  <tipoBase_2> <nombreCampo_2>;
   ...
  <tipoBase_n> <nombreCampo_n>;
};
```

donde:

nombre Es el identificador de la tupla.

nombreCampo$_i$ Es el identificador del campo i.

tipoBase$_i$ Es el tipo de datos del campo i.

También es posible definir un nuevo tipo que, a partir de ese momento, se puede utilizar igual que el `int`, `float`, `char` ..., de la siguiente forma:

typedef struct {
  <tipoBase$_1$> <nombreCampo$_1$>;
  <tipoBase$_2$> <nombreCampo$_2$>;
  ...
  <tipoBase$_n$> <nombreCampo$_n$>;
} <nombreNuevoTipo>;

Ejemplos:

- La declaración

```
struct Persona {
   char nombre[80];
   int edad;
};
```

  define una tupla, `Persona`, con dos campos, el `nombre` que es un vector de caracteres y la `edad` que es un entero. También se podría hacer la definición de tipo siguiente,

```
typedef struct {
   char nombre[80];
   int edad;
} tPersona;
```

  obteniéndose el tipo `tPersona`. Cada objeto definido de este tipo, será también una tupla de dos campos.

- Con la declaración

```
struct complejo {
   float partereal;
   float parteimag;
};
```

  se define una tupla, `complejo`, con dos campos, `partereal` y `parteimag`, ambos de tipo real, mientras que,

```
typedef struct {
   float partereal;
   float parteimag;
} tComplejo;
```

  permite definir el tipo `tComplejo`.

- Como último ejemplo,

```
typedef struct {
   float x;
   float y;
   float z;
} tPunto;
```

define el tipo `tPunto` formado por tres campos de tipo real, `x,y,z` que representan las componentes de un punto.

La declaración

```
tPersona a1,a2;
tPunto p1,p2,a,b;
tComplejo x1, x2;
```

define dos variables de tipo `tPersona`, cuatro de tipo `tPunto` y dos de tipo `tComplejo`.

5.3.3. Operaciones elementales de acceso en C.

Una tupla está formada por un conjunto de campos que forman parte de una entidad superior que los engloba. Por lo tanto, para hacer referencia a un campo de una tupla, se ha de indicar la tupla a la que pertenece. En `C` se indica el nombre de la tupla, seguido del operador "`.`", seguido del nombre del campo:

<**nombreTupla**>.<**nombreCampo**>

Asignación y Consulta en C.

Al igual que ocurre al considerar elementos individuales de un vector, un campo de una tupla es, a todos los efectos, igual que una variable. Por lo tanto, un campo de una tupla se puede utilizar de la misma forma que una variable.

Al presentar las operaciones básicas de asignación y consulta para los vectores, se hizo hincapié en que tales operaciones siempre se hacen con elementos individuales. Algo similar ocurre con las tuplas, las operaciones se hacen sobre campos individuales, pero con una salvedad: sí se puede realizar una asignación de la tupla completa, siempre que se haga la *asignación se haga entre tuplas con la misma estructura de campos* (o tuplas de un mismo tipo definido). Es decir, mientras que no es posible realizar la asignación,

```
int s[10], t[10];
  ...
s=t;
```

sí que es posible realizar la siguiente:

```
tComplejo x1, x2;
  ...
x1=x2;
```

cuyo efecto sería igual al de ejecutar las asignaciones individuales

```
x1.partereal=x2.partereal;
x1.parteimag=x2.parteimag;
```

El lenguaje C permite utilizar tuplas como parámetros de una función o procedimiento, y puede definirse tanto como un parámetro de entrada, como de entrada/salida, de la misma forma que se hace con otro parámetro de cualquiera de los tipos básicos[9].

Ejemplos de Asignación:

- La asignación

  ```
  a1.edad=16;
  ```

  almacena un 16 en el campo `edad` de la tupla `a1` que es del tipo `tPersona`,

- La asignación

  ```
  p1=(0,0,0);
  ```

  es errónea.

- La secuencia

  ```
  p1.x=0;
  p1.y=1;
  p1.z=7.2;
  ```

  almacena un 0 en el campo `x` de la variable `p1`, un 1 en el campo `y` y un 7.2 en el campo `z` de la misma variable.

Ejemplos de Consulta:

- La siguiente instrucción

  ```
  i=a2.edad;
  ```

  guarda en `i` (suponiendo que está definida como una variable de tipo `int`) el valor que hay en el campo `edad` de la tupla `a2` que es del tipo `tPersona`.

- Un ejemplo de acceso a un elemento de un vector que es uno de los campos de una tupla: la asignación

  ```
  c=a1.nombre[j*2];
  ```

  guarda en `c` (suponiendo que está definida como una variable de tipo `char`) el carácter que hay en el elemento que ocupa la posición que da como resultado `j*2` (siendo `j` una variable de tipo `int`) del campo `nombre` de `a1`. De nuevo el valor de `j*2` se debe evaluar a un valor entre 0 y 79.

- Por último, y como ya se ha comentado

  ```
  p1=p2;
  ```

  copia todos los campos de `p2` en los respectivos campos de `p1`.

[9] En el primer caso, cuando se llama a la función se realiza una copia completa de todos los campos de la tupla (del parámetro real al parámetro formal). En el segundo caso, se le pasa la dirección de memoria de la tupla. Por eso, en ocasiones, aunque el parámetro deba ser sólo de entrada se define como de E/S por motivos de eficiencia.

Ejemplos de Entrada/Salida:

- Para leer el valor de un campo, se procede como con cualquier variable definida del mismo tipo:

  ```
  scanf("%d", &a2.edad);
  ```

  almacena en en el campo `edad` de la tupla `a2` un entero leído de la entrada estándar,

- Lo mismo ocurre cuando se desea escribir un valor. Mediante el bucle

  ```
  for (i=0;i<40;i=i+1)
    printf("%c",a1.nombre[i]);
  ```

  por ejemplo, se escribe en la salida estándar los primeros 40 caracteres del campo `nombre` de la variable `a1`.

5.3.4. Combinación de distintos tipos

Cualquier combinación de los tipos anteriores es válida, siempre y cuando contribuyan a modelizar correctamente los datos. Las posibilidades son muchas; en esta subsección, se comenta cuáles suelen ser las más usuales.

- **Tuplas con vectores**

  Los campos de una tupla pueden ser de cualquier tipo: enteros, reales, ... y, por supuesto, vectores. Ya se ha mostrado un ejemplo, el tipo `tPersona`, en el que uno de los campos era un vector, el nombre de la persona.

  Se podría pensar en una estructura de datos para un alumno, que constara de su nombre, edad, créditos superados y un campo `notas_primero` que fuera un vector con las notas de cada una de las asignaturas de primer curso. Ese sería un ejemplo de una tupla con dos campos de tipo vector. También podría pensarse en un ejemplo similar, en el que el campo `notas` almacenase todas las calificaciones de la carrera, dispuestas en una matriz en la que el primer índice denotase el curso y el segundo, una asignatura de ese curso.

- **Tuplas con tuplas**

  Un campo de una tupla puede ser a su vez una tupla. Como muestra, se propone el siguiente ejemplo,

  ```
  typedef struct {
      tPunto p1;
      tPunto p2;
      tPunto p3;
      tPunto p4;
  } tRectangulo;
  ```

  que define una tupla con 4 campos que a su vez son tuplas formadas por tres campos cada una. Si se realiza la declaración

  ```
  tRectangulo r1;
  ```

  entonces, la expresión `r1.p3.x` permite acceder al campo `x` del campo `p3` de la variable `r1`.

- **Vectores de tuplas**

  En la sección 5.2.5 se ha visto que los componentes de un vector pueden ser de cualquier tipo, incluidas las tuplas. Mediante la declaración

  ```
  tPersona alumnos[80];
  ```

  se define un vector, `alumnos`, en el que cada componente es del tipo `tPersona`.

  Así, `alumnos[10].edad` hace referencia al campo `edad` del alumno que está en la posición 10, `alumnos[0].nombre` hace referencia al campo `nombre` del primer alumno y `alumnos[1].nombre[0]` hace referencia a la inicial del campo `nombre` del segundo alumno.

5.4. Algoritmos de manipulación de las estructuras de datos

Esta sección tiene como objetivo presentar dos esquemas generales, *recorrido* y *búsqueda*, que son válidos para cualquier secuencia de datos. No se intenta mostrar algoritmos concretos para resolver un determinado problema, sino presentar esquemas generales que pueden servir como guía para desarrollar soluciones particulares, siempre y cuando sean convenientemente adaptados.

Se van a aplicar a los elementos almacenados en un vector, pero en temas posteriores podrán generalizarse para cualquier otra estructura lineal.

5.4.1. Esquema de Recorrido

El esquema de recorrido se utiliza cuando se pretende realizar el mismo tratamiento sobre los elementos de un vector o sobre una parte particular de ellos. El caso que se desarrollará, sin pérdida de generalidad, es aquel en el que se aplica a todos los elementos (ya que, para tratar un subrango del vector, basta con cambiar el valor inicial y/o final del índice de recorrido).

El último ejemplo de la sección 5.2.3, ejemplo de entrada/salida de vectores, era, de hecho, un ejemplo muy simple de esquema de recorrido: se pretende visualizar mediante la salida estándar *TODOS* los elementos de un vector. Mientras que el último ejemplo de la sección 5.3.3, ejemplo de entrada/salida del campo `nombre` de una tupla, muestra como visualizar *SÓLO* los 40 primeros caracteres.

El esquema genérico de recorrido es el siguiente:

```
preparar secuencia;
tratamiento inicial;
while (! fin(secuencia)){
    tratamiento del elemento;
    avanzar secuencia;
}
tratamiento final;
```

donde

- `preparar secuencia` consistirá en darle el valor inicial a la variable que se utilizará para referenciar los elementos del vector (*variable índice*); por ejemplo, `i=0`.
- `tratamiento inicial` son las acciones que se deban realizar, si las hay, antes de iniciar el recorrido; por ejemplo, iniciar un acumulador.
- `fin(secuencia)` indicará cuándo se ha llegado al final del bucle. Típicamente serán condiciones del estilo `i==TAM` o `i>=TAM`, donde `i` es la *variable índice* y `TAM` indica el número de elementos del vector.
- `tratamiento del elemento` serán todas las operaciones que se quieran realizar con cada uno de los elementos del vector. Si se asume que el recorrido se realiza sobre un vector denominado `s`, este tratamiento se realizará sobre `s[i]`. Dado que `i` irá tomando todos los valores entre `0` y `TAM-1`, la operación se ejecutará sobre todos los elementos del vector.

  Por ejemplo, `s[i]=0;` asigna el valor `0` al elemento `i` del vector `s`. Después de ejecutarse todo el esquema, todos los elementos del vector `s` valdrán `0`.
- `avanzar secuencia` hace que la *variable índice* apunte al siguiente elemento del vector. Normalmente tendrá la forma `i=i+1`.
- `tratamiento final` son las acciones que se deban realizar, si las hay, al terminar el recorrido.

Ejemplo: Se quiere duplicar el valor de todos los elementos del vector `s`, de tamaño 10 y tipo base numérico.

```
1 i=0;                              /* preparar secuencia */
2                                   /* tratamiento inicial, no hay */
3 while (!(i>=10)){                 /* fin(secuencia) --> i==10 */
4   s[i]=s[i]*2;                    /* tratamiento del elemento */
5   i=i+1;                          /* avanzar secuencia */
6 }
7                                   /* no hay tratamiento final */
```

En muy habitual que el esquema de recorrido se realice mediante un bucle `for`, ya que resulta muy cómodo: permite integrar "preparar secuencia", "fin(secuencia)" y "avanzar secuencia" en la misma línea. El mismo ejemplo quedaría de la siguiente forma:

```
for(i=0;i<10;i=i+1)
  s[i]=s[i]*2;
```

Para recorrer toda una matriz, se deben utilizar dos bucles, uno para recorrer las filas, y el otro para recorrer las columnas. Si se suponen declaradas dos variables, `n` y `m`, asociadas a la matriz y que indican el número de filas y columnas que se quieren recorrer, entonces

```
for(i=0;i<n;i=i+1)
  for(j=0;j<m;j=j+1)
    mat[i][j]=0;
```

asigna el valor `0` a todos los elementos de la matriz. En el ejemplo se ha aplicado un esquema de recorrido por filas (el índice de fila varía más lentamente que el índice de columna). Esto es habitual en los lenguajes de programación que, como el lenguaje `C`, almacenan las matrices en memoria por filas. Un recorrido por columnas como el siguiente,

```
for(j=0;j<m;j=j+1)
  for(i=0;i<n;i=i+1)
    mat[i][j]=0;
```

es tan válido como el recorrido por filas, pero suele redundar en una menor eficiencia (un algoritmo más lento), especialmente si la matriz a recorrer es de gran tamaño.

En el esquema de recorrido presentado, se ha asumido que se conocía el tamaño del vector (o el de la matriz). Conviene recordar que, como ya se ha comentado, los vectores son estructuras estáticas: se define su tamaño al inicio del algoritmo y éste no varía[10]. La cuestión que puede surgir es que, seguramente, no siempre se van a ocupar con datos tantas posiciones de memoria como se haya reservado al definir el vector. Por ejemplo, cuando se ha definido un vector con capacidad para almacenar las notas de una asignatura de hasta 50 alumnos, pero, en realidad sólo hay 38 matriculados. ¿Cómo se indica entonces cuántos elementos son válidos en el vector? Existen tres soluciones:

Se utilizan siempre todos los elementos El algoritmo se diseña de forma que es obligatorio trabajar siempre con todos los elementos del vector. El problema de este planteamiento es que no es general y es poco operativo en la práctica.

Se utiliza una variable asociada al vector que indica su tamaño Esta variable en todo momento indica cuántos elementos del vector son válidos. Cada vez que se introducen nuevos valores o se eliminan valores se actualiza su valor. La forma general de trabajar consiste en considerar válidos los elementos entre el primero y el indexado por la variable asociada.

Se utiliza un centinela Es un valor que no puede tomar ningún elemento del vector y que se almacena después de la última posición ocupada realmente. Es decir, indica el fin de los elementos válidos. Si se añaden elementos, se desplaza hacia la izquierda (el índice del centinela se incrementa), si se eliminan, se desplaza hacia la derecha (el índice del centinela se decrementa).

Por lo tanto, el predicado genérico `fin(secuencia)` se tendrá que diseñar de acuerdo a la forma en la que se ha definido el número de elementos válidos. La tabla siguiente indica tres predicados típicos de evaluación, dependiendo de cuál de los tres criterios anteriores es el utilizado y asumiendo que `i` es la *variable índice* :

Con todo el vector	`i>=TAM`	siendo `TAM` el tamaño del vector
Con variable asociada	`i>=n`	siendo `n` la variable asociada al vector
Con centinela	`v[i]==centinela`	siendo `v` el vector, y `centinela` el valor utilizado como tal

[10]Esto, aunque pueda parecer un inconveniente, les da mucha eficiencia en rapidez de acceso a los elementos en memoria cuando se transforma el algoritmo en un programa. Existen otras estructuras, denominadas dinámicas, a las que se les puede cambiar el tamaño durante la ejecución del programa.

Ejemplo: Se desea hacer una función para calcular el número de alumnos que están por debajo de la nota media de la clase, suponiendo que se dispone de una variable asociada que indica el número de elementos válidos.

```
/* Pre: notas=NOTAS es un vector de reales y n=N>0 */
int alBajoMed(float notas[], int n) {
  int i,contador;
  float med;

  i=0;                  /* preparar secuencia */
  med=media(notas,n);   /* tratamiento inicial */
  contador=0;

  while (!(i==n)){      /* fin(secuencia) <-> (i==n) */
    if (notas[i]<med)   /* tratamiento del elemento */
      contador=contador+1;
    i=i+1;               /* avanzar secuencia */
  }
                        /* tratamiento final, no hay */
  return contador;
}
/* Post: notas=NOTAS y n=N y alBajoMed(notas, n) devuelve la
cantidad  de los n elementos primeros de notas que están por
debajo de la media de los N primeros elementos */
```

En este ejemplo, se ha tomado como predicado de fin de secuencia `i==n`, lo cual es correcto ya que el contador se incrementa de 1 en 1 y por tanto en algún momento la variable `i` alcanzará el valor `n`, momento en el que el recorrido habrá finalizado. En los casos en que el contador del bucle se incrementa con valores mayor a la unidad es preferible utilizar como predicado de fin de secuencia `i>=n`; esto supone utilizar un bucle `while` gobernado por la condición contraria, `while (i<n)`. De esta forma, finalizará el recorrido en cuanto se supere el valor de `n`.

La función `media`, que se muestra a continuación, también se puede resolver aplicando el esquema de recorrido, esta vez utilizando un bucle `for`:

```
/* Pre: v=V es un vector de reales y n=N */
float media(float v[], int n) {
  int i;
  float suma,m;

  suma=0;               /* tratamiento inicial */

  for (i=0;i<n;i++){    /* preparar, fin y avanzar secuencia */
     suma=suma+v[i];    /* tratamiento del elemento */
  }
  m=suma/n;             /* tratamiento final */
  return m;
}
/* Post: v=V y n=N y media(v,n) devuelve la media de los
primeros N valores del vector V */
```

5.4.2. Esquema de Búsqueda

Este esquema se utiliza cuando se quiere determinar si existe algún elemento en el vector que cumpla alguna condición determinada, o cuando se quiera aplicar un tratamiento a los elementos del vector hasta encontrar que se satisface una condición establecida. De nuevo, el esquema se puede aplicar a cualquier secuencia de datos, a cualquier estructura lineal, aunque aquí el estudio se centra en la utilización con vectores.

```
preparar secuencia;
encontrado=FALSO;
tratamiento inicial;
while ((!fin(secuencia) && (!encontrado)) {
    actualizar(encontrado);
    if (!(encontrado)) {
        tratamiento del elemento;
        avanzar secuencia;
    }
}
tratamiento final;
```

La principal diferencia con el esquema de recorrido es que ahora se dispone de una variable, `encontrado`, que puede hacer que finalice el bucle, y por lo tanto el tratamiento de los elementos, antes de alcanzar el fin del vector (en el caso general, de la secuencia de datos que se esté tratando.

Con respecto al esquema de recorrido sólo aparece una nueva operación:

actualizar(encontrado) que actualiza la variable que indica si hay que finalizar o no la búsqueda. Ejemplos típicos de esta función son: `v[i]==valor` si se está buscando un determinado `valor` en el vector, `v[i]==v[0]` si se está buscando en el vector un elemento cuyo valor sea igual al del primer elemento, `v[i]<0` si se está buscando un valor negativo, etc.

Ejemplo: Determinar si una cadena de caracteres es un palíndromo, es decir, se lee igual de izquierda a derecha que de derecha a izquierda.

La resolución del problema se puede plantear como una búsqueda de un elemento de la cadena que sea distinto de su simétrico. Si se encuentra un elemento que cumpla esta condición, se puede afirmar que la cadena no es un palíndromo, aunque no se haya recorrido completamente.

```
1  typedef enum {FALSO, VERDADERO} boole;
2
3  /* Pre: c=C es una cadena de caracteres, acabada en '\0' */
4  boole palidromo(char c[]) {
5    int i,n;
6    boole encontrado;
7
8
9    i=0;                              /* preparar secuencia */
10   n=strlen(c);
11   encontrado=FALSO;
12                                     /* tratamiento inicial, no hay */
```

```
13    while (i<n/2 && !(encontrado)){ /* fin(secuencia) <--> (i>=n/2) */
14      encontrado=(c[i]!=c[n-i-1]);  /* actualizar(encontrado); */
15      if (!(encontrado)){
16                                    /* tratamiento del elemento, no hay */
17         i=i+1;                     /* avanzar secuencia */
18       }
19    }
20    return !(encontrado);  /* Si no se encuentra, es palindromo */
21  }
```

Una vez aplicado el esquema, se puede refinar el algoritmo por motivos de eficiencia. En este ejemplo, puesto que no hay tratamiento, se podría eliminar la línea 15 y hacer que se incrementara el valor de `i` de forma incondicional (tal y como está, indicaría en qué posición se ha detectado que los caracteres son distintos, pero esa información realmente no se ha pedido).

5.5. Resumen y Consideraciones de Estilo.

Los tipos básicos, predefinidos en el lenguaje, no bastan para procesar eficientemente la información. Las estructuras de datos son una herramienta que permite que la información del mundo real se pueda almacenar de forma significativa y, además, la buena elección de esta representación redunda en la construcción de algoritmos más eficientes y sencillos.

Sin embargo, la elección de la estructura de datos es sólo el principio del diseño de un *tipo de datos*, concepto que engloba también las operaciones propias para manipular y procesar la estructura.

En este tema se han introducido dos de los tipos estructurados más básicos, vectores y tuplas. A lo largo del tema se han comentado las operaciones básicas de manipulación de estas estructuras, crear, extraer y almacenar, que suelen ser suministradas por el lenguaje. Pero que estos tipos sean *realmente* tipos de datos, dependerá en buena medida de la capacidad del programador para *enriquecerlos* y dotarlos realmente de un buen conjunto de operaciones básicas de manipulación.

Para entender el concepto de *enriquecimiento* del tipo, tómese como ejemplo el tipo entero; por supuesto, se puede crear una variable de tipo entero (cuando se declara) y se puede consultar su valor y realizar asignaciones sobre ella. Pero, además, se pueden sumar dos objetos enteros, restarlos, multiplicarlos, dividirlos, se pueden leer, se pueden escribir... Se puede realizar gran número de operaciones ya definidas en el lenguaje. Sin embargo, cuando se define un tipo vector de reales el lenguaje sólo permite las tres operaciones básicas ya mencionadas. ¿Quiero esto decir que los vectores no se pueden sumar, restar, leer... ? No directamente (usando una única operación), salvo si el programador dota a la estructura del adecuado *conjunto de operaciones* que permitan manipularla de una forma similar a como se manipula un entero. Es decir, con las herramientas de las que se dispone hasta el momento, una idea que podría surgir de este tema es la definición de un módulo de vectores reales, por ejemplo, en donde se realizara una definición de estructura y una definición de operaciones comunes (lectura, escritura, suma, valor medio, etc.).

La primera ventaja de este módulo sería la reutilización del código. La segunda ventaja es menos evidente y de carácter teórico: permite aproximarse a dos temas que habrá que reforzar en cursos posteriores,

- el concepto de *Tipo Abstracto de Datos*, es decir, de la definición teórica de un tipo mediante la definición de sus operaciones, sin pensar en cómo se implementa, sólo cuál deber ser su comportamiento, y

- el paradigma de la *Programación Orientada a Objetos*, cuya práctica, basada en la identificación de ítems de información y cómo manipularlos, dota al programador de herramientas que permiten garantizar la generidad y, además, dotarla de la privacidad necesaria para garantizar su uso correcto (en el tema 8, se ampliarán estas ideas básicas sobre POO).

La contribución de este tema en el cumplimiento de los objetivos marcados en el tema 1, se plasma especialmente en el paso 2, de *planificación de la solución*: una buena identificación de la estructura de datos más adecuada para modelizar una información, y la identificación de las operaciones que la manejen, suele ser el primer paso en la resolución de problemas complejos. Ya se conocía el mecanismo de definición de acciones no primitivas; ahora, se han presentado las estructuras básicas y, además, dos esquemas generales (recorrido y búsqueda) muy utilizados en su proceso.

Por supuesto, en este proceso de definición de nuevos tipos, no deben dejarse de lado los aspectos habituales en cuanto a la definición de un buen estilo: es conveniente al definir el tipo, documentarlo adecuadamente, comentando no sólo las operaciones que se definan, sino también la estructura elegida para implementarlo.

5.6. Glosario.

centinela Valor utilizado para indicar el final de los datos reales de un vector (de una o más dimensiones). Normalmente, es un valor predefinido, que se elige con la seguridad de que ningún otro elemento del vector tendrá dicho valor.

dimensión principal En un vector de dos dimensiones, es la dimensión que indica el criterio mediante el cual se almacena la estructura en la memoria, por filas o por columnas. En el lenguaje `C` la dimensión principal es la asociada al índice de filas (*row major*). Puede condicionar la eficiencia de un recorrido. Se puede generalizar el concepto a más de dos dimensiones.

estructura de datos lineal Estructura de datos en las que los elementos se disponen de forma que hay sólo un elemento anterior y sólo un elemento posterior a un elemento dado. Por ejemplo, un vector.

indexar Identificar cada elemento de un vector mediante un índice, de forma unívoca: elementos distintos tienen distinto índice, distintos índices denotan elementos distintos.

rango Conjunto de valores entre los que puede variar el índice de un vector (de una o más dimensiones). El rango típico en `C` es `0..TAM-1`, siendo `TAM` el tamaño del vector.

tipo predefinido (o básico o elemental) Tipo (estructura + operaciones) completamente definido en un lenguaje de programación. Por ejemplo, el entero, el real y el carácter en `C`.

tipo definido por el programador Tipo definido por el programador completamente, tanto la estructura de datos como las operaciones de manipulación.

tipo estructurado Tipo definido por el programador, normalmente basado en una estructura proporcionada por el lenguaje (implementación, operaciones básicas de acceso) y que el programador completa con la definición de operaciones de manipulación.

tipo base En un vector de una o más dimensiones, tipo de todos los elementos agrupados en la estructura. En una tupla, cada campo puede ser de un tipo base distinto. En ambos casos, puede ser un tipo elemental o un tipo definido por el programador.

tipo índice En un vector, tipo al que pertenece el índice que identifica a cada uno de sus elementos. En el lenguaje `C` se identifica con el tipo entero.

5.7. Bibliografía.

1. "Fonaments de Programació I", M.J. Marco, J. Álvarez & J. Villaplana. Universitat Oberta de Catalunya, Setembre 2001.
2. "Introducción a la Programación (vol. I)", Biondi & Clavel. Ed. Masson. 1991.
3. "El Lenguaje de Programación C. Diseño e Implementación de Programas". Félix García, Jesús Carretero, Javier Fernández & Alejandro Calderón. Pearson Education, Madrid 2002.
4. "The C Programming Language". Brian W. Kernigham & Dennis M. Ritchie. Prentice-Hall Inc. 1988.

5.8. Problemas Propuestos.

1. Definidos

   ```
   int i, aux, n;
   int v[10];
   ```

   realizar una traza del siguiente fragmento de código:

   ```
   for (i=0; i<n; i=i+2) {
     aux=v[i];
     v[i]=v[i+1];
     v[i+1]=aux;
   }
   ```

   sabiendo que `n=10` y `v=(6, 2, 5, 1, 4, 5, 6, 3, 7, 6).`

2. Definidos

   ```
   int i, n;
   float v[10];
   ```

   realizar una traza del siguiente fragmento de código:

   ```
   for (i=0; i<(n/2); i=i + 1) {
     v[i]=v[n-(i+1)];
     v[n-(i+1)]=v[i+1];
   }
   ```

   sabiendo que `n=10` y `v=(6.5,2.0,5.1,4.1,7.5,5.0,6.25,3.1,7.8,6.0).`

3. Dada la siguiente secuencia de instrucciones:

   ```
   cont=0;
   suma=0;
   for (i=0; i<n; i=i+1){
       if (v[i]>n1){
           if (v[i]<n2){
               cont=cont+1;
               suma=suma+v[i];
           }
       }
   }
   ```

se pide lo siguiente:

a) Convertir la secuencia anterior en una función o procedimiento, es decir, darle una cabecera, indicando cuál o cuáles son los parámetros de entrada e indicando cuál o cuáles son los resultados que devuelve.

b) Establecer la precondición, es decir, describir bajo qué condiciones dicha secuencia será correcta (funciona correctamente y produce algún resultado).

c) Establecer la postcondición, es decir, indicar qué calcula.

4. Dado el siguiente fragmento de código:

```
blancos = 0;
varios = 0;
contLetra = 0;
i=0;
while (cad[i] != '\0'){
  if ((cad[i] == car)||(cad[i] == car-32)){
    contLetra = contLetra + 1;
  }
  else if (cad[i] == ' '){
    blancos = blancos + 1;
  }
  else {
    varios = varios + 1;
  }
  i = i+1;
}
porcentaje=contLetra/i;
```

a) Convertir la secuencia anterior en una función o procedimiento, es decir, indicando qué objetos deberían ser datos, cuáles resultados y cuáles variables propias del proceso, darle una cabecera, hacer la declaración de variables y reescribir el código para asegurar que el resultado o resultados se devolverán adecuadamente.

b) Establecer la postcondición, es decir, indicar qué calcula.

c) Establecer la precondición, es decir, describir bajo qué condiciones dicha secuencia será correcta (funciona correctamente y produce resultados correctos).

5. ¿Verdadero o Falso (justificar)?:

   "El predicado `(i==0 || i==N || v[i]==c)` es un invariante del siguiente bucle:"

```
int buscar (float v[], int N, float c) {
/*Pre: v=vector d tamaño N, c=elem. a buscar*/
  int i=0;

  while (v[i]!=c && i<N)
    i++;
  return i;
}
/*Post: devuelve índice [0..N-1] de la posi-*/
/*ción de c en v si lo encuentra, o N si no */
```

6. Escribe todo lo que se podría leer en la pantalla del ordenador al ejecutar el siguiente fragmento de programa, sabiendo que:

- se han definido estos dos tipos

```
typedef struct {
   char nombre[N];
   float s1, s2, s3;
} tSaltador;
```

```
typedef struct {
   char nombre[N];
   float marca;
} tClasificado;
```

- Es posible que el siguiente procedimiento tenga instrucciones inútiles. Elimina cualquier cosa que te parezca que sobra, justificando cada cambio, de forma que quede una versión lo más simple posible del procedimiento. Y, por supuesto, que siga produciendo el mismo efecto:

```
void chungoChungo(int a[], int n) {
/* Pre: n > 0 */
/* Se supone definido el tipo boole */
   int i,j,k;
   boole chivato;
   i=0;
   while (i<n){
      chivato=falso;
      j=i+1;
      while ((j<n)&& !(chivato)){
         chivato=chivato&&(a[i]==a[j]);
         j=j+1;
      }
      if (j==n){
         a[i]=0;
      }
      else {
         for(k=j;k<n;k=k+1){
            a[k]=a[j]+a[i];
            a[k]=2*a[i];
         }
         if (a[n]<a[i]){
             a[n]=0;
         }
         else{
             a[i]=i;
         }
      }
      i=i+1;
   }
}
```

- el vector `v` es un vector de `TAM=6` elementos de tipo `tSaltador`, cuyos valores son:

nombre	"J.Lino"	"Fiona"	"Yago"	"Niurka"	"Concha"	"Iván"
s1	6.75	0	0	6.38	6.69	6.05
s2	7.1	0	7.06	0	0	6.23
s3	7.32	6.37	7.01	0	6.17	7.08
	0	1	2	3	4	5

- que `v2` es un vector, también de `TAM=6`, y con elementos del tipo `tClasificado`.

```
#include <string.h>
#define N 10
#define TAM 6
#define TOPE 6.5

... /* Las definiciones de tipo */

int main() {
  int indice,i;
```

```
  float max;
  tSaltador v[TAM];
  tClasificado v2[TAM];

  ... /* Se asignan los valores de v */

  indice=0;
  for (i=0; i<TAM; i=i+1) {
    if ((v[i].s1>=TOPE) || (v[i].s2>=TOPE) ||
                                (v[i].s3>=TOPE)) {
      printf("\n%s se clasifica,", v[i].nombre);
      strcpy(v2[indice].nombre,v[i].nombre);
      max=v[i].s1;
      if (v[i].s2 > max) {
        max=v[i].s2;
        printf(" no por el primer salto,");
      }
      if (v[i].s3 > max) {
        max=v[i].s3;
        printf(" ni por el segundo.");
      }
      v2[indice].marca=max;
      indice=indice+1;
    }
  }

  printf ("\n\nClasificados: \n");
  for (i=0; i<indice; i=i+1) {
    printf("\t>>%s, con un mejor salto de
         %5.2f m.\n", v2[i].nombre, v2[i].marca);
  }
}
```

7. Escribe todo lo que se podría leer en la pantalla del ordenador al ejecutar el siguiente fragmento de programa, sabiendo que el vector `v` es un vector de `TAM=6` elementos de tipo `tHalterofilia`,

```
typedef struct {
   char nombre[N];
   int peso;
} tHalterofilia;
```

y que los valores que se asignarán a los elementos de `v` son:

nombre	"Paco"	"Pedro"	"Ana"	"Alicia"	"Pepe"	"Luis"
peso	40	80	45	70	90	42
	0	1	2	3	4	5

```
int main() {
  int indice,i;
  tHalterofilia v[TAM];
```

```
  ... /* Se asignan los valores de v */

  indice=0;
  printf("\nEl valor de indice es %d",indice);
  for (i=1; i<TAM; i++) {
     if (v[i].peso > v[indice].peso) {
        indice = i;
        printf("\nEl valor de indice es %d",indice);
     }
  }

  printf("\nGana %s con una marca de %d kilos.\n",
     v[indice].nombre, v[indice].peso);

  printf ("\nDiferencias entre participantes: \n");
  for (i=0; i<TAM; i++) {
     if (i!=indice) {
        printf("\t>> Entre %s y %s, de %d kilos.\n",
             v[indice].nombre, v[i].nombre,
             (v[indice].peso-v[i].peso));
     }
  }
}
```

8. Escribir un algoritmo que permita sumar dos vectores de `N` elementos.

9. Escribir un algoritmo que calcule la media de los elementos de un vector real de `N` elementos.

10. Escribir un algoritmo que permita obtener el producto escalar de dos vectores.

11. Dado un vector de `N` componentes reales, diseñar un algoritmo que permita obtener su elemento máximo y otro algoritmo que permita obtener su elemento mínimo.

12. Obtener los algoritmos que, para un vector a de `N` componentes, determinen:

    a) El recorrido, r = max(a[i]) - min(a[i]), i=1.. `N`,

    b) El valor medio de los componentes de a, $\tilde{a}$,

    c) La desviación típica,

    $$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(a_i - \tilde{a}\ )^2}{n}}$$

    d) El coeficiente de variación, $\frac{\sigma}{\tilde{a}}$.

13. Alguien ha definido la función `amigos` cuyo prototipo es:

```
boole amigos(int n1, int n2);
/* pre: n1=N1, n2=N2, enteros positivos */
/* post: devuelve cierto si N1 y N2 son amigos, */
/*       falso en caso contrario */
```

    Teniendo en cuenta esta definición (además de la habitual para el tipo `boole`), indicar cuáles son los 5 errores (**no sintácticos**) cometidos al desarrollar el siguiente procedimiento, justificando para cada error el porqué: