Índice general

TEMA 4. VECTORES Y MATRICES

4.1. VECTORES

4.1.1. MOTIVACIÓN

En casi todos los problemas, es necesario mantener una relación entre variables diferentes o almacenar y referenciar variables como un grupo. Para ello, los lenguajes ofrecen tipos más complejos que los vistos hasta ahora.

Un *tipo de dato compuesto* es una composición de tipos de datos simples (o incluso compuestos) caracterizado por la *organización* de sus datos y por las *operaciones* que se definen sobre él.

Un *vector* es un tipo de dato, compuesto de un número fijo de componentes del mismo tipo y donde cada una de ellas es directamente accesible mediante un índice.

Ejemplo. Leed notas desde teclado y decid cuántos alumnos superan la media

```
#include <stdio.h>

int main(){
  int cuantos;
  float nota1, nota2, nota3, media;

  printf("Introduce nota 1: ");
  scanf("%f", &nota1);
  printf("Introduce nota 2: ");
  scanf("%f", &nota2);
  printf("Introduce nota 3: ");
  scanf("%f", &nota3);

  media = (nota1+nota2+nota3)/3.0;
  cuantos=0;

  if (nota1 > media)
    cuantos++;
  if (nota2 > media)
    cuantos++;
  if (nota3 > media)
    cuantos++;

  printf("Media Aritmetica = %f\n", media);
  printf("%d alumnos han superado la media\n", cuantos);
}
```

***Problema:* ¿Qué sucede si queremos almacenar las notas de 50 alumnos? Número de variables imposible de sostener y recordar.**

***Solución:* Introducir un tipo de dato nuevo que permita representar dichas variables en una única *estructura de datos*, reconocible bajo un nombre único.**

	notas[0]	notas[1]	notas[2]
notas=	2.4	4.9	6.7

4.1.2. Declaración y Representación en Memoria

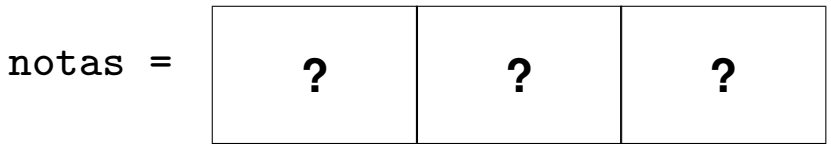
```
<tipo> <identificador>[<N.Componentes>];
```

- ▷ `<tipo>` **indica el tipo de dato común a todas las *componentes* del vector.**
- ▷ `<N.Componentes>` **determina el número de componentes del vector, al que llamaremos *dimensión* del vector. El número de componentes debe conocerse a priori y no es posible alterarlo durante la ejecución del programa. Pueden usarse literales o defines en mayúsculas en C++ también pueden usarse constantes enteras (char, int, ...), pero nunca una variable**

No puede dimensionarse con una variable

- ▷ **Las componentes ocupan posiciones contiguas en memoria.**

```
float notas[3];
```



- ▷ ***Consejo*: Fomentad el uso de defines para especificar la dimensión de los vectores.**

```
#define TOTAL_ALUMNOS 100

int main(){
   float notas[TOTAL_ALUMNOS];
}
```

Otros ejemplos:

```
#define NUM_REACTORES 20
int main(){
  int longitud=50;

  int TemperaturasReactores[NUM_REACTORES];  /* Correcto. */
  int casados[40];                           /* Correcto. */
  char  NIF[8];                              /* Correcto. */
  int  vector[longitud];          /* Error en compilación. */
  ..............
}
```

4.1.3. Operaciones con Vectores

4.1.3.1. Acceso

Dada la declaración:

<tipo> <identificador>[<N.Componentes>];

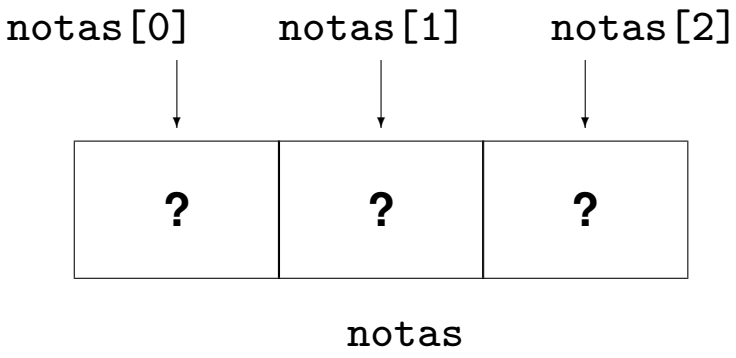
Cada componente se accede de la forma:

<*Identificador de vector*> [<*índice*>]

$$0 \leq \textit{índice} < \texttt{N.Componentes}$$

▷ **El índice de la primera componente del vector es 0.**
El índice de la última componente es <N.Componentes>-1**.**

```
float notas[3];
```



`notas[9]` **y** `notas['3']` **no son componentes correctas.**

▷ **Cada componente *es una variable* más del programa, del tipo indicado en la declaración del vector. Por ejemplo, si declaramos**

```
float vector[3];
```

entonces `vector[2]` **y** `vector[0]` **son variables de tipo** `float`

Por lo tanto, con dichas componentes podremos realizar todas las operaciones disponibles (asignación, lectura con `scanf`**, pasarla como parámetros actuales, etc). Lo detallamos en los siguientes apartados.**

4.1.3.2. ASIGNACIÓN

▷ **No se permiten asignaciones globales sobre todos los elementos del vector. Las asignaciones se deben realizar componente a componente.**

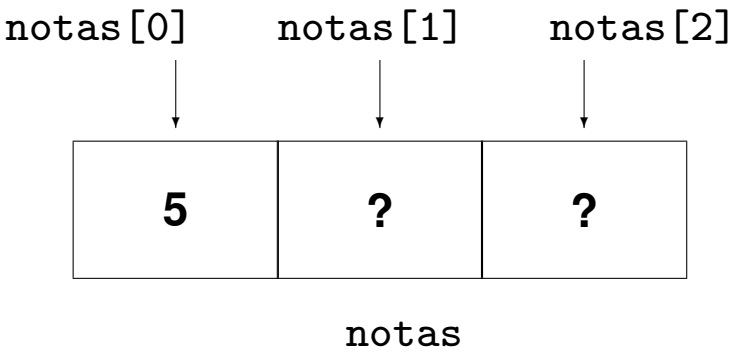
```
int main(){
   float notas[3];

   notas = <lo que sea>  /* Error de compilación */
```

▷ **Asignación componente a componente:**

$$<\textit{Ident. vector}>\ [\ <\textit{índice}>\ ] = <\textit{Expresión}>;$$

$<$*Expresión*$>$ ha de ser del mismo tipo que el definido en la declaración del vector (o al menos *compatible*).

```
int main(){
   float notas[3];

   notas[0] = 5;

   int i = 0;
   notas[i] = 5;
```



```
#define DIM_VECTOR 10
int  main(){
    float vector[DIM_VECTOR], dato;
    int i;

    dato = vector[0];
    printf("%f", dato);    /* Error lógico. Contiene basura */

    for (i=0 ; i<DIM_VECTOR ; i++)
        vector[i] = 2*i;                  /* <- float = int */

    dato = vector[0];                     /* Correcto */
    vector[15]=7;      /* Posible error grave en ejecución! */
 }
```

El compilador no comprueba que el índice de acceso a las componentes esté en el rango correcto, por lo que cualquier modificación de una componente inexistente tiene consecuencias imprevisibles.

4.1.3.3. LECTURA Y ESCRITURA

La lectura y escritura se realiza componente a componente.

Para leer con `scanf` o escribir con `printf` *todas* las componentes utilizaremos un bucle, como por ejemplo:

```
#include <stdio.h>
#define NUM_NOTAS 10
int  main(){
   float notas[NUM_NOTAS], media;
   int i;

   for (i=0; i<NUM_NOTAS; i++){
      printf("Introducir nota del alumno %d: ", i);
      scanf("%f", &notas[i]);
   }

   media = 0;

   for (i=0; i<NUM_NOTAS; i++){
      media = media + notas[i];
   }

   media = media / NUM_NOTAS;

   printf("\nMedia = %f", media);
}
```

4.1.3.4. TRABAJANDO CON EL VECTOR

No es necesario utilizar todas las componentes del vector. Los huecos suelen dejarse en la zona de índices altos. Por eso, el programador debe usar una variable entera, `util` **que indique el número de componentes usadas. Como convención, usaremos identificadores del tipo** `utilNombreDelVector`

Los índices de las componentes utilizadas irán desde 0 hasta `utilNombreDelVector-1`

```
#include <stdio.h>
#define DIM_NOTAS 100

int main(){
   int util_notas, i;
   float notas[DIM_NOTAS], media;

   do{
     printf("Introduzca el número de alumnos (entre 1 y %d): ",
             DIM_NOTAS);
     scanf("%d", &util_notas);
   }while (util_notas<1 || util_notas>DIM_NOTAS);

   for (i=0; i<util_notas; i++){
      printf("nota[%d] --> ", i);
      scanf("%f", &notas[i]);
   }

   media=0;
   for (i=0; i<util_notas; i++){
      media=media+notas[i];
   }

   printf("\nMedia: %f\n", media/util_notas);
   return 0;
}
```

Ejemplo. Buscar un elemento en un vector.

***Descripción*:**

```
Recorre las componentes vector[i] del vector
mientras no se terminen Y
mientras no se encuentre el elemento
```

```
#include <stdio.h>
#define DIM_VECTOR 100

int main(){
  float vector[DIM_VECTOR], buscado;
  int i, util_vector;
  int encontrado;

  util_vector = 50;
  for (i=0 ; i<util_vector ; i++)
     vector[i] = i*i;


  printf("\n\nIntroduzca valor a buscar");
  scanf("%f", &buscado);
```

```
  encontrado=0;
  i=0;

  while ((i<util_vector) && !encontrado)
     if (vector[i] == buscado)
        encontrado = 1;
     else
        i++;

  if (encontrado)
     printf("\nEncontrado en la posición %d", i);
  else
     printf("\nNo encontrado");
}
```

```
  encontrado=0;

  for (i=0; i<util_vector && !encontrado ; i++)
     if (vector[i] == buscado)
        encontrado = 1;

  if (encontrado)
     printf("\nEncontrado en la posición %d", i-1); /*:-( */
  else
     printf("\nNo encontrado\n");
}
```

```
  encontrado=0;

  for (i=0; i<util_vector && !encontrado ; i++)
     if (vector[i] == buscado){
        posicion = i;
        encontrado = 1;
     }

  if (encontrado)
     printf("\nEncontrado en la posición %d", posicion);
  else
     printf("\nNo encontrado\n");
}
```

Cuando usemos el índice de un vector como variable contadora de un bucle for, recordad que automáticamente se incrementa en cada iteración.

Ejercicio. **Invertir el contenido de un vector.**

`(2,-3,5,6,?,?)` $\longrightarrow$ `(6,5,-3,2,?,?)`

4.1.3.5. INICIALIZACIÓN

C permite inicializar una variable vector en la declaración.

```
int vector[3]={4,5,6};
```

inicializa `vector[0]=4, vector[1]=5, vector[2]=6`

```
int vector[7]={3,5};
```

inicializa `vector[0]=3, vector[1]=5` **y el resto se inicializan a cero.**

```
int vector[7]={0};
```

inicializa todas las componentes a cero.

```
int vector[7]={8};
```

inicializa la primera a 8 y el resto a cero.

```
int vector[]={1,3,9};
```

automáticamente el compilador asume `int vector[3]`

4.1.4. MODULARIZACIÓN Y VECTORES

4.1.4.1. PASANDO INDIVIDUALMENTE UNA COMPONENTE COMO PARÁMETRO

Las componentes de un vector son variables cualesquiera por lo que las podemos pasar como parámetro actual a cualquier función.

Ejemplo. **De un vector de `char`, imprimid la mayúscula correspondiente a la primera componente.**

```
char palabra[10];

palabra[0]='h';
palabra[1]='o';
palabra[2]='l';
palabra[3]='a';

printf("%c", toupper(palabra[0]));
```

Obviamente, también podemos pasar por referencia una componente.

Ejemplo. **De un vector de `int`, incrementad en 1 el valor de todas las componentes.**

```
void Incrementa(int *dato){
   (*dato) = (*dato)+1;
}

int main(){
   int vector[10], int util_vector=5;
   int i;

   /* Asignación de valores a las componentes */

   for(i=0; i<util_vector; i++)
      Incrementa(&vector[i]);
}
```

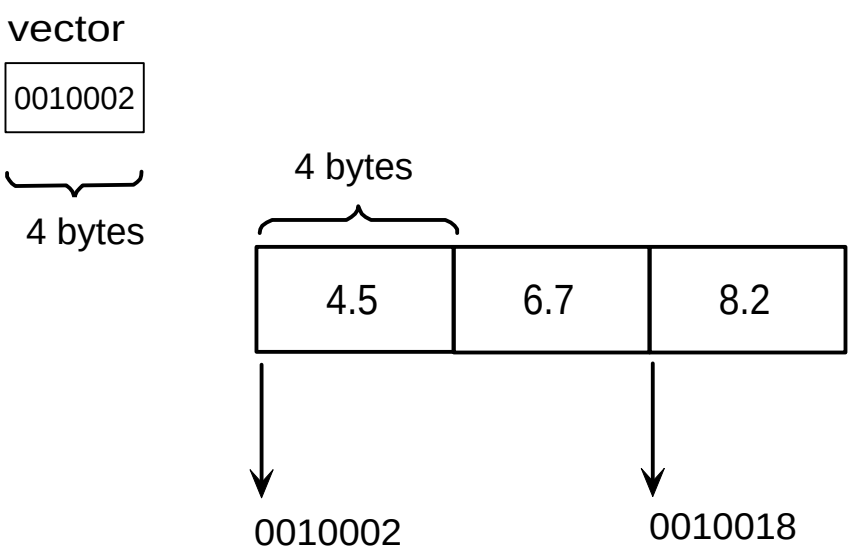
Ejercicio. **Reescribid el ejemplo de invertir un vector usando una función para intercambiar dos datos de tipo `float`**

4.1.4.2. Pasando todas las componentes como parámetro

Para entender el paso de un vector como parámetro debemos ver con detalle cómo maneja el compilador un vector.

El compilador trata a un vector como un dato constante que almacena la dirección de memoria dónde empiezan a almacenarse las componentes. En un SO de 32 bits, para guardar una dirección de memoria, se usan 32 bits (4 bytes).

```
int main(){
   float vector[3];

   vector[0] = 4.5;
   vector[1] = 6.7;
   vector[2] = 8.2;
```



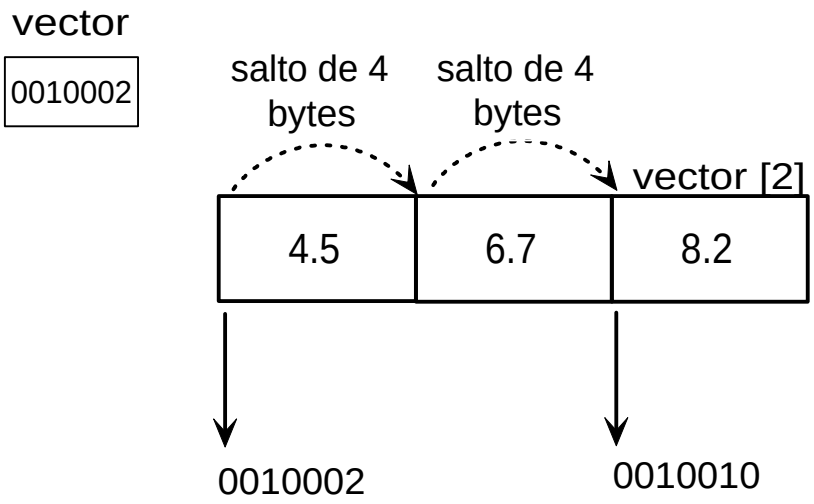
Es como si el programador hiciese la asignación siguiente:

```
vector = 0010002;
```

Realmente, dicha asignación es realizada por el compilador que declara `vector` como un dato constante:

```
int main(){
   float vector[3];   /* Compilador -> vector = 0010002 */

   vector = 00100004;  /* Error Comp. vector es cte. */
   vector[0] = 4.5;    /* Correcto */
```

Para saber dónde está la variable `vector[2]`, el compilador debe dar dos *saltos*, a partir de la dirección de memoria 0010002. Cada salto es de 4 bytes (suponiendo que los `float` ocupan 4 bytes). Por tanto, se va a la variable que empieza en la dirección *0010010*

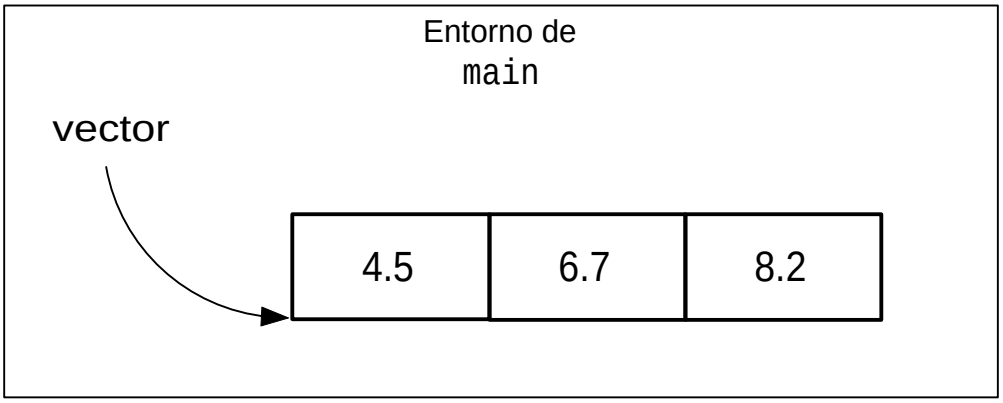


En general, si declaramos

```
TipoBase vector[DIMENSION];
```

para poder acceder a `vector[indice]`, el compilador dará un total de `indice` saltos tan grandes como diga el `TipoBase`, a partir de la primera posición de `vector`.

Para reflejar gráficamente que un vector *apunta* a una dirección de memoria, usaremos una flecha en la forma siguiente:



Ahora podemos responder a la siguiente pregunta ¿Podemos pasar de golpe todas las componentes a una función?

- ▷ **NO puede pasarse una copia *de golpe*, de todas las componentes del vector**
- ▷ **Pero sí podemos pasar una copia de la dirección de memoria que contiene el vector (*0010002* en el ejemplo). Para indicarlo, se pone `[]` en el parámetro formal.**

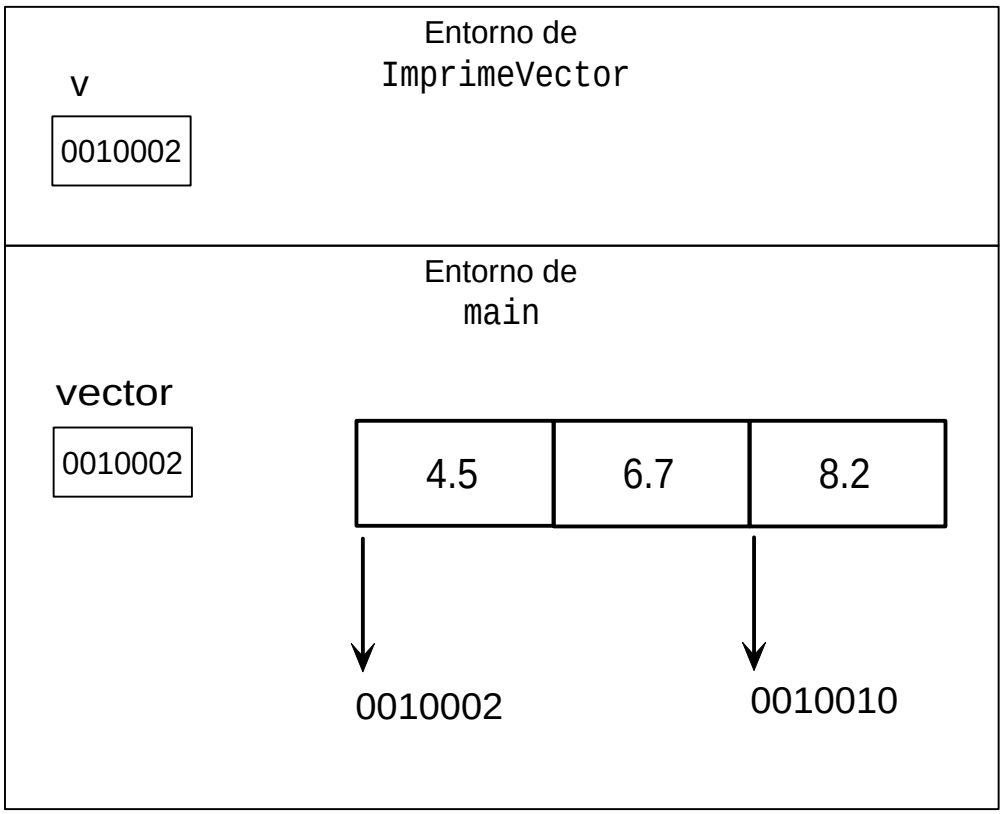
  ```
  ... (tipoBase vector[], ...)
  ```

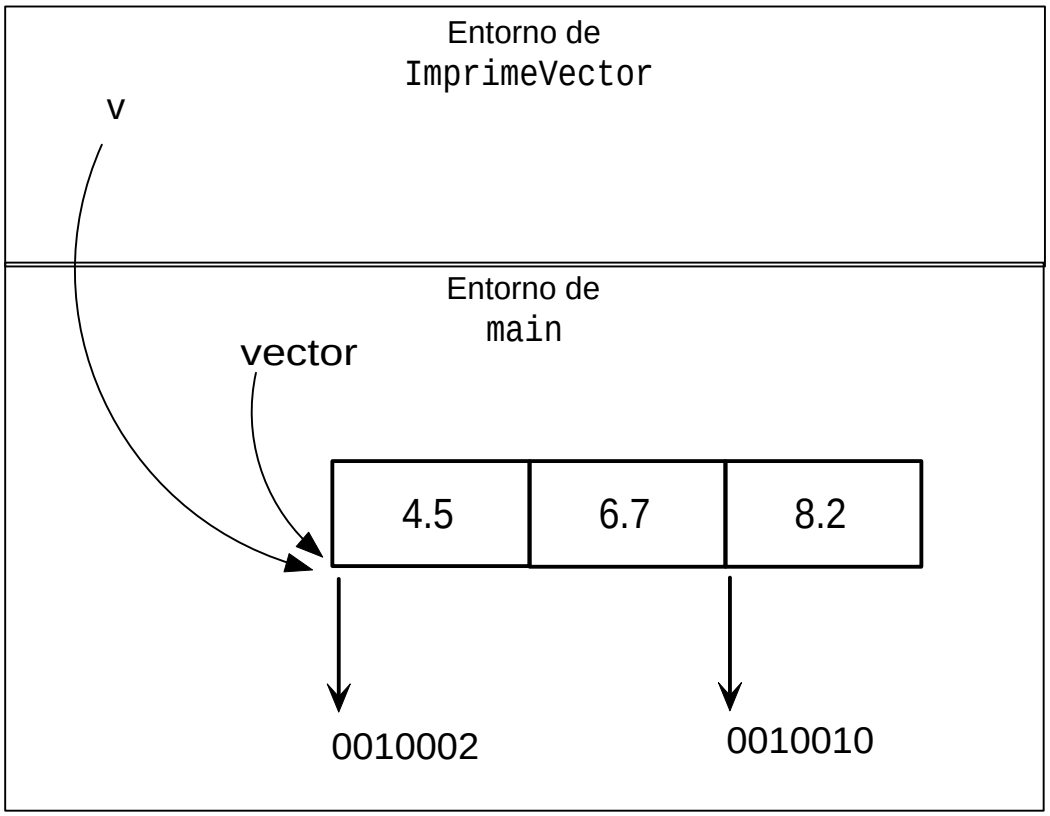
  En la llamada, se pasa como parámetro actual el identificador de un vector con el mismo tipo base que el correspondiente parámetro formal (sin poner los corchetes `[]`)

Observad que una dirección de memoria ocupa 4 bytes, por lo que esta forma de *pasar un vector* apenas requiere memoria adicional.

En C, pasar un vector como parámetro significa pasar la dirección de memoria de la primera componente del vector.
A partir de ella, se accede al resto de componentes.

```
#include <stdio.h>

void ImprimeVector(float v[]){  /* <- Con corchetes */
   int i;

   for (i=0; i<3; i++)
      printf("%f\n", v[i]);
}
int main(){
   float vector[3]={4.5, 6.7, 8.2};

   ImprimeVector(vector);        /* <- Sin corchetes */
}
```



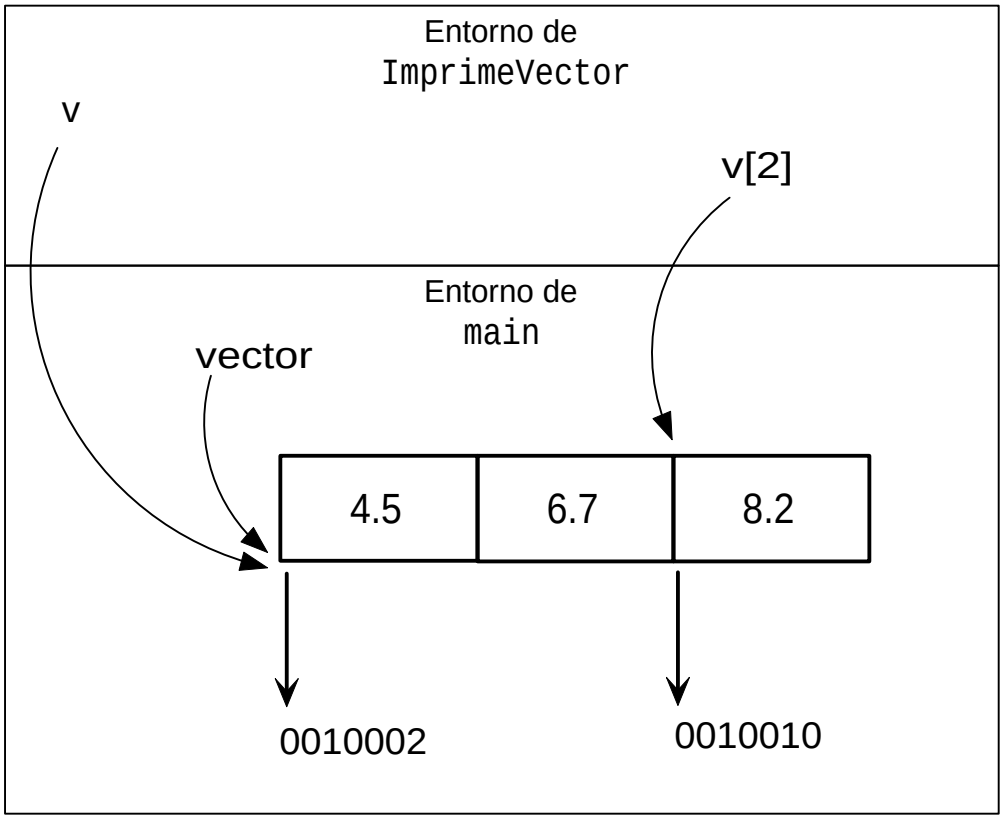

Entorno de
ImprimeVector
v
Entorno de
main
vector
4.5
6.7
8.2
0010002
0010010

En general, en una función cualquiera

```
<tipo> funcion (TipoBase v[], ....)
```

indicamos al compilador que la función va a recibir la primera dirección de unas posiciones contiguas en memoria (las del vector parámetro actual): cada una es de tipo de dato `TipoBase`**. De esta forma, cuando dentro del módulo accedamos a** `v[2]`**, por ejemplo, el compilador dará dos saltos tan grandes como diga el** `TipoBase` **a partir de la primera posición del parámetro actual.**

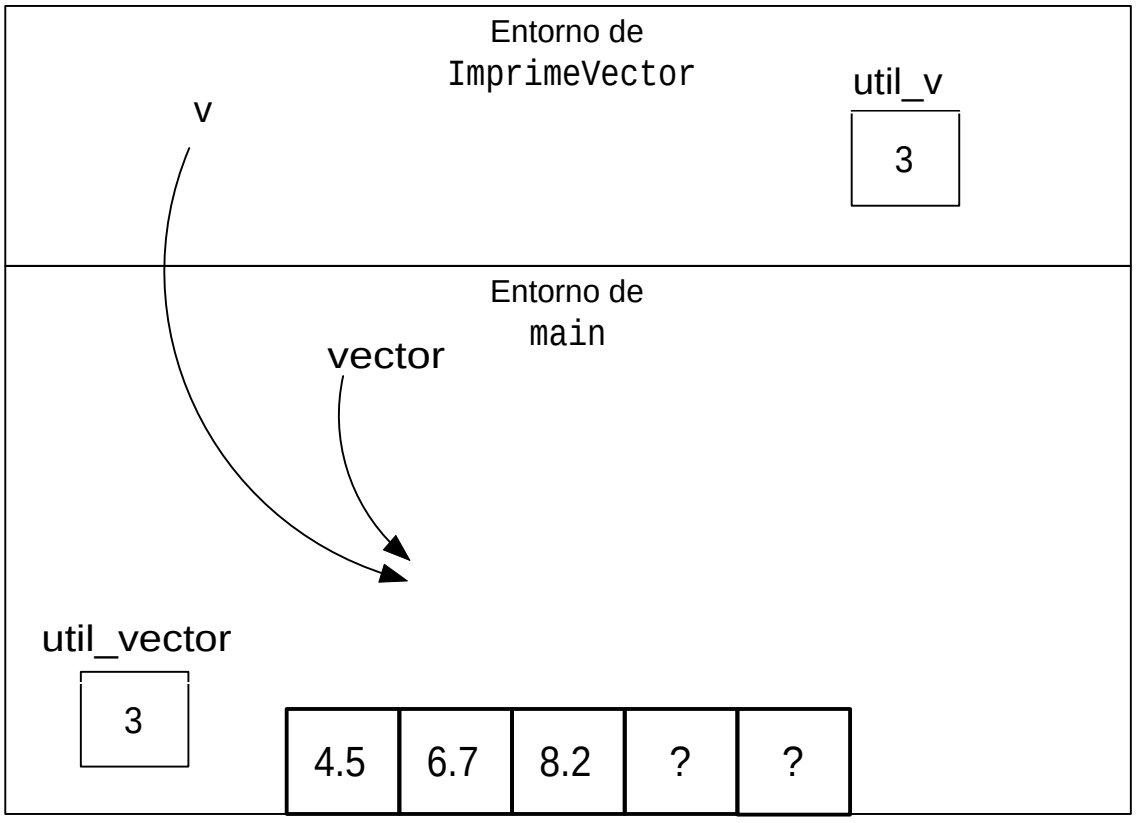
```
void ImprimeVector(float v[]){
   int i;

   for (i=0; i<3; i++)
      printf("%f\n", v[i]);
}
```


Entorno de
ImprimeVector
v
v[2]
Entorno de
main
vector
4.5
6.7
8.2
0010002
0010010

La función `ImprimeVector` sólo sirve para un vector de 3 componentes. En general, para saber hasta dónde debemos llegar:

> ***Usualmente, además de pasar el vector a una función, también le pasaremos el número de componentes utilizadas (pero no la dimensión)***

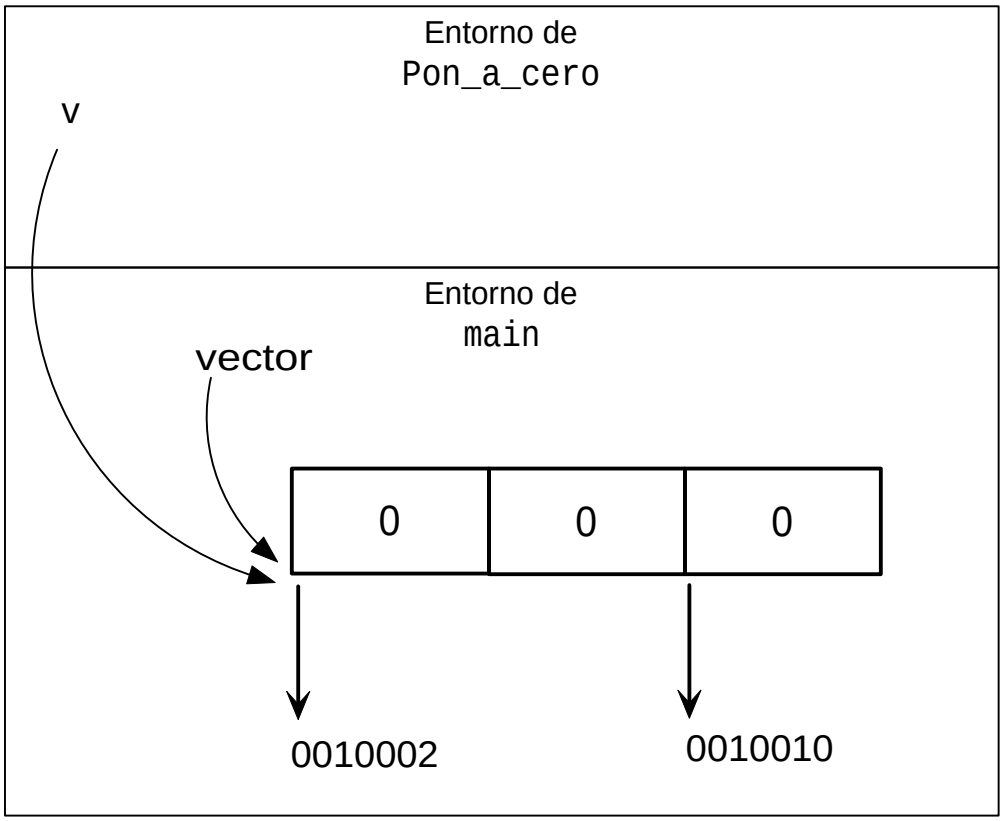
```
#include <stdio.h>
#define DIM_VECTOR 5

void ImprimeVector (float v[], int util_v){
   int i;

   for (i=0; i<util_v; i++)
      printf("%f\n", v[i]);
}
int main(){
   float vector[DIM_VECTOR] = {4.5, 6.7, 8.2};
   int util_vector = 3;

   ImprimeVector(vector, util_vector);
}
```



Ejercicio. **Calculad la mayor componente de un vector de `float`.**

Cuando una función recibe un vector (es decir, la dirección de memoria de la primera componente) tiene acceso a todas sus componentes, incluso para modificarlas.

```
#include <stdio.h>

void Pon_a_cero(float v[]){
   int i;

   for (i=0; i<3; i++)
      v[i] = 0;
}
int main(){
   float vector[3]={4.5, 6.7, 8.2};

   Pon_a_cero(vector);
}
```


Entorno de
Pon_a_cero
v
Entorno de
main
vector
0
0
0
0010002
0010010

Al tener acceso desde la función a las componentes del vector pasado como parámetro actual, se pueden producir efectos colaterales indeseados.

```
#include <stdio.h>

#define DIM_VECTOR 5

void Imprime_doble_de_vector (float v[], int util_v){
  int i;

  for (i=0; i<util_v; i++)
      v[i]=2*v[i];

  for (i=0; i<util_v; i++)
      printf("%f\n", v[i]);
}

int main(){
   float vector[DIM_VECTOR]={4,2,7};
   int util_vector=3;

   Imprime_doble_de_vector(vector, util_vector);
                                    /* -> 8, 4, 14 */

   for (i=0; i<util_v; i++)
      printf("%f\n", vector[i]);    /* -> 8, 4, 14 !!!! */
}
```

Si no se indica lo contrario, JAMÁS modificaremos las componentes de un vector de entrada

Si se quiere pasar un vector a una función y que ésta no pueda modificar sus componentes es necesario utilizar el calificador `const`.

```
#include <stdio.h>

void Imprime_doble_de_vector (const float v[], int util_v){
   int i;

   for (i=0; i<util_v; i++)
      v[i]=2*v[i];     /* Error en compilación  :-) */

   for (i=0; i<util_v; i++)
      printf("%f\n", v[i]);
}
```

Representación en memoria: exactamente la misma.

Es una buena norma de programación usar el calificador `const` ***en aquellos casos en los que la función no necesita modificar las componentes del vector***

Ejemplo. **Cread un `void` llamado `LecturaVectorFloat` para leer desde teclado los valores de un vector.**

Primero, leyendo el útil del vector dentro del `void`.

```
#define DIM_VECTOR 5

void LecturaVectorFloat (float v[], int *util_v){  /* <- * */
   int i;

   printf("Introduzca número de componentes: ");
   scanf("%d", util_v);

   printf("\nIntroduzca valores reales:\n");

   for (i=0 ; i<*util_v ; i++){
      printf("Posición %d: ", i);
      scanf("%f", &v[i]);
   }
}

int main(){
   float vector[DIM_VECTOR];
   int util_vector;

   LecturaVectorFloat(vector, &util_vector);
   ..............
}
```

Segundo, leyendo el útil antes de llamar al `void`

```
#define DIM_VECTOR 5

void LecturaVectorFloat (float v[], int util_v){
   int i;

   print("\nIntroduzca valores reales:\n");

   for (i=0 ; i<util_v ; i++){
      printf("Posición %d: ", i);
      scanf("%f", &v[i]);
   }
}

int main(){
   float vector[DIM_VECTOR];
   int util_vector;

   printf("Introduzca número de componentes: ");
   scanf("%d", &util_vector);

   LecturaVectorFloat(vector, util_vector);
   ............
}
```

¿Cual es preferible?

- ▷ **En la primera aproximación, una misma función realiza dos tareas: leer** `util_v` **y leer las componentes.** `:-(`
- ▷ **En la segunda, cada tarea se realiza en una función aparte. Mayor cohesión.** `:-)`

Al independizar las dos tareas, podemos hacer cosas del tipo:

```
#define DIM_VECTOR 5

int main(){
   float vector[DIM_VECTOR];
   int util_vector;

   do{
      printf("Introduzca número de componentes: ");
      scanf("%d", &util_vector);
   }while (util_vector>DIM_VECTOR || util_vector<0);

   LecturaVectorFloat(vector, util_vector);
   ..............
}
```

Con la primera solución, también podríamos hacer:

```
void LecturaVectorFloat (float v[], int *util_v){
   int i;
   do{
      printf("Introduzca número de componentes: ");
      scanf("%d", util_v);
   }while ((*util_v)>DIM_VECTOR || (*util_v)<0);

   printf("\nIntroduzca valores reales:\n");

   for (i=0 ; i<*util_v ; i++){
      printf("Posición %d: ", i);
      scanf("%f", &v[i]);
   }
}
```

Lo más importante es que seguramente habrá situaciones en las que el valor de `util` venga determinado de antemano, y no quiera leerlo desde teclado, por lo que es mejor hacerlo en sitios (funciones) distintos.

El parámetro actual y el formal han de ser del mismo tipo de dato: Con vectores, esto significa que han de tener el mismo tipo de dato base (aunque tengan distintas dimensiones)

```
#define DIM_VECTOR_GRANDE 10
#define DIM_VECTOR_PEQUE  5

void ImprimeVector (const float v[], int util_v){
   int i;
   for (i=0; i<util_v; i++)
      printf("%f\n", v[i]);
}

int main(){
   float vector_grande[DIM_VECTOR_GRANDE];
   float vector_peque[DIM_VECTOR_PEQUE];

   <Asignación de valores a los vectores>

   /* Todas estas llamadas son correctas: */

   ImprimeVector (vector_grande , 4);
   ImprimeVector (vector_grande , 5);
   ImprimeVector (vector_peque  , 4);
   ImprimeVector (vector_peque  , 5);
   ImprimeVector (vector_peque  , 9);
                                /* Imprime basura a partir
                                   de la sexta posición o
                                   error en ejecución. */
}
```

Sin embargo no podemos pasar como parámetros actuales, vectores de tipo de dato base *compatible* con el formal.

```
#define DIM_VECTOR_INT 10
#define DIM_VECTOR_FLOAT 10
void ImprimeVectorFloat (const float v[], int util_v){
   int i;
   for (i=0; i<util_v; i++)
      printf("%f\n", v[i]);
}
int main(){
 int    vector_int[DIM_VECTOR_INT];
 float vector_float[DIM_VECTOR_FLOAT];

 <Asignacion de valores a los vectores>

 ImprimeVectorFloat(vector_float, 3); /* Correcto. */
 ImprimeVectorFloat(vector_int, 3);
}                          /* warning en compilación */
```

***Solución*. Definiremos una función por cada tipo de dato**

```
   void ImprimeVectorInt (const int v[], int util_v){
      int i;
      for (i=0; i<util_v; i++)
         printf("%d\n", v[i]);
   }
   void ImprimeVectorFloat (const float v[], int util_v){
      int i;
      for (i=0; i<util_v; i++)
         printf("%f\n", v[i]);
   }
```

4.1.5. ALGORITMOS DE BÚSQUEDA

Al proceso de encontrar un elemento específico en un vector se denomina búsqueda.

- ▷ ***Búsqueda secuencial*. Técnica más sencilla.**
- ▷ ***Búsqueda binaria*. Técnica más eficiente, aunque requiere que el vector esté ordenado.**

4.1.5.1. BÚSQUEDA SECUENCIAL

- ▷ **El vector no se supone ordenado.**
- ▷ **Como no se modifica, se pasa como** `const <TipoBase> v[]`
- ▷ **Si no se encuentra, podríamos pasar una variable** `bool` **por referencia y ponerla a** `false`**. Pero no es necesario.**

  Devolvemos la posición dónde se encuentra o un valor imposible de posición si no se encuentra (por ejemplo, -1)

```
#include <stdio.h>

#define DIM_VECTOR 10

void LecturaVectorFloat (float v[], int util_v);
int Busca(const float v[], int util_v, float buscado);

int main(){
   float vector[DIM_VECTOR],encontrar;
   int util_vector = 5;
   int posicion;

   LecturaVectorFloat (vector, util_vector);

   scanf("%f", &encontrar);
   posicion = Busca(vector, util_vector, encontrar);

   if (posicion == -1)
      printf("\nEl valor %f no se encuentra en el vector",
             encontrar);
   else
      printf("\nEl valor %f se encuentra en la posición %d",
              encontrar, posicion);
}
```

Algoritmo:

```
Ir recorriendo las componentes del vector
  a) Mientras no se encuentre el elemento buscado, Y
  b) Mientras no lleguemos al final del mismo
Devolver la posición donde se encontró
o -1 en caso contrario
```

```
int Busca(const float v[], int util_v, float buscado){
     int i;
     int encontrado;

     i = 0;
     encontrado = 0;

     while ((i<util_v) && !encontrado)
        if (v[i] == buscado)
           encontrado = 1;
        else
           i++;

     if (encontrado)
        return i;
     else
        return -1;
}
```

▶ *Versión con un bucle* `for`

```
int Busca(const float v[], int util_v, float buscado){
   int i;
   int encontrado = 0;

   for (i=0; i<util_v && !encontrado; i++)
      if (v[i] == buscado)
         encontrado = 1;

   if (encontrado)
      return i;      /* :-( Error logico */
   else
      return -1;
}
```

Mejor sin error logico:

```
   if (encontrado)
      return i-1;
   else
      return -1;
```

O mejor:

```
int Busca(const float v[], int util_v, float buscado){
   int i, pos_encontrado;
   int encontrado = 0;

   for (i=0; i<util_v && !encontrado; i++)
      if (v[i] == buscado){
         encontrado = 1;
         pos_encontrado = i;
      }

   if (encontrado)
      return pos_encontrado;
   else
      return -1;
```

¿Valdría lo siguiente?

```
int Busca(const float v[], int util_v, float buscado){
   int i;
   int encontrado = 0;

   for (i=0; i<util_v && !encontrado; i++)
      if (v[i] == buscado)
         encontrado = 1;

   if (i == util_v)
      return -1;
   else
      return i;
}
```

Precondiciones de uso:

- ▷ `util_v` **debe estar en el rango correcto.**

Batería de pruebas:

- ▷ **Que el valor a buscar esté.**
- ▷ **Que el valor a buscar no esté.**
- ▷ **Que el valor a buscar esté varias veces (devuelve la primera ocurrencia).**
- ▷ **Que el valor a buscar esté en la primera o en la última posición.**
- ▷ **Que el vector esté vacío o tenga una única componente.**

A veces, podemos estar interesados en buscar entre una componente izquierda y otra derecha, ambas inclusive.

```
int BuscaEntre (const float v[], int izda, int dcha,
                float buscado){
   int i;
   int encontrado = 0;

   for (i=izda ; i<=dcha && !encontrado ; i++)
      encontrado = (v[i]==buscado);

   if (encontrado)
      return i-1;
   else
      return -1;
}
```

4.1.5.2. BÚSQUEDA BINARIA

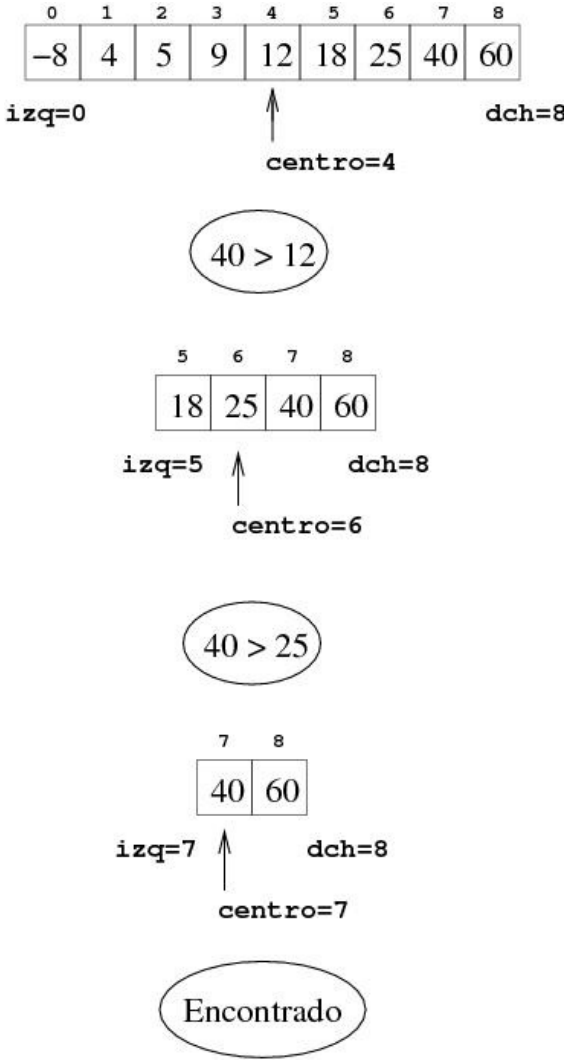
***Precondiciones de uso*: Se aplica sobre un vector ordenado.**

Los parámetros a pasar son los mismos.

```
int BuscaBinaria(const float v[], int util_v,
                 float buscado)
```

***Nota*. El nombre del identificador de la función refleja que no es una búsqueda cualquiera, sino que necesita una precondición.**

Ejemplo. Buscar el 40



Algoritmo:

```
El elemento a buscar se compara con el elemento que
   ocupa la mitad del vector.
Si coinciden, se habrá encontrado el elemento.
En otro caso, se determina la mitad del vector
   en la que puede encontrarse.
Se repite el proceso con la mitad correspondiente.
```

```
int BuscaBinaria(const float v[], int util_v,
                 float buscado) {
   int izq, dch, centro;

   izq = 0;
   dch = util_v-1;
   centro = (izq+dch)/2;

   while ((izq<=dch) && (v[centro]!=buscado)) {
      if (buscado < v[centro])
         dch = centro-1;
      else
         izq = centro+1;
      centro = (izq+dch)/2;
   }

   if (izq>dch)
      return -1;
   else
      return centro;
}
```

Ejercicio. **Eliminar una componente de un vector. Para ello, desplazaremos una posición hacia la izquierda, todas las componentes que haya a la derecha.**

Ejercicio. **Comprobar si todos los elementos de un vector están en un segundo vector, respetando el orden en el que aparecen. Por ejemplo, el vector**
(-3,8)
se encuentra dentro del vector
(2,-3,1,8,-5)

4.1.6. Construcción de vectores en una función

Supongamos que queremos que un módulo *construya* un vector. Éste no puede ser local ya que al terminar el módulo su zona de memoria *desaparece*.

Debemos declarar dicho vector en el `main` (en general, en la función que realiza la llamada) y pasarlo como parámetro, para *rellenar* las componentes desde dentro de la función. Esto ya lo hicimos en la página 37

Ejemplo. **Crear un vector con las componentes pares de otro.**

```
   vector = (2,1,7,3,4,8,?,?)  util_vector = 6
   salida = (2,4,8,?,?,?,?,?)  util_salida = 3
```

```
#define DIM 100
void LecturaVectorInt (int v[], int util_v);
void ImprimeVectorInt (const int v[], int util_v);
void SoloPares(const int v[], int util_v, int vPares[],
               int *util_vPares);

int main(){
   int vector[DIM], salida[DIM];
   int util_vector = 7, util_salida;

   LecturaVectorInt (vector, util_vector);
   SoloPares (vector, util_vector, salida, &util_salida);
   ImprimeVectorInt (salida, util_salida);
}
```

```
<Definición de LecturaVectorInt e ImprimeVectorInt>

void SoloPares(const int v[], int util_v, int vPares[],
               int *util_vPares){
   int i;
   (*util_vPares)=0;

   for (i=0; i<util_v; i++)
      if ((v[i]%2) == 0){
         vPares[*util_vPares]=v[i];
         (*util_vPares)++;
      }
}
```

Ejemplo. **Quitar los elementos repetidos de un vector, guardando el resultado en otro vector.**

```
   vector = (2,1,2,1,4,1,?,?)  util_vector = 6
   salida = (2,1,4,?,?,?,?,?)  util_salida = 3
```

```
#include <stdio.h>
#define DIM 100

void QuitaRepetidos (const float original[],
                     int util_original,
                     float destino[],
                     int *util_destino);

void LecturaVectorFloat (float v[], int util_v);
void ImprimeVectorFloat (const float v[], int util_v);

int main(){
   float vector[DIM], salida[DIM];
   int util_vector = 7, util_salida;

   LecturaVectorFloat(vector, util_vector);
   QuitaRepetidos(vector, util_vector, salida, &util_salida);
   ImprimeVectorFloat(salida, util_salida);
}
```

Algoritmo:

```
destino[0]=original[0]
Recorrer todas las componentes i de original
   Si original[i] se encuentra en original sin original[i]
      i++
   Si no
      Añadir original[i] a destino
      Incrementar util_destino
      i++
```

Como se repite `i++`, podemos sacarlo del condicional. Por lo tanto, el primero se queda vacío, y re-escribimos el algoritmo como sigue:

```
destino[0]=original[0]
Recorrer todas las componentes i de original
   Si original[i] NO se encuentra en original sin original[i]
      Añadir original[i] a destino
      Incrementar util_destino
   i++
```

Observad que, en vez de buscar dentro de `original`, podemos buscar dentro de `destino`. Al tener menos elementos, la búsqueda será más eficiente:

```
destino[0]=original[0]
Recorrer todas las componentes i de original
   Si original[i] NO se encuentra en destino /* <- */
      Añadir original[i] a destino
      Incrementar util_destino
   i++
```

```
void QuitaRepetidos (const float original[],
                     int util_original,
                     float destino[],
                     int *util_destino){
   int encontrado, i, j;

   (*util_destino) = 0;

   for (i=0 ; i<util_original ; i++){
      encontrado = 0;

      for (j=0 ; (j<(*util_destino)) && !encontrado ; j++){
         if (original[i] == destino[j])
            encontrado = 1;
      }

      if (!encontrado){
         destino[(*util_destino)] = original[i];
         (*util_destino)++;
      }
   }
}
```

Mejor si utilizamos la función `Busca` que habíamos construido anteriormente:

```
void QuitaRepetidos (const float original[],
                     int util_original,
                     float destino[],
                     int *util_destino){
   int pos_encontrado, i;
   (*util_destino) = 0;

   for (i=0 ; i<util_original ; i++){
      pos_encontrado =
         Busca (destino, (*util_destino), original[i]);

      if (-1==pos_encontrado){
         destino[(*util_destino)] = original[i];
         (*util_destino)++;
      }
   }
}
```

4.1.7. TRABAJANDO CON VECTORES LOCALES

Ejemplo. **Calculad la moda de un vector de enteros, es decir, el elemento que más se repite. La moda de** `(4,2,1,2,3,2,5,1)` **sería el** `2`**.**

Vamos a calcular el conteo de la moda (su frecuencia) en la misma función:

```
#include <stdio.h>

#define DIM 20

void ModaVector(const float v[], int util_v,
                float *moda, int *conteo_moda);

int main(){
   float notas[DIM] = {1,2,4,2,3,2,5,5,5,2};
   float moda_notas;
   int conteo_moda_notas;

   ModaVector(notas, 10, &moda_notas, &conteo_moda_notas);
   printf("%f - %d", moda_notas, conteo_moda_notas);
}
```

Para hacer los cómputos, vamos a usar un vector de procesados (luego se verá la razón).
¿Cómo declaramos el vector local `procesados`**?**

a)
```
float Moda(const float v[], int util_v){
    float procesados[util_v];
    ..............
```

Imposible: no podemos dimensionar con una variable.

b)
```
float Moda(const float v[], int util_v){
    const int DIM = util_v;
    float procesados[DIM];
    ..............
```

Imposible: En C++ no podemos inicializar una constante con una variable.

c)
```
float Moda(const float v[], int util_v){
    float procesados[100];
    ..............
```

¿y por qué no 200?

d)
```
#define DIM 100

float Moda(const float v[], int util_v){
   float procesados[DIM];
   ..............
}
int main(){
   float vector[DIM];
   ..............
}
```

Es la única solución :-(

Algoritmo:

Usamos un vector local `procesados` **para almacenar las componentes que ya se han recorrido**

```
Recorrer todos los elementos v[pos] del vector
   Si v[pos] no está en procesados =>
      Añadimos v[pos] a procesados.
      Contamos las apariciones de v[pos] en v
         (desde pos hasta el final)
      Actualizamos, en su caso, la moda
```

***Nota*. Si** `v[pos]` **está en procesados, ya lo hemos procesado, por lo que no hay que hacer nada**

```
void ModaVector(const float v[], int util_v,
                float *moda, int *conteo_moda){
   float procesados[DIM];
   int  conteo_parcial, util_procesados;
   int pos, dcha;

   (*conteo_moda) = 0;
   util_procesados = 0;

   for (pos=0 ; pos<util_v ; pos++){
      if (-1 == Busca (procesados, util_procesados, v[pos])){
         procesados[util_procesados] = v[pos];
         util_procesados++;
         conteo_parcial = 0;

         for (dcha=pos; dcha<util_v; dcha++)
            if (v[pos] == v[dcha])
               conteo_parcial++;

         if (conteo_parcial > (*conteo_moda)){
            (*conteo_moda) = conteo_parcial;
            (*moda) = v[pos];
         }
      }
   }
}
```

Batería de pruebas:

- ▷ **Que el vector esté vacío**
- ▷ **Que el vector tenga todas las componentes iguales**
- ▷ **Que el vector tenga todas las componentes distintas**
- ▷ **Que el vector tenga todas las componentes iguales excepto 1**
- ▷ **Que el vector tenga todas las componentes distintas excepto 1 con 2 repeticiones**
- ▷ **Que la moda sea la primera o la última componente**

Nota. En este ejemplo concreto, no era necesario el vector local

```
Recorrer todos los elementos v[pos] del vector
   Si v[pos] es distinto a todas las componentes
   que hay a su izquierda  =>
      Contamos las apariciones de v[pos] en v
         (desde pos hasta el final)
      Actualizar, en su caso, la moda
```

Ya sabemos declarar un vector local dentro de una función. ¿Podemos hacer return de dicho vector? NO

```
Error compilación --> int[] MiFuncion(int parametro){
                          float v[5];
                          .......
                          return v;  /* <- No es posible */
                      }
```

Para construir un vector dentro de `MiFuncion`, hay que reservar las componentes en el `main` (en general en la función que llame a `MiFuncion`) y pasarlo a `MiFuncion` para que las modifique. Esto ya lo hemos hecho en la página 54.

Una función no puede devolver un vector

4.1.8. ALGORITMOS DE ORDENACIÓN

La ordenación es un procedimiento mediante el cual se disponen los elementos de un vector en un orden especificado, tal como orden alfabético u orden numérico.

Aproximaciones:

- ▷ **Construir un segundo vector con las componentes del primero, pero ordenadas**
- ▷ **Modificar el vector original, cambiando de sitio las componentes.**
  Ésta aproximación es la que seguiremos.

Existen dos tipos de ordenación:

- ▷ ***Ordenación interna*: Todos los datos están en memoria principal durante el proceso de ordenación.**
  - **– Inserción.**
  - **– Selección.**
  - **– Intercambio.**
- ▷ ***Ordenación externa*: Parte de los datos a ordenar están en memoria externa mientras que otra parte está en memoria principal siendo ordenada.**

```
#include <stdio.h>
#define DIM 10

void ImprimeVectorFloat(const float v[], int util_v);

void LecturaVectorFloat (float v[], int util_v);

void Ordenar (float v[], int util_v);  /* util_v no cambia */

int main(){

   float vector[DIM];
   int util_vector = 4;

   LecturaVectorFloat(vector, util_vector);
   Ordenar (vector, util_vector);
   ImprimeVectorFloat(vector, util_vector);
}
```

4.1.8.1. ORDENACIÓN POR SELECCIÓN

Ejemplo. **Calcular el mínimo elemento de un vector.**

`v = (2,4,7,1,8,?,?,?)`

Mejor que devolver el elemento `1`, devolvemos su posición `3`

***Algoritmo*:**

```
Inicializar minimo a la primera componente
Recorrer el resto de componentes v[i] (i>0)
   Actualizar, en su caso, minimo (y la posición)
```

```
int PosMinimo (const float v[], int util_v){
   int posicion_minimo;
   float minimo;
   int i;

   minimo = v[0];
   posicion_minimo = 0;

   for (i=1; i<util_v ; i++)
      if (v[i] < minimo){
         minimo = v[i];
         posicion_minimo = i;
      }
   return posicion_minimo;
}
```

Ejemplo. **Calcular el mínimo elemento de un vector, pero pasándole al módulo las componentes** `izda` **y** `dcha` **donde se quiere trabajar.**

`v = (6,8,7,2,4,9,3,1,?,?)  izda=2 , dcha=5`

`PosMinimo(v,izda,dcha)` **devuelve la posición** `3` **(índice de** `v`**)**

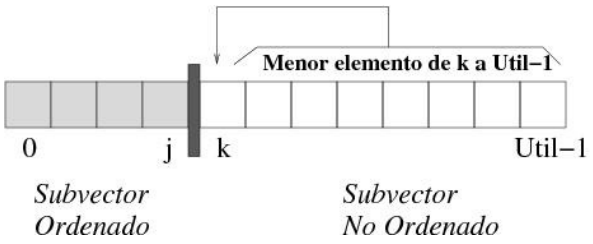
Precondición: `0 <= izda <= dcha <= util_v-1`

```
int PosMinimo (const float v[], int izda, int dcha){
   int posicion_minimo;
   float minimo;
   int i;

   minimo = v[izda];
   posicion_minimo = izda;

   for (i=izda+1 ; i <= dcha ; i++)
      if (v[i] < minimo){
         minimo = v[i];
         posicion_minimo = i;
      }

   return posicion_minimo;
}
```

***Idea común a algunos algoritmos de ordenación*:**

- ▷ **El vector se dividirá en dos sub-vectores. El de la izquierda, contendrá componentes ordenadas. Las del sub-vector derecha no están ordenadas.**
- ▷ **Se irán cogiendo componentes del sub-vector derecha y se colocarán adecuadamente en el sub-vector izquierda.**

***Ordenación por selección*: En cada iteración, se selecciona la componente más pequeña del sub-vector derecha y se coloca al final del sub-vector izquierdo.**

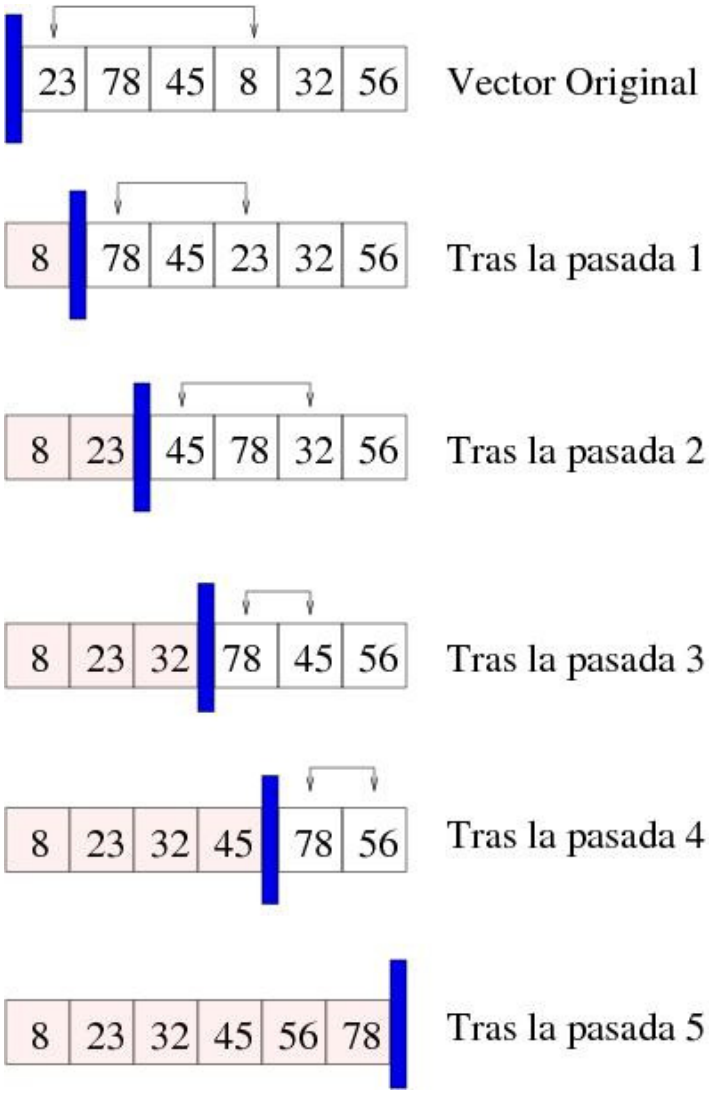


***Algoritmo*:**

```
Recorrer todos los elementos v[izda] de v
   Hallar la posición pos_min del menor elemento
      del subvector delimitado por las componentes
      [izda , util_v-1] ¡ambas inclusive!
   Intercambiar v[izda] con v[pos_min]
```

En el bucle principal no es necesario recorrer todos los elementos de `v`**, ya que si sólo queda 1 componente, el vector está ordenado. Por lo tanto, el bucle principal será de la forma**

```
izda<util_v-1
```


23 78 45 8 32 56
Vector Original
8 78 45 23 32 56
Tras la pasada 1
8 23 45 78 32 56
Tras la pasada 2
8 23 32 78 45 56
Tras la pasada 3
8 23 32 45 78 56
Tras la pasada 4
8 23 32 45 56 78
Tras la pasada 5

```
#include <stdio.h>
#define DIM 10

void ImprimeVectorFloat(const float v[], int util_v);
void LecturaVectorFloat(float v[], int util_v);
int PosMinimo(const float v[], int izda, int dcha);
void IntercambiaFloat(float *a, float *b);
void OrdSeleccion(float v[], int util_v);

int main(){
   float vector[DIM];
   int util_vector = 4;

   LecturaVectorFloat(vector, util_vector);
   OrdSeleccion(vector, util_vector);
   ImprimeVectorFloat(vector, util_vector);
}

void ImprimeVectorFloat(const float v[], int util_v){
   ......................
}

void LecturaVectorFloat(float v[], int util_v){
   ......................
}

int PosMinimo(const float v[], int izda, int dcha){
   ......................
}
```

```
void IntercambiaFloat(float *a, float *b){
   float aux;

   aux = (*a);
   (*a)   = (*b);
   (*b)   = aux;
}

void OrdSeleccion(float v[], int util_v){
   int pos_min,izda;

   for (izda=0 ; izda<util_v-1 ; izda++){
      pos_min = PosMinimo(v, izda, util_v-1);
      IntercambiaFloat(&v[izda], &v[pos_min]);
   }
}
```

Versión sin llamadas a funciones:

```
void OrdSeleccion(float v[], int util_v){
   int izda, dcha, pos_min;
   float aux;

   for (izda=0; izda<util_v-1; izda++){
      pos_min=izda;

      for (dcha=izda+1; dcha<util_v; dcha++)
         if (v[dcha] < v[pos_min])
            pos_min=dcha;

      aux = v[izda];
      v[izda] = v[pos_min];
      v[pos_min] = aux;
   }
}
```

Compromiso reutilización versus eficiencia. Usualmente prevalecerá la reutilización. Pero si tenemos que construir un módulo, verificando alguna de estas restricciones:

- ▷ **que realice muchas operaciones**
- ▷ **que sea utilizable en otros programas**

merece la pena construirlo eficientemente.

Batería de pruebas:

- ▷ **Que el vector esté vacío**
- ▷ **Que tenga un número de componentes par/impar**
- ▷ **Que el vector ya estuviese ordenado**
- ▷ **Que el vector tenga todas las componentes iguales**
- ▷ **Que tenga dos componentes iguales al principio**
- ▷ **Que tenga dos componentes iguales al final**
- ▷ **Que tenga dos componentes iguales en medio**

Applet de demostración del funcionamiento:

`http://www.sorting-algorithms.com/`

4.1.8.2. ORDENACIÓN POR INSERCIÓN

Ejemplo. **Insertar un elemento en una posición concreta de un vector.**

```
v = (2,7,4,8,1,?,?,?)
pos_insercion = 3
valor = 5
v = (2,7,4,5,8,1,?,?)
```

***Prototipo*:**

```
void Inserta(float v[], int *util_v,
             int pos_insercion, float valor);
```

Importante:

- ▷ **Hay que incrementar** `util_v` **en 1. Por eso se pasa por referencia.**
- ▷ **Precondición:** `util_v < DIM_V`

```
#define DIM_VECTOR 8
void Inserta(float v[], int *util_v,
             int pos_insercion, float valor);
int main(){
   float vector[DIM_VECTOR] = {2,7,4,8,1};
   int util_vector = 5;

   Inserta(vector, &util_vector, 3, 5);
}
```

Algoritmo: (primera versión):

```
Recorrer (i) las componentes posteriores a pos_insercion
hasta llegar al final de v
   v[i] = v[i-1];
   i++;
v[pos_insercion] = valor;
```

Algoritmo: (versión correcta):

```
Recorrer (i) las componentes desde el final del vector
hasta llegar a pos_insercion
   v[i] = v[i-1];
   i--;
v[pos_insercion] = valor;
```

```
#define DIM_VECTOR 8
void Inserta(float v[], int *util_v,
             int pos_insercion, float valor);
int main(){
   float vector[DIM_VECTOR] = {2,7,4,8,1};
   int util_vector = 5;

   Inserta(vector, &util_vector, 3, 5);
}

void Inserta(float v[], int *util_v,
             int pos_insercion, float valor){
   int i;

   for (i=(*util_v) ; i>pos_insercion ; i--)
      v[i] = v[i-1];

   v[pos_insercion] = valor;
   (*util_v)++;
}
```

Supongamos que queremos quitar la precondición

`util_v < DIM_V`

Posibles soluciones:

▷ **Usar el define** `DIM_VECTOR` **y comprobar su valor dentro del módulo** `Inserta`

```
#define DIM_VECTOR 100
void Inserta(float v[], int *util_v, int pos_insercion,
             float valor, int *error){

   (*error) = ((*util_v) >= DIM_VECTOR);

   if (!(*error)){
      for (int i=(*util_v) ; i>pos_insercion ; i--)
         v[i] = v[i-1];

      v[pos_insercion] = valor;
      (*util_v)++;
   }
}
int main(){
   float vect[DIM_VECTOR];
   int err;
   ............
   Inserta(vect, &util_vect, 3, 5, &err);
   if (err)
      printf("Número de componentes insuficiente");
   else
      ............
}
```

▷ Pasar la dimensión como parámetro a `Inserta`

```
#define DIM_VECTOR 100
void Inserta(float v[], int DIMEN, int *util_v,
             int pos_insercion, float valor, int *error){

   (*error) = ((*util_v) >= DIMEN);

   if (!(*error)){
      for (int i=(*util_v) ; i>pos_insercion ; i--)
         v[i] = v[i-1];

      v[pos_insercion] = valor;
      (*util_v)++;
   }
}
int main(){
   float vect[DIM_VECTOR];
   int err;
   ............
   Inserta(vect, DIM_VECTOR, &util_vect, 3, 5, &err);

   if (err)
      printf("Número de componentes insuficiente");
   else
      ............
}
```

Ventajas de las soluciones anteriores:

- ▷ **Conseguimos una función más *robusta*, sin peligro de que ejecute una sentencia potencialmente peligrosa.**

Inconvenientes:

- ▷ **En la primera alternativa no podemos *separar* la definición de la función del define. Están fuertemente acoplados.**
- ▷ **En la segunda alternativa, aumentamos el número de parámetros (al pasar también la dimensión).**

Siempre es difícil conseguir un equilibrio entre robustez y simplicidad en la llamada. En este ejemplo, optaríamos por la simplicidad, manteniendo la precondición:

```
/* Precondición: util_v < dimensión de v */

void Inserta(float v[], int *util_v,
             int pos_insercion, float valor)
```

Ejemplo. **Insertad un elemento de forma ordenada en un vector ya ordenado.**

```
v = (1,2,4,7,8,?,?,?)
valor = 5
v = (1,2,4,5,7,8,?,?)
```

Prototipo:

```
void InsertaOrd(float v[], int *util_v, float valor)
```

Importante. `util_v` **se pasa por referencia ya que debe aumentar en 1, después de la inserción.**

***Idea*:**

- ▷ **Hay que desplazar componentes como en el ejemplo anterior.**
- ▷ **La condición de parada cambia.**

***Algoritmo*:**

```
Recorrer las componentes desde el final del vector
hasta llegar a una componente que sea menor que el valor
o hasta llegar al principio
   v[i] = v[i-1];

v[i] = valor;
```

```
void InsertaOrd(float v[], int *util_v, float valor){
     int i;

     for (i=*util_v; i>0 && valor<v[i-1]; i--)
          v[i] = v[i-1];

     v[i] = valor;
     (*util_v)++;
}
```

***Batería de pruebas*:**

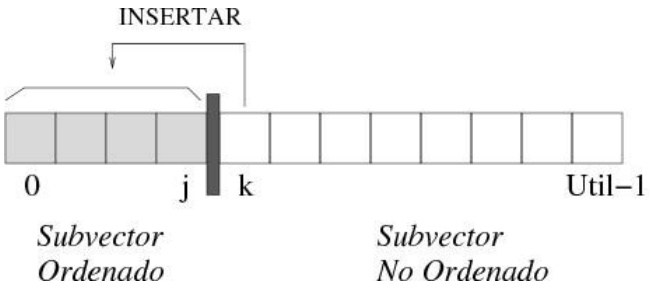
- ▷ **Que el vector esté vacío**
- ▷ **Que el elemento a insertar se sitúe el último**
- ▷ **Que el elemento a insertar se sitúe el primero**
- ▷ **Que el vector tenga todas las componentes iguales**
- ▷ **Que el elemento a insertar sea igual a alguna de las componentes**

Ejercicio. **Cread un módulo para leer 100 elementos desde teclado, y conforme los va leyendo, vaya creando un vector ordenado.**

Ordenación por inserción

***Idea*:**
El vector se divide en dos subvectores: el de la izquierda ordenado, y el de la derecha desordenado.

Cogemos el primer elemento del subvector desordenado y lo insertamos de forma ordenada en el subvector ordenado.

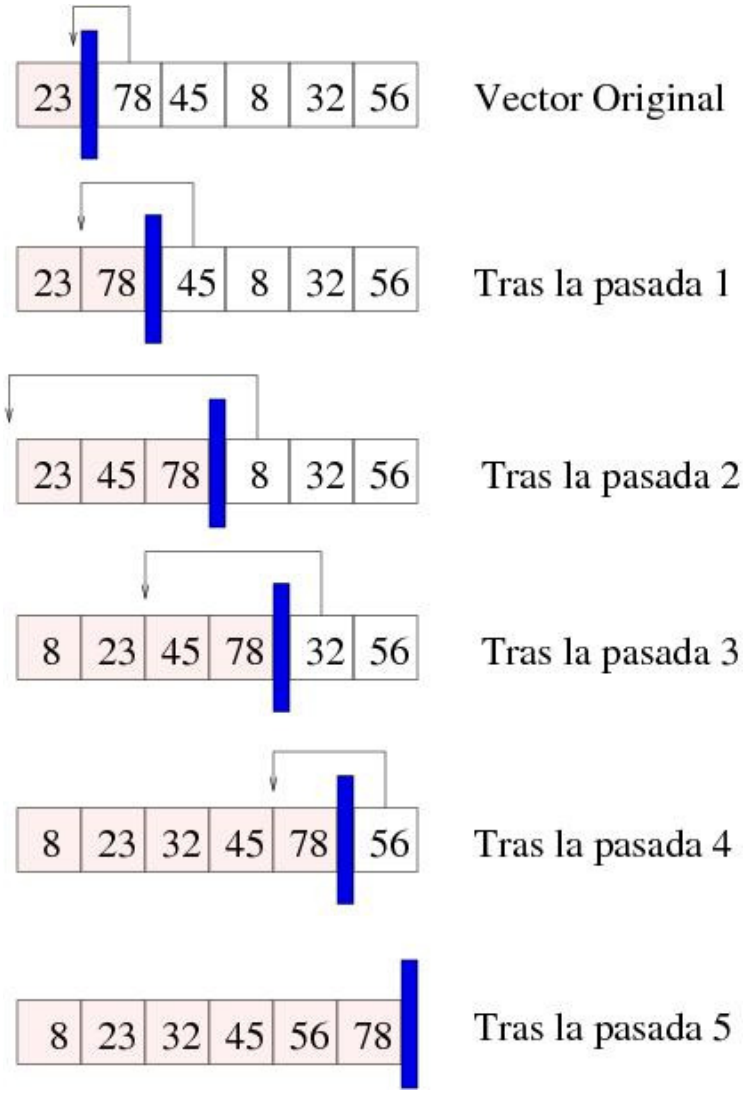


***Importante.* La componente de la posición k (primer elemento del subvector desordenado) será remplazada por la anterior (después de desplazar)**

Algoritmo:

```
Ir fijando el inicio del subvector izquierda
con un contador izda desde 1 hasta util_v
   Seleccionar el valor v[izda]
   Insertar dicho valor de forma ordenada
   en el subvector de v delimitado por [0,izda-1]
```

***Nota*. Empezamos desde 1, ya que la primera componente forma un subvector ya ordenado**


23 78 45 8 32 56
Vector Original
23 78 45 8 32 56
Tras la pasada 1
23 45 78 8 32 56
Tras la pasada 2
8 23 45 78 32 56
Tras la pasada 3
8 23 32 45 78 56
Tras la pasada 4
8 23 32 45 56 78
Tras la pasada 5

```
    void InsertaOrd(float v[], int *util_v, float valor);

void OrdInsercion (float v[], int util_v){
    int izda;
    float valor;

    for (izda=1; izda<util_v; izda++){
       valor = v[izda];
       InsertaOrd(v, &izda, valor);
    }
 }
```

Obviamente, no es necesaria la variable `valor`:

```
void OrdInsercion (float v[], int util_v){
    int izda;

    for (izda=1; izda<util_v; izda++)
       InsertaOrd(v, &izda, v[izda]);
 }
```

Sin embargo, el algoritmo no funciona correctamente!
Razón: `izda` **se pasa por referencia y en cada iteración se modifica en la llamada a** `InsertaOrd`**.**

Solución:

```
void OrdInsercion (float v[], int util_v){
   int izda = 1;

   while (izda < util_v)
      InsertaOrd(v, &izda, v[izda]);
}
```

Sin llamadas a funciones:

```
   void OrdInsercion (float v[], int util_v){
      int izda, i;
      float valor;

      for (izda=1; izda<util_v; izda++){
          valor = v[izda];

          for (i=izda; i>0 && valor<v[i-1]; i--)
              v[i] = v[i-1];

          v[i] = valor;
      }
   }
```

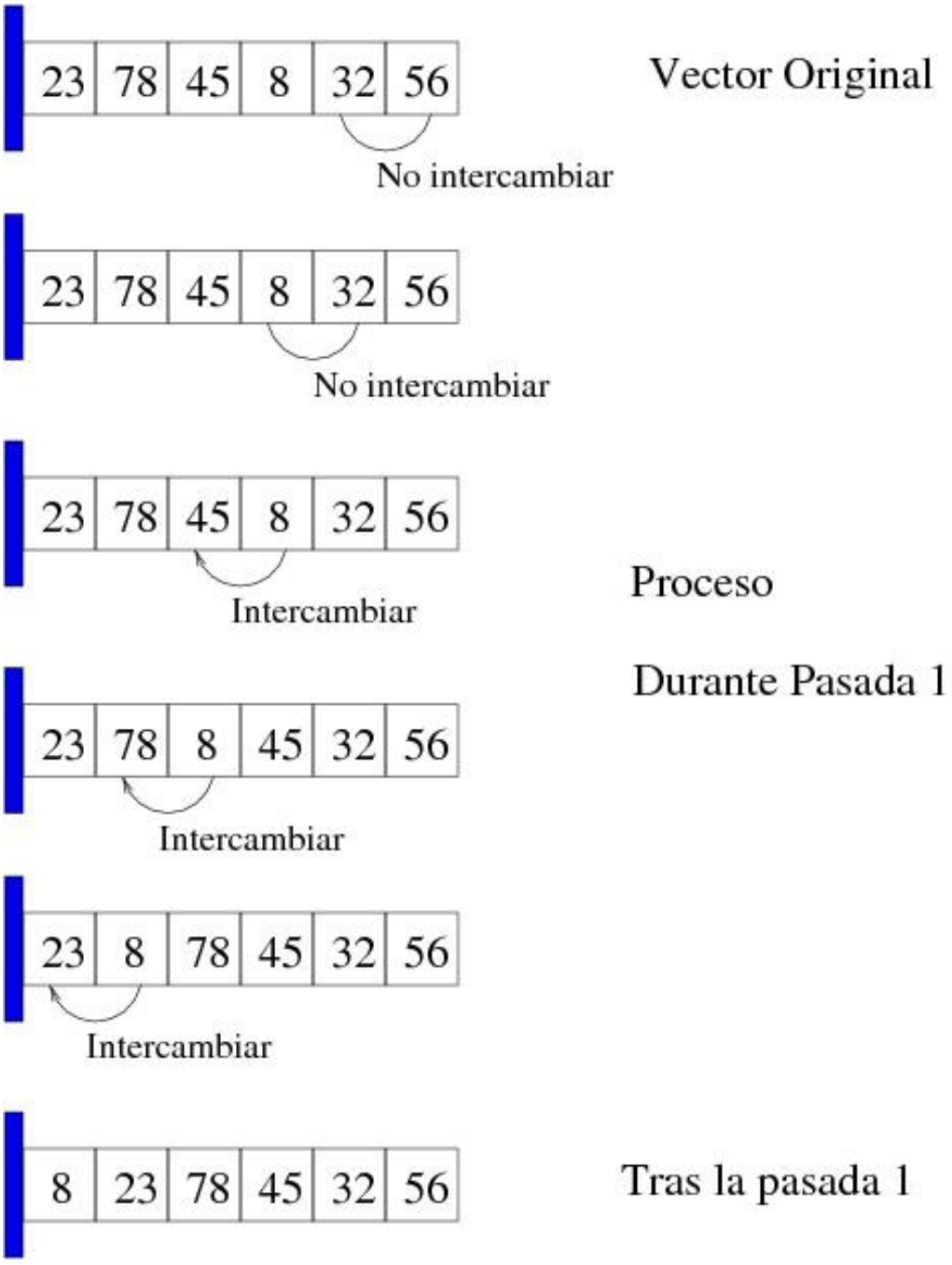
Ejercicio. Borrad un elemento de un vector.

4.1.8.3. Ordenación por Intercambio Directo (Método de la Burbuja)

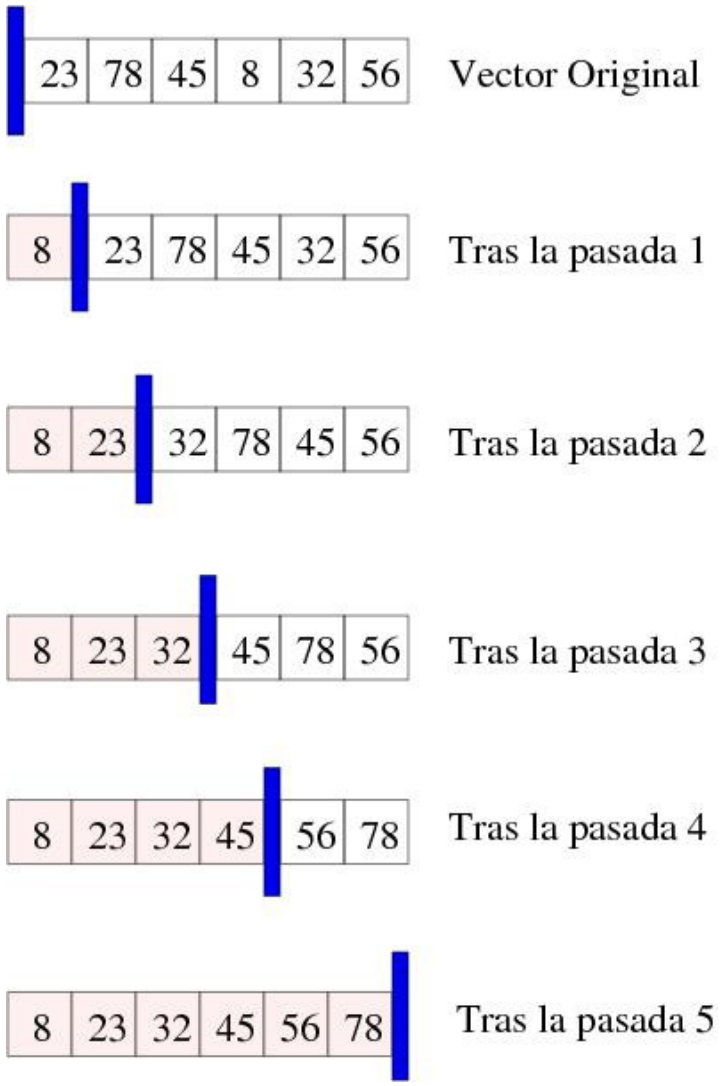
Al igual que antes, a la izquierda se va dejando un subvector ordenado.

Desde el final y hacia atrás, se van comparando elementos dos a dos y se deja a la izquierda el más pequeño (hay que intercambiarlos).

Ejemplo. (Primera Pasada)

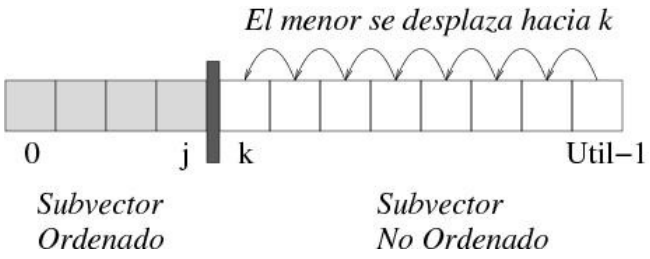


Ejemplo. (Resto de Pasadas)



Algoritmo:

```
Ir fijando el inicio del subvector izquierda
con un contador izda desde 0 hasta util_v
   Recorrer el vector de la derecha desde
   el final (util_v) hasta el principio (izda)
   con un contador i
      Si v[i] < v[i-1] intercambiarlos
```



```
void OrdBurbuja (float v[], int util_v){
   int izda, i;

   for (izda=0; izda<util_v; izda++)
     for (i=util_v-1 ; i>izda ; i--)
       if (v[i] < v[i-1])
          IntercambiaFloat(&v[i], &v[i-1]);
}
```

***Mejora*. Si en una pasada del bucle más interno no se produce ningún intercambio, el vector ya está ordenado. Lo comprobamos con una variable lógica.**

▶ *Segunda Aproximación*

```
void OrdBurbuja (float v[], int util_v){
   int izda, i;
   int cambio;

   cambio=1;

   for (izda=0; izda<util_v && cambio; izda++){
     cambio=0;

     for (i=util_v-1 ; i>izda ; i--)
       if (v[i] < v[i-1]){
          IntercambiaFloat(&v[i], &v[i-1]);
          cambio=1;
       }
   }
}
```

4.1.9. CADENAS DE CARACTERES

4.1.9.1. INTRODUCCIÓN

Una cadena de caracteres es un secuencia ordenada de caracteres de longitud variable. Permiten trabajar con datos como apellidos, direcciones, etc. En C podemos encontrar dos alternativas para representar este tipo de información:

- **El tipo *char ** que son las cadenas de caracteres propias de C que se introdujeron en el primer tema.**

```
char *mensaje;


mensaje = "Esto es un literal de cadena de caracteres";
printf(mensaje);
```

- **Un vector de `char`.**

```
char mensaje[10];
```

Para delimitar el final del vector NO se usa la típica variable `util_mensaje` sino un carácter especial `'\0'` que se le asignará a la última componente utilizada.
`'\0'` se conoce como *carácter nulo* o *marca de fin de cadena*

En general, declararemos:

```
char <identificador> [<tamaño>];
```

dónde el tamaño ha de ser el número de caracteres máximo a almacenar, más uno (para poder guardar el carácter nulo)

La asignación se realiza componente a componente, como cualquier otro vector.

```
char mensaje[10];

mensaje[0] = 'H';
mensaje[1] = 'o';
mensaje[2] = 'l';
mensaje[3] = 'a';
mensaje[4] = '\0';
```

'H'	'o'	'l'	'a'	'\0'	?	?	?	?	?

```
mensaje = "Hola"; /* <- Error de compilación. */
                  /*  mensaje es un vector !! */
```

```
char mensaje[4];

mensaje[0] = 'H';
mensaje[1] = 'o';
mensaje[2] = 'l';
mensaje[3] = 'a';
mensaje[4] = '\0';   /* Error lógico :-( */
```

```
char mensaje[5];

mensaje[0] = 'H';
mensaje[1] = 'o';
mensaje[2] = 'l';
mensaje[3] = 'a';
mensaje[4] = '\0';   /* :-) */
```

4.1.9.2. INICIALIZACIÓN Y ASIGNACIÓN

Al declarar un *string*, podemos darle un valor inicial como cualquier otro vector:

```
char mensaje[5] = {'H','o','l','a','\0'};
char no_cabe[4] = {'H','o','l','a','\0'};
                                    /* Warning o error */
```

Incluso podemos suprimir la dimensión (ya vimos que esto es válido para cualquier vector, siempre que lo inicialicemos en la declaración)

```
char mensaje[] = {'H','o','l','a','\0'};
```

El compilador reserva automáticamente 5 componentes.

Para los vectores de `char`, C permite inicializarlos con un *literal de cadena de caracteres*:

```
char mensaje[5] = "Hola";
char mensaje[]  = "Hola";
char no_cabe[4] = "Hola";  /* <- Warning o error */
```

El compilador pondrá automáticamente el carácter nulo al final.

Esta asignación sólo puede hacerse en la inicialización:

```
char mensaje[5] = "Hola";
char otro_mensaje[5];

otro_mensaje = "Hola";  /* Error de compilación
                        vector = literal cadena caracteres */

otro_mensaje = mensaje; /* Error de compilación
                           Asignación entre vectores */
```

4.1.9.3. USO DE FUNCIONES

Como cualquier otro vector (salvo que no pasamos las componentes útiles)

```
int longitud(const char cadena[]){
   int i=0;
   while (cadena[i]!='\0')
      i++;

   return i;
}
int main(){
  char mensaje[] = "Hola";
  char vacio[20];

  printf("%d", longitud(mensaje)); /* Imprime 4 */
  printf("%d", longitud(vacio)); /* A saber cuando para! :-( */
}
```

Ejercicio. **Construir una función que dada una cadena `cad` y un carácter `buscado`, devuelva la posición que ocupa la primera ocurrencia del carácter `buscado` en la cadena `cad`. Devolver -1 si el carácter no está en la cadena.**

4.1.9.4. ASIGNACIÓN ENTRE STRING

```
char mensaje[5] = "Hola";
char otro_mensaje[5];

otro_mensaje = mensaje; /* Error de compilación
                           Asignación entre vectores!! */
```

```
void CopiaCadena(char destino[], const char origen[]){
   int i;
   for (i=0 ; origen[i]!='\0'; i++)
      destino[i] = origen[i];
   destino[i] = '\0';
}
int main(){
   char mensaje[5] = "Hola";
   char otro_mensaje[5];

   CopiaCadena(otro_mensaje, mensaje);  /* :-) */
   .....
}
```

```
int main(){
  char mensaje[5];
  char otro_mensaje[5];

  mensaje[0] = 'H';
  CopiaCadena(otro_mensaje, mensaje);  /* Error lógico grave */
```

Importante. **Es tarea del programador asegurar que el tamaño de la cadena destino es suficiente para realizar la copia de la cadena fuente**

La biblioteca `string.h` **contiene la definición de ésta y muchas otras funciones para trabajar con vectores de char:**

```
strcpy(char cadena1[], const char cadena2[]);
```

```
#include <string.h>
int main(){
   char mensaje[5] = "Hola";
   char otro_mensaje[5];

   strcpy(otro_mensaje, mensaje);
   .....
}
```

Otras funciones de la biblioteca `string`**:**

- ▷ `int strlen(const char s[])` **que devuelve la longitud de la cadena** `s`**.**
- ▷ `strcat(char s1[], const char s2[])` **que concatena las cadenas** `s1` **y** `s2` **y el resultado se almacena en** `s1`**.**
- ▷ `int strcmp(const char s1[], const char s2[])` **que compara las cadenas** `s1` **y** `s2`**. Si la cadena** `s1` **es menor que** `s2` **devuelve un valor menor que cero, si son iguales devuelve 0 y en otro caso devuelve un valor mayor que cero.**

4.1.9.5. ESCRITURA Y LECTURA DE CADENAS STRING

Con un vector cualquiera NO podemos hacer lo siguiente:

```
int vector[30] = {1,2,3};

printf("", vector);   /* Error compilación */
scanf("", vector);    /* Error compilación */
```

Sin embargo, con un vector de `char` sí podemos usar `printf`. No presenta ningún problema:

```
char nombre[30] = "Juan Carlos Rey";

printf(nombre);   /* Imprime Juan Carlos Rey */
```

Sin embargo, la lectura con `scanf` es más problemática:

```
char nombre[30];

scanf("%s", nombre);
printf("El nombre introducido es: %s", nombre);
```

Si la cadena que se desea introducir contiene separadores (espacios y/o tabuladores), la variable contendrá sólo los caracteres hasta el primer separador.

Para arreglarlo, usaremos `gets`. Pero previamente debemos entender cómo se realiza la entrada de datos.

Lo siguiente es aplicable a lecturas del teclado, ficheros de texto, u otros dispositivos que puedan crear un flujo de caracteres.

- **Las entradas se realizan a través de un *buffer* de datos intermedio.**
- **Existe un *cursor* o *apuntador* que indica el sitio exacto del buffer dónde se realizará la próxima lectura.**
- **Cada petición de una lectura de datos, como por ejemplo:**

```
scanf("%d", dato);
gets(cadena);
```

  cogerá un dato desde el buffer, a partir de dónde esté situado el cursor. La cantidad de datos que coge dependerá del tipo de dato que se vaya a leer y de la orden usada para leer.
- **Si el buffer está vacío, el sistema lo llena pidiendo datos desde el teclado, desde un fichero, o en general, del dispositivo usado por defecto.**

▶ *Comportamiento de* `scanf(" %d", &entero)`

▷ **Lee el entero (4 bytes usualmente) que hay a partir del cursor, saltándose, previamente, todos los separadores (espacios en blanco, tabuladores y retornos de carro) que hubiese al principio del buffer. Dichos separadores se eliminan del buffer.**
Una vez leído el entero, el cursor apunta al byte siguiente.

```
buffer = <b><b>\n<b>35\n45
scanf("%d", &entero);    =>  entero = 35
buffer = \n45
```

Nota: El símbolo `<b>` **representa un espacio en blanco.**

▷ **Si el dato no corresponde a un entero, se lee un valor basura.**

```
buffer = <b><b>\n<b>a\n45


entero = 7;
scanf("%d", &entero);    =>  entero = 20013391348
buffer = \n45
```

▷ **Si el buffer está vacío, se piden datos al dispositivo asociado.**

Si es desde el teclado, se leen datos hasta que el usuario pulsa ENTER (el cual también pasa al buffer)

```
scanf("%d", &entero);

buffer = vacío -> Se pide al dispositivo
Dispositivo -> <b><b>\n
buffer = <b><b>\n
  La ejecución de scanf("%d", &entero); borra
  todos los separadores del buffer
  Como todavía no ha leído un entero,
  se piden datos al dispositivo
buffer = vacío -> Se pide al dispositivo
Dispositivo -> <b>35<b>a\n
buffer = <b>35<b>a\n
entero = 35
buffer = <b>a\n
```

▶ *Comportamiento de* `scanf( "%c", &caracter)`

Análogo al anterior, es decir:

▷ **Lee el carácter (1 byte usualmente) que hay a partir del cursor, no se salta los separadores que hubiese al principio del buffer.**
Una vez leído el carácter, el cursor apunta al byte siguiente.

```
char caracter;
scanf("%c", &caracter);

buffer = vacío -> Se pide al dispositivo
Dispositivo -> <b><b>\n

scanf("%c", &caracter);
caracter = ' '
buffer = <b>\n
```

Otro ejemplo sería:

```
buffer = <b>3<b>a\n

scanf("%c", &caracter);
caracter = ' '
buffer = 3<b>a\n
```

▶ *Comportamiento de* `scanf(" %c", &caracter)` *continuación*

- ▷ **Cada llamada a** `scanf("%c", &caracter)` **lee un carácter del buffer, lo elimina (del buffer) y el cursor pasa al siguiente carácter.**
- ▷ **Si hay un separador lo lee tal cual.**

```
char caracter;

buffer = <b>a3\n<b>35\n45
scanf("%c", &caracter);
caracter = ' '
buffer = a3\n<b>35\n45

buffer = a3\n<b>35\n45
scanf("%c", &caracter);
caracter = 'a'
buffer = 3\n<b>35\n45

buffer = 3\n<b>35\n45
scanf("%c", &caracter);
caracter = '3'
buffer = \n<b>35\n45

buffer = \n<b>35\n45
scanf("%c", &caracter);
caracter = '\n'
buffer = <b>35\n45
```

Por ejemplo, el siguiente bucle lee todos los ENTER que haya al principio:

```
char caracter;
do{
   scanf("%c", &caracter);
}while (caracter == '\n');
```

▷ **Si se lee desde un fichero y se ha llegado al final, la lectura del final del fichero con** `scanf("%c", &caracter)` **devuelve EOF cuando no es fin de fichero devuelve el número de campos asignados.**

Obsérvese que `scanf()` **se declara como una función que devuelve un** `int`**.**

▶ *Comportamiento de* `scanf(" %s", string)`

Análogo al anterior, es decir:

- ▷ **Lee la secuencia de caracteres que hay a partir del cursor, saltándose, previamente, todos los separadores que hubiese al principio del buffer (dichos separadores se eliminan del buffer). La lectura para cuando se ha llegado a otro separador, o cuando se llega al final del buffer.**
  Una vez leída la secuencia de caracteres, el cursor apunta al byte siguiente.
- ▷ **El compilador añade automáticamente el carácter** `'\0'` **al final del** `string`

```
char nombre[30];

Dispositivo -> <b><b>Juan<b>Carlos<b>Rey\n
scanf("%s", nombre);
nombre = "Juan"  ('J''u''a''n''\0')
buffer = <b>Carlos<b>Rey\n
```

▶ *Comportamiento de* `char *gets(char *s)`

Se van leyendo caracteres desde el buffer (los espacios en blanco y los tabuladores se leen sin problemas) hasta que se verifica la condición de parada de abajo. Una vez leídos los caracteres, se suprimen del buffer, y se asignan a la variable `string`, añadiéndole automáticamente el carácter terminador `\0`.

***Condición de parada*:**

- ▷ **Se encuentre un ENTER, es decir, `'\n'`. Saca el ENTER del buffer, pero no lo introduce en la cadena. Al igual que siempre, se añade `\0`.**

```
char string[10];

buffer = \nHola\nAdios
gets(string);
string = \0?????????
buffer = Hola\nAdios
gets(string);
string = Hola\0?????
buffer = Adios
```

Si lo primero a leer es el ENTER, `gets` lo descarga del buffer, y en la cadena sólo se almacenará `\0` (en la posición 0)

En el caso de que se lean más caracteres de los que caben en `string`, el comportamiento es indeterminado

***Ejemplo*. Cread una función que lea desde el buffer un `string`, saltándose antes todos los ENTER (líneas vacías) que pudiese haber seguidos:**

```
void LeeCadena_SaltaENTERs(char cadena[]){
   do{
      gets(cadena);
   }while (cadena[0]=='\0');
}
```

4.1.9.6. EL CARÁCTER NULO

Para indicar el final de la cadena hemos usado el carácter `'\0'`. En C, este literal es el mismo que el literal `0`, por lo que los recorridos con cadenas también pueden hacerse usando `0`.

```
int longitud(const char cadena[]){
   int i=0;
   while (cadena[i]!=0)
      i++;

   return i;
}
```

4.2. MATRICES

4.2.1. MOTIVACIÓN

Supongamos una finca rectangular dividida en parcelas. Queremos almacenar la producción de aceitunas, en TM.

La forma natural de representar la parcelación sería usando el concepto matemático de matriz.

9.1	**0.4**	**5.8**
4.5	**5.9**	**1.2**

Para representarlo en C podríamos usar un vector `parcela`**:**

9.1	**0.4**	**5.8**	**4.5**	**5.9**	**1.2**

pero la forma de identificar cada parcela (por ejemplo `parcela[4]`**) es poco intuitiva para el programador.**

4.2.2. Declaración y Operaciones con matrices

4.2.2.1. Declaración

```
<tipo de dato><identificador>[DIM_FIL][DIM_COL];
```

Como con los vectores, el tipo base de la matriz es el mismo para todas las componentes, ambas dimensiones han de ser de tipo entero, y comienzan en cero.

```
#define DIM_FIL_parcela  2
#define DIM_COL_parcela  3
int main(){
   float parcela[DIM_FIL_parcela][DIM_COL_parcela];
}
```

4.2.2.2. Acceso y asignación

`<Ident>[<ind1>] [<ind2>]`

$$0 \leq \texttt{ind}_i < \texttt{Dim}_i$$

`<Ident>[<ind1>] [<ind2>]` **es una variable más del programa y se comporta como cualquier variable del tipo de dato base de la matriz.**

Por ejemplo, para asignar una expresión a una celda:

<Ident>[<ind1>] [<ind2>] = <Expresión>;

<Expresión> **ha de ser del mismo tipo de dato que el tipo base de la matriz.**

Ejemplo.

```
#define DIM_FIL_parcela 2
#define DIM_COL_parcela 3
int main(){
 float parcela[DIM_FIL_parcela][DIM_COL_parcela];
 int valor = 7;

 parcela[0][0] = 4.5;  /* Correcto */
 parcela[0][1] = 5.9;  /* Correcto */
 parcela[2][0] = 7.2;  /* Error lógico: Fila 2 No existe */
 parcela[0][3] = 7.2;  /* Error lógico: Columna 3 No existe */
 parcela[0][0] = 4;    /* Correcto*/
                       /* Casting automático: float=int */
}
```

4.2.2.3. GESTIÓN DE COMPONENTES ÚTILES CON MATRICES

Por cada dimensión necesitamos gestionar una variable que indique el número de componentes útiles.

```
#include <stdio.h>
#define DIM_FIL_parcela 5
#define DIM_COL_parcela 8

int main(){
 int utilFil_parcela, utilCol_parcela;
 int fil, col;
 float parcela[DIM_FIL_parcela][DIM_COL_parcela];
 float produccion;

 utilFil_parcela=2;
 utilCol_parcela=3;

 for (fil=0 ; fil<utilFil_parcela ; fil++)
   for (col=0 ; col<utilCol_parcela ; col++){
     printf("Introduce produccion en TM de la parcela %d,%d ",
              fil, col);
     scanf("%f", &produccion);
     parcela[fil][col]= produccion;
   }

 printf("\nDatos introducidos:\n");
```

```
for (fil=0 ; fil<utilFil_parcela ; fil++){
  for (col=0 ; col<utilCol_parcela ; col++){
    printf("\nProduccion en TM de la parcela %d,%d --> ",
             fil, col);
    printf("%f", parcela[fil][col]);
  }
  printf("\n");
}
```

***Nota*. Podemos poner directamente:**

```
scanf("%f", &parcela[fil][col])
```

Ejemplo. Buscar un elemento en una matriz.

```
#include <stdio.h>
#define DIM_FIL_m  2
#define DIM_COL_m  3
int main(){
  float m[DIM_FIL_m][DIM_COL_m];
  int FilEncontrado, ColEncontrado, utilFil_m,
      utilCol_m, fil, col;
  float buscado;
  int encontrado;

  do{
     printf("Introducir el numero de filas: ");
     scanf("%d", &utilFil_m);
  }while ( (utilFil_m<1) || (utilFil_m>DIM_FIL_m) );

  do{
     printf("Introducir el numero de columnas: ");
     scanf("%d", &utilCol_m);
  }while ( (utilCol_m<1) || (utilCol_m>DIM_COL_m) );
```

```
for (fil=0 ; fil<utilFil_m ; fil++)
   for (col=0 ; col<utilCol_m ; col++){
      printf("Introducir el elemento (%d,%d): ",
            fil, col);
      scanf("f", &m[fil][col]);
   }
printf("\nIntroduzca elemento a buscar: ");
scanf("%f", &buscado);

encontrado=0;

for (fil=0; !encontrado && (fil<utilFil_m) ; fil++){
   for (col=0; !encontrado && (col<utilCol_m) ; col++)
      if (m[fil][col] == buscado){
         encontrado    = 1;
         FilEncontrado = fil;
         ColEncontrado = col;
      }
   }
}
if (encontrado)
   printf("\nencontrado en la posicion %d,%d\n",
         FilEncontrado, ColEncontrado);
else
   printf("\nElemento no encontrado\n");
```

Importante: la variable `encontrado` **ha de ser la misma en los dos bucles** `for`**.**

4.2.2.4. INICIALIZACIÓN

En la declaración de la matriz se pueden asignar valores a toda la matriz. Posteriormente, no es posible: es necesario acceder a cada componente independientemente.

La forma *segura* es poner entre llaves los valores de cada fila.

```
int parc[2][3]={{1,2,3},{4,5,6}};  /* parc tendrá: 1 2 3 */
                                   /*              4 5 6 */
```

Si no hay suficientes inicializadores para una fila determinada, los elementos restantes se inicializan a 0.

```
int parc[2][3]={{1},{3,4,5}};   /* parc tendrá: 1 0 0 */
                                /*              3 4 5 */
```

Si se eliminan las llaves que encierran cada fila, se inicializan los elementos de la primera fila y después los de la segunda, y así sucesivamente.

```
int parc[3][4]={1, 2, 3, 4, 5} /* parc tendrá: 1 2 3 4 */
                               /*              5 0 0 0 */
                               /*              0 0 0 0 */
```

4.2.2.5. REPRESENTACIÓN EN MEMORIA

Todas las posiciones de una matriz están realmente contiguas en memoria. La representación exacta depende del lenguaje. En C se hace por filas:

m

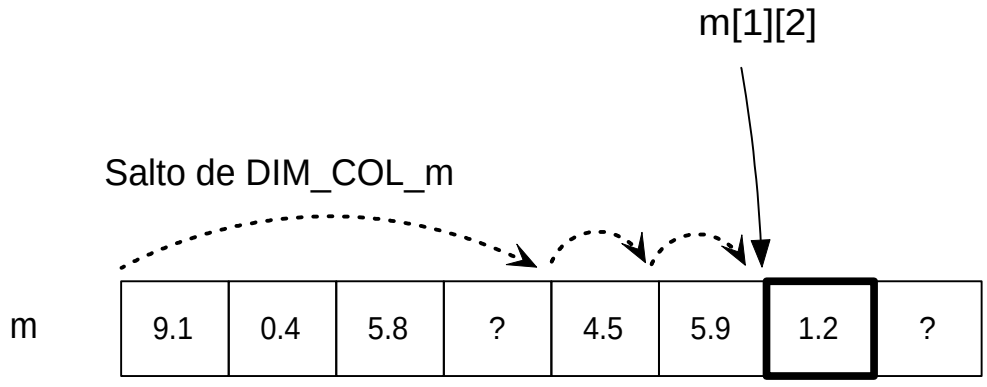
9.1	0.4	5.8	?
4.5	5.9	1.2	?

m

9.1	0.4	5.8	?	4.5	5.9	1.2	?

`m` contiene la dirección de memoria de la primera componente.

Para calcular dónde se encuentra la componente `m[1][2]` el compilador debe *pasar a la segunda fila*. Lo consigue trasladándose tantas posiciones como diga `DIM_COL_m`, a partir del comienzo de `m`. Una vez ahí, *salta* 2 posiciones y ya está en `m[1][2]`.



Conclusión: Para saber dónde está `m[i][j]`, el compilador necesita saber cuánto vale `DIM_COL_m`, pero no `DIM_FIL_m`. Para ello, da tantos saltos como indique la expresión:

```
i*DIM_COL_m + j
```

4.2.3. MODULARIZACIÓN CON MATRICES

Para pasar una matriz bidimensional hay que especificar en la cabecera `DIM_COL_m`

```
void ImprimeMatriz(double m[][DIM_COL_m] , int utilFil_m,
                   int utilCol_m)
```

Pero entonces, `DIM_COL_m` **no puede ser local a** `main`**. Debe ser un define global** `:-(`**. No hay otra solución.**

> ***Al pasar una matriz como parámetro, debemos incluir la constante de dimensión de las columnas. Además, ésta ha de ser un define o un literal.***

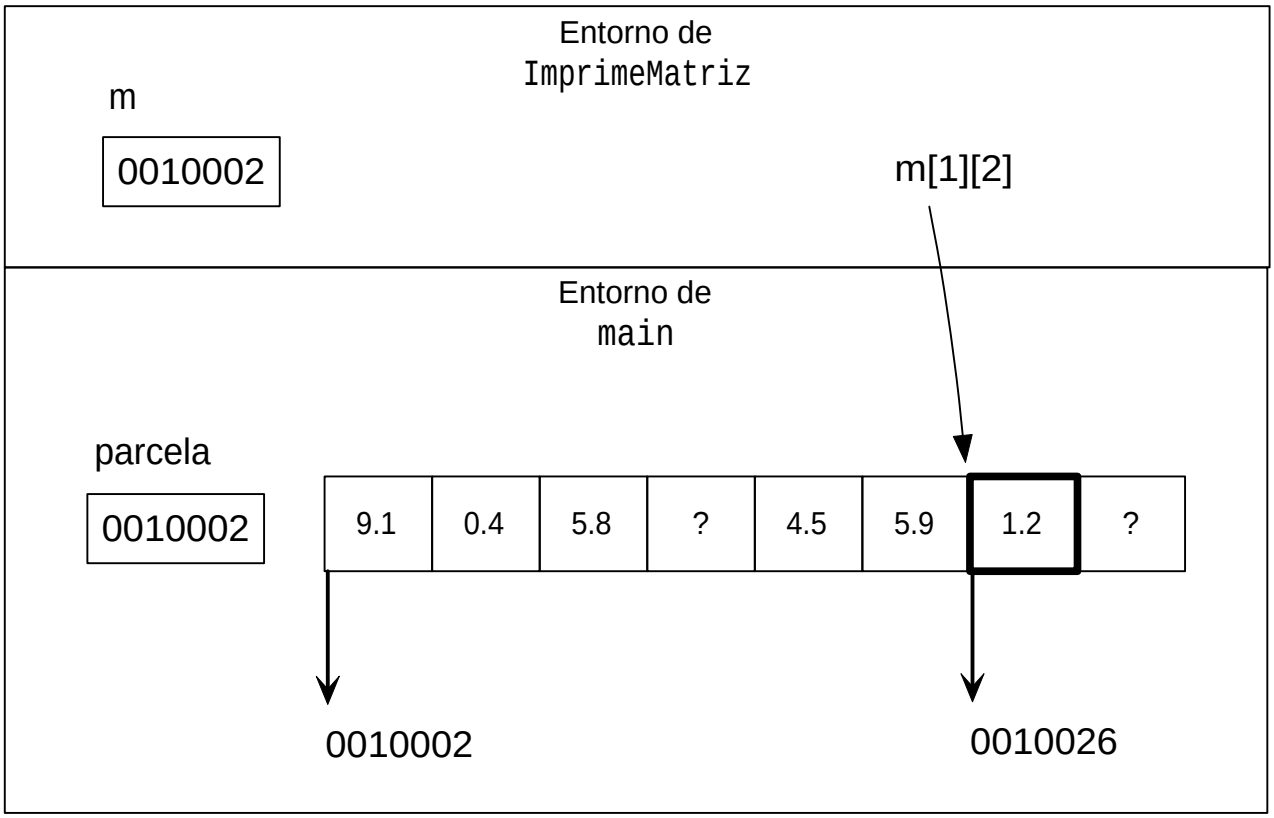
Al igual que ocurría con los vectores, al pasar una matriz como parámetro, se copia la dirección de memoria que contiene la matriz pasada como parámetro actual.

```
#include <stdio.h>

#define DIM_FIL 5
#define DIM_COL 4

void ImprimeMatriz(const float m[][DIM_COL] , int utilFil,
                   int utilCol){
   int fil,col;

   for (fil=0 ; fil<utilFil ; fil++){
      for (col=0 ; col<utilCol ; col++)
         printf("%f ", m[fil][col]);

      printf("\n");
   }
}

int main(){
   float parcela[DIM_FIL][DIM_COL]={{5.4,3},{6,7},{1,2.4}};
   int utilFil_parcela=3, utilCol_parcela=2;

   ImprimeMatriz(parcela , utilFil_parcela , utilCol_parcela);
}
```

Dentro de la función, para acceder a `m[fil][col]` el compilador dará tantos saltos como indique la operación

```
   fil*DIM_COL + col
```

Esta es la razón por la que hay que incluir `DIM_COL` en el prototipo de la función.


Entorno de
ImprimeMatriz
m
0010002
m[1][2]
Entorno de
main
parcela
0010002
9.1
0.4
5.8
?
4.5
5.9
1.2
?
0010002
0010026

Ejemplo. Buscar un elemento en una matriz.

```
#include <stdio.h>

#define DIM_FIL  5
#define DIM_COL  4

void BuscaEnMatriz(const float m[][DIM_COL],
          int utilFil, int utilCol,
          float buscado, int *encontrado,
          int *fil_encontrado, int *col_encontrado){
   int fil;
   int col;

   *encontrado=0;

   for (fil=0; !*encontrado && (fil<utilFil) ; fil++){
      for (col=0; !*encontrado && (col<utilCol) ; col++)
         if (m[fil][col] == buscado){
            *encontrado    = 1;
            *fil_encontrado = fil;
            *col_encontrado = col;
         }
  }
}

int main(){
  float m[DIM_FIL][DIM_COL];
  int fil, col, utilFil_m, utilCol_m,
      posFil, posCol;
  float elemento;
```

```
  int encontrado;

  do{
     printf("Introduce el numero de filas: ");
     scanf("%d", &utilFil_m);
  }while ( (utilFil_m<1) || (utilFil_m>DIM_FIL) );

  do{
     printf("Introduce el numero de columnas: ");
     scanf("%d", &utilCol_m);
  }while ( (utilCol_m<1) || (utilCol_m>DIM_COL) );

  for (fil=0; fil<utilFil_m; fil++)
     for (col=0; col<utilCol_m; col++){
        printf("\nIntroduce el %d,%d: ", fil, col);
        scanf("%f", &m[fil][col]);
     }

  printf("\n\nIntroduzca elemento a buscar: ");
  scanf("%f", &elemento);


  BuscaEnMatriz(m, utilFil_m, utilCol_m, elemento,
                &encontrado, &posFil, &posCol);

  if (encontrado)
     printf("\nEncontrado en la posición %d,%d",
           posFil, posCol);
  else
     printf("\nElemento no encontrado\n");
}
```

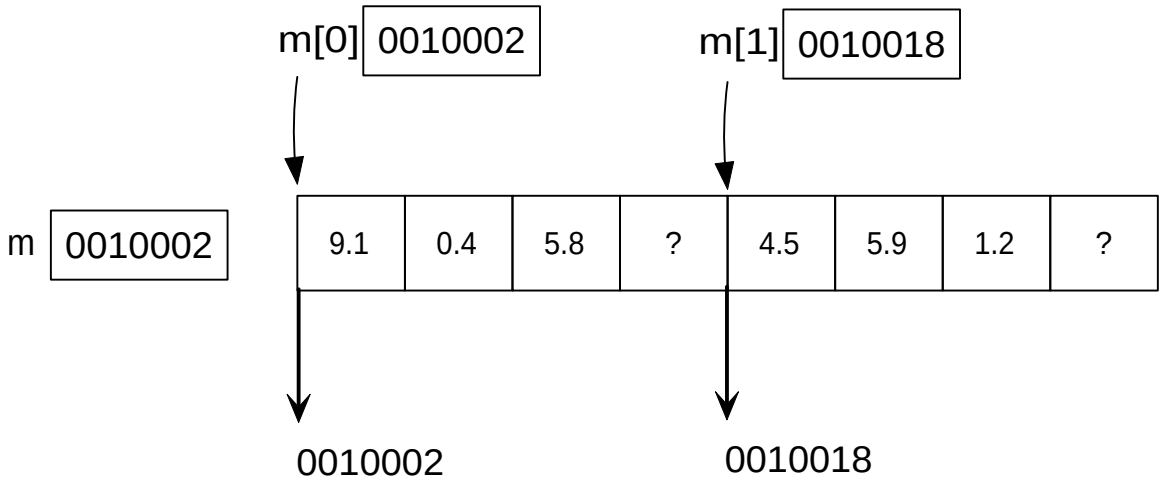
4.2.4. GESTIÓN DE FILAS DE UNA MATRIZ COMO VECTORES

Internamente, una matriz es una serie de posiciones contiguas en memoria. Como cada fila también es una serie de posiciones contiguas en memoria, podemos decir que una fila de una matriz es un vector.

¿Cómo se llama la fila `i` de una matriz `m`? `m[i]`

```
Fila 0 -> m[0]
Fila 1 -> m[1]
Fila 2 -> m[2]
```

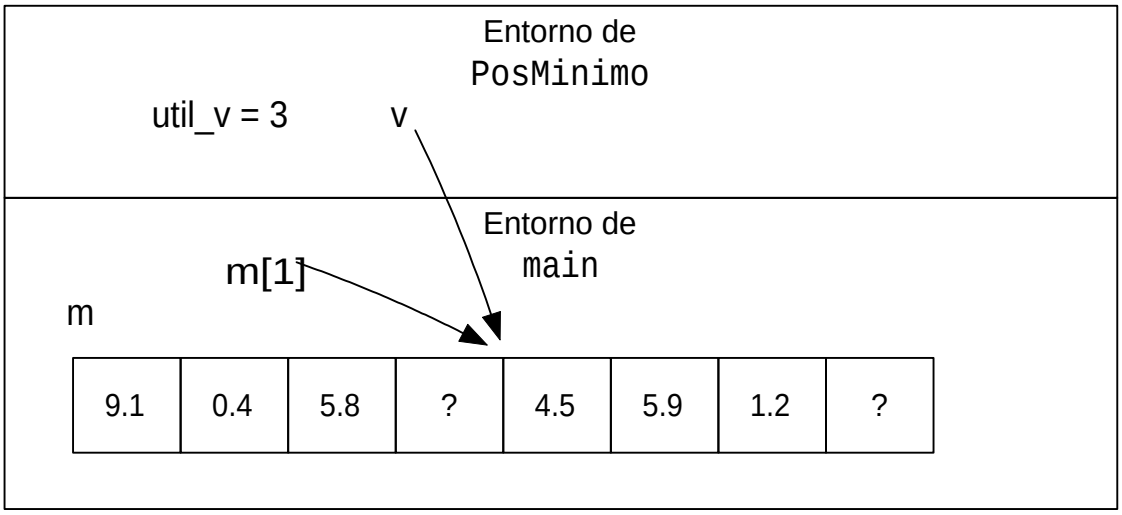
Formalmente, `m[i]` es la dirección de la primera componente de la fila `i`.



¿Cuántas componentes utilizadas tiene el vector `m[i]`? `util_Col_m`

Ejemplo. Calcular la posición del mínimo elemento de una fila.

```
#define DIM_FIL  2
#define DIM_COL  4

int PosMinimo (const float v[], int util_v){ /* Vector ! */
   int posicion_minimo;
   float minimo;
   int i;

   minimo = v[0];
   posicion_minimo = 0;

   for (i=1; i<util_v ; i++)
      if (v[i] < minimo){
         minimo = v[i];
         posicion_minimo = i;
      }
   return posicion_minimo;
}
int main(){
   float m[DIM_FIL][DIM_COL];
   int fil, col, utilFil_m = 2, utilCol_m = 3, pos_min_fila;

   <Rellenar componentes de m>

   /* Calculamos el mínimo de la segunda fila */
   pos_min_fila = PosMinimo (m[1], utilCol_m);
}
```


Entorno de
PosMinimo
util_v = 3
v
Entorno de
main
m[1]
m
9.1
0.4
5.8
?
4.5
5.9
1.2
?

Ejemplo. **Ordenar la fila `4` de una matriz bidimensional.**

```
void OrdSeleccion (float v[], int util_v){
   ..............
}

void ImprimeVector(const float vector[], int util_vector){
   ..............
}
int main(){
  float m[DIM_FIL][DIM_COL];
  int utilFil_m, utilCol_m,
      Filencontrado, Colencontrado;

  <Rellenar datos de m>
  OrdSeleccion (m[3],utilCol_m);
  ImprimeVector(m[3],utilCol_m);
}
```

Nota. **Obviamente, no ocurre lo mismo con las columnas de una matriz.**

Ejercicio. **Supongamos que queremos ordenar todos los elementos de una matriz. Por ejemplo:**

```
3 5 7        1 2 3
2 1 9  -->   4 5 5
5 6 4        6 7 9
```

¿Qué condiciones se han de dar para poder llamar simplemente al procedimiento de ordenación de un vector en la forma `OrdSeleccion(m[0],Tope)`?

Ejemplo. **Hallar el máximo elemento de los mínimos de cada fila de una matriz.**

```
3 5 7        3 |
2 1 9  -->   1 | --> 4
5 6 4        4 |
```

▶ ***Primera aproximación***

```
Ordenar todas las filas
Coger el primer elemento de cada fila
y calcular el máximo de todos ellos.
```

***Inconvenientes*:**

▷ **Recordad el principio de programación: *JAMÁS modificaremos los datos originales si no se indica explícitamente***

▷ **Ordenar un vector es más lento que buscar el mínimo.**

▶ ***Segunda aproximación***

```
Calcular el mínimo de cada fila y almacenarlo en
un vector de mínimos
Calcular el máximo de dicho vector.
```

▷ **El vector de mínimos tendrá tantas componentes como diga `utilFil`. Pero como no podemos dimensionar con una variable tendremos que usar `DIM_FIL`**

▷ **Usamos las funciones `PosMinimo` y `PosMaximo`, que ya construimos en el apartado de Ordenación.**

▷ **Debemos inicializar la primera componente del vector de mínimos al mínimo de la primera fila.**

```
float MaxMin (const float m[][DIM_COL],
   int utilFil, int utilCol){

   float minimos[DIM_FIL]; /* viene del define! */
   int pos_minimo, pos_maximo;
   int fil;

   for (fil=0 ; fil<utilFil ; fil++){
      pos_minimo = PosMinimo(m[fil],utilCol);
      minimos[fil] = m[fil][pos_minimo];
   }

   pos_maximo = PosMaximo(minimos,utilFil);
   return minimos[pos_maximo];
}
```

Mejoras:

- ▷ **Modificar para devolver las posiciones (fila y columna) dónde se alcanza el valor.**
- ▷ **No necesitamos un vector de mínimos. Podemos ir calculando el máximo de los mínimos poco a poco.**

▶ *Tercera aproximación*

```
Inicializar Mayor al mínimo de la primera fila
Para el resto de las filas:
   Calcular el mínimo de cada fila
   Actualizar Mayor, en su caso.
```

```
float MaxMin (const float m[][DIM_COL],
              int utilFil, int utilCol){
   float maximo, minimo;
   int pos_minimo, pos_maximo;
   int fil;

   pos_minimo = PosMinimo(m[0],utilCol);
   maximo = m[0][pos_minimo];

   for (fil=1 ; fil<utilFil ; fil++){
      pos_minimo = PosMinimo(m[fil],utilCol);
      minimo = m[fil][pos_minimo];

      if (minimo > maximo)
         maximo = minimo;
   }

   return maximo;
}
```

Mejoras:

- **En cuanto encontremos un elemento de una fila menor que el máximo hasta ese momento, podemos parar la búsqueda en esa fila.**

```
3 5 7        3
2       -->  2 < 3  => Parar la búsqueda en esta fila
5 6 4        4
```

Ejercicio. Calcular la traspuesta de una matriz.

4.2.5. MATRICES DE VARIAS DIMENSIONES

Podemos declarar tantas dimensiones como queramos. Sólo es necesario añadir más corchetes.

Por ejemplo, para representar la producción de una finca dividida en 2×3 parcelas, y dónde en cada parcela se practican cinco tipos de cultivos, definiríamos:

```
#define DIV_HOR  2
#define DIV_VERT 3
#define TOTAL_CULTIVOS 5

float parcela[DIV_HOR][DIV_VERT][TOTAL_CULTIVOS];

   /* Asignación al primer cultivo de la parcela 1,2 */

   parcela[1][2][0] = 4.5;
```

En general, para una matriz n-dimensional hay que especificar todas las dimensiones excepto la primera.

```
#define DIM_FIL 3
#define DIM_COL 2
#define DIM_PROF 5
void ImprimeMatriz(const float m[][DIM_COL][DIM_PROF] ,
                   int utilFil, int utilCol, int utilProf){
   int fil,col,prof;

   for (fil=0 ; fil<utilFil ; fil++){
      for (col=0 ; col<utilCol ; col++){
         for (prof=0 ; prof<utilProf ; prof++)
            printf("%f ", m[fil][col][prof]);
         printf("\n");
      }
      printf("\n");
   }
}
```