

# Tema III

## Vectores y Matrices

### *Objetivos:*

- ▷ Introduciremos los vectores de C++ que nos permitirán almacenar conjuntamente datos del mismo tipo.
- ▷ Veremos varios algoritmos de búsqueda y ordenación.
- ▷ Introduciremos las matrices (vectores de varias dimensiones).

## III.1. Fundamentos

### III.1.1. Introducción

#### III.1.1.1. Motivación

**Ejemplo.** Lea tres notas desde teclado y diga cuántos alumnos superan la media

```
#include <iostream>
using namespace std;

int main(){
    int superan_media;
    double nota1, nota2, nota3, media;

    cout << "Introduce nota 1: ";
    cin >> nota1;
    cout << "Introduce nota 2: ";
    cin >> nota2;
    cout << "Introduce nota 3: ";
    cin >> nota3;

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

    if (nota1 > media)
        superan_media++;

    if (nota2 > media)
        superan_media++;
```



```
if (nota3 > media)
    superan_media++;

cout << superan_media << " alumnos han superado la media\n";
}
```

**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.

Un **vector (array)** 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. El índice será un entero, siendo el primero el 0.

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

### III.1.1.2. Declaración

`<tipo> <identificador> [<núm. componentes>];`

- ▷ `<tipo>` indica el tipo de dato común a todas las **componentes (elements/components)** del vector.
- ▷ `<núm. componentes>` determina el número de componentes del vector, al que llamaremos **tamaño (size)** del vector. El número de componentes debe conocerse en el momento de la declaración y no es posible alterarlo durante la ejecución del programa. Pueden usarse literales ó constantes enteras (`char`, `int`, ...), pero nunca una variable (en C sí, pero no en C++)

***No puede declararse un vector con un tamaño variable***

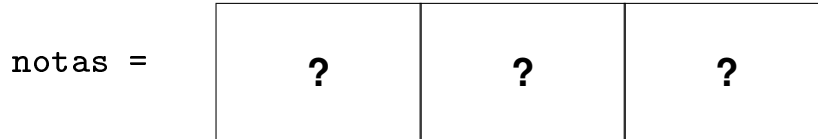
```
int main(){
    const int MAX_ALUMNOS = 3;
    double notas[MAX_ALUMNOS];
    int variable = 3;
    double notas[variable]; // Error de compilación
```

**Consejo:** *Fomente el uso de constantes en vez de literales para especificar el tamaño de los vectores.*



- ▷ Las componentes ocupan posiciones contiguas en memoria.

```
const int MAX_ALUMNOS = 3;  
double notas[MAX_ALUMNOS];
```



### Otros ejemplos:

```
int main(){  
    const int NUM_REACTORES = 20;  
  
    int    TemperaturasReactores[NUM_REACTORES]; // Correcto.  
    bool   casados[40];                          // Correcto.  
    char   NIF[8];                                // Correcto.  
    .....  
}
```

## III.1.2. Operaciones básicas

### III.1.2.1. Acceso

Dada la declaración:

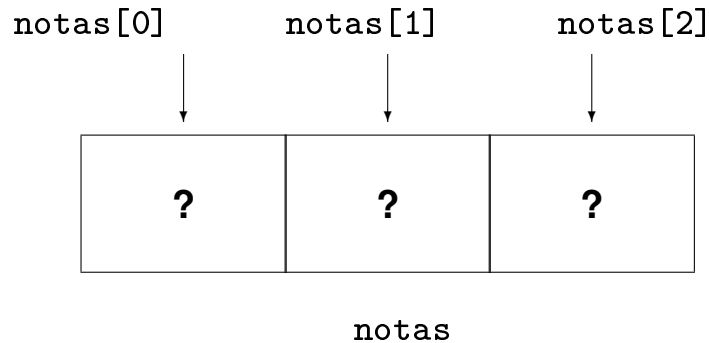
<tipo> <identificador> [<núm. componentes>];

A cada componente se accede de la forma:

<identificador> [<índice>]

- ▷ El índice de la primera componente del vector es 0.  
El índice de la última componente es <núm. componentes> - 1

```
const int TOTAL_ALUMNOS = 3;  
double notas[TOTAL_ALUMNOS];
```



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

- Podemos acceder a cualquier componente en cualquier momento: diremos que es un **acceso directo (direct access)**, también conocido como **acceso aleatorio (random access)**.
- Cada componente **es una variable** más del programa, del tipo indicado en la declaración del vector.

Por ejemplo, `notas[0]` es una variable de tipo `double`.

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

*Las componentes de un vector son datos como los vistos hasta ahora, por lo que le serán aplicables las mismas operaciones definidas por el tipo de dato asociado.*

### III.1.2.2. Asignación

- ▷ **No se permiten asignaciones globales sobre todos los elementos del vector (salvo en la inicialización, como veremos posteriormente). Las asignaciones se deben realizar componente a componente.**

```
int main(){
    const int MAX_ALUMNOS = 3;
    double notas[MAX_ALUMNOS];

    notas = {1,2,3};    // Error de compilación
```

- ▷ **Asignación componente a componente:**

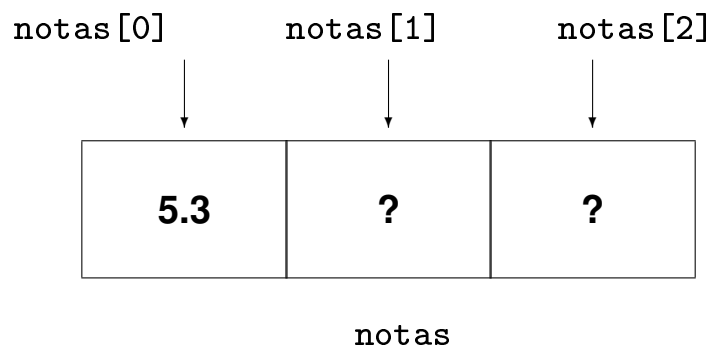
**<identificador> [<índice>] = <expresión>;**

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

```
int main(){
    const int MAX_ALUMNOS = 3;
    double notas[MAX_ALUMNOS];

    notas[0] = 5.3;

    int i = 0;
    notas[i] = 5.3;
```



```
int main(){
    const int MAX_ALUMNOS = 3;
    double notas[MAX_ALUMNOS];
    double una_nota;

    notas[0] = 5.3;           // Correcto. double = double
    notas[1] = 7;             // Correcto. double = int
    una_nota = notas[1];      // Correcto. double = double
```

### III.1.2.3. Lectura y escritura

**La lectura y escritura se realiza componente a componente. Para leer con `cin` o escribir con `cout` *todas* las componentes utilizaremos un bucle:**

```
for (int i = 0; i < MAX_ALUMNOS; i++){
    cout << "Introducir nota del alumno " << i << ": ";
    cin >> notas[i];
}

media = 0;

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

media = media / MAX_ALUMNOS;

cout << "\nMedia = " << media;
}
```



### III.1.2.4. Inicialización

**C++ permite inicializar una variable vector en la declaración.**

▷ `int vector[3];`

**Los tres datos tienen un valor indeterminado**

`vector[0]=?, vector[1]=?, vector[2]=?`

▷ `int vector[3] = {4,5,6};`

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

▷ `int vector[7] = {8};`

***¡Cuidado!* Declara un vector de 7 componentes e inicializa la primera a 8 (sólo la primera) y el resto a cero.**

▷ `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[] = {1,3,9};`

**automáticamente el compilador asume** `int vector[3]`

**Si queremos declarar un vector de componentes constantes, pondremos el cualificador `const` en la definición del vector y a continuación habrá que, obligatoriamente, inicializarlo.**

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

## III.1.3. Trabajando con las componentes

### III.1.3.1. Representación en memoria

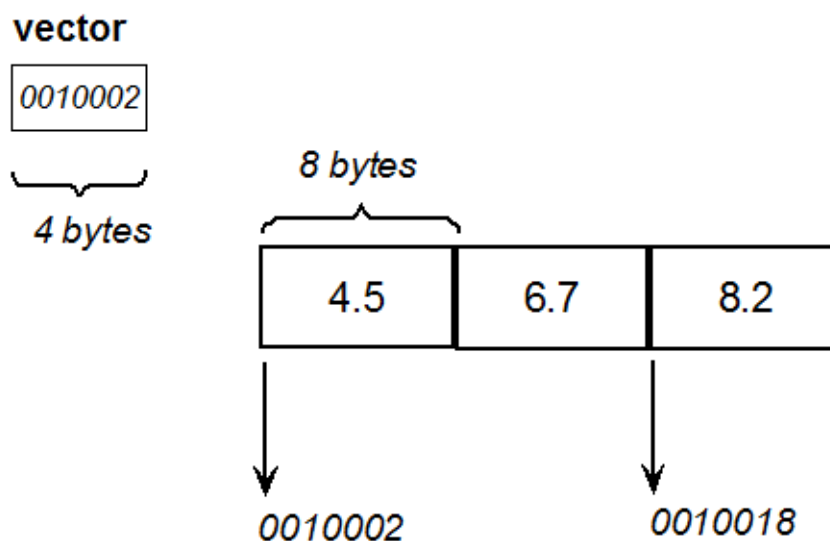
¿Qué ocurre si accedemos a una componente fuera de rango?

```
cout << notas[15];    // Posible error grave en ejecución!  
notas[15] = 5.3;      // Posible error grave en ejecución!
```



El compilador trata a un vector como un dato constante que almacena la dirección de memoria donde 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(){  
    double vector[3];  
  
    vector[0] = 4.5;  
    vector[1] = 6.7;  
    vector[2] = 8.2;
```



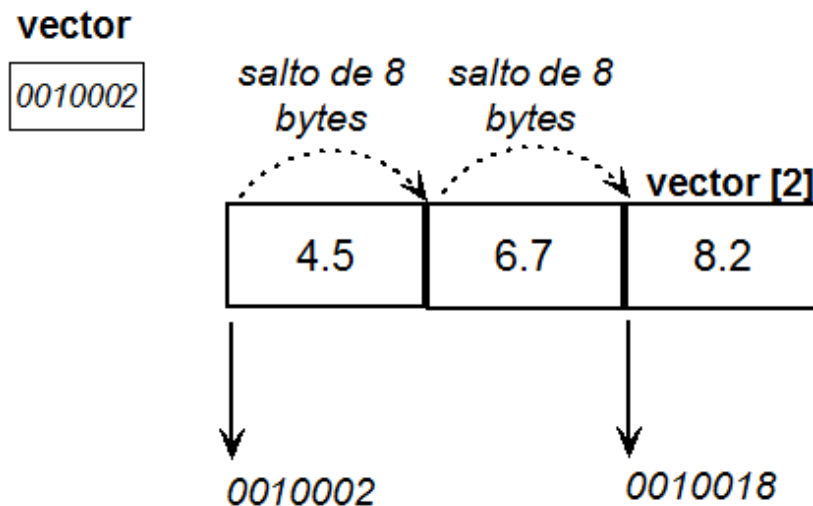
Es como si el programador hiciese la asignación siguiente:

```
vector = 0010002;
```

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

```
double vector[3];    // Compilador -> vector = 0010002;  
  
vector = 00100004;   // Error compilación. 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 8 bytes (suponiendo que los `double` ocupan 8 bytes). Por tanto, se va a la variable que empieza en la dirección `0010018`**



**En general, si declaramos**

```
TipoBase vector[TAMANIO];
```

**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`, es decir:**

$$\text{vector[indice]} \equiv \text{vector} + \text{indice} * \text{sizeof}(\text{<tipo base>})$$

```
int main(){  
    const int MAX_ALUMNOS = 3;  
    double vector[MAX_ALUMNOS];  
  
    vector[15] = 5.3;        // Posible error grave en ejecución!
```



***Para aumentar la eficiencia, el compilador no comprueba que el índice de acceso a las componentes esté en el rango correcto, por lo que cualquier acceso de una componente con un índice fuera de rango tiene consecuencias imprevisibles.***

---

### **Nota:**

El simple acceso a la componente para observar su valor, aún sin modificarla, puede ocasionar un error durante la ejecución. El estándar de C++ indica que un [acceso fuera de rango \(out of bound access\)](#) a una componente de un vector provocan un [comportamiento indeterminado \(undefined behaviour\)](#) .

¿Qué significa comportamiento indeterminado? Que el creador del compilador es libre de decidir la respuesta a dar ante esa situación. Puede decidir, por ejemplo, ¡formatear el disco duro!

---

### III.1.3.2. Gestión de componentes utilizadas

Dado un vector, ¿cómo gestionamos el uso de sólo una parte del mismo?

Supongamos el vector `notas` dimensionado para 50 alumnos.

```
const int MAX_ALUMNOS = 50;  
double vector[MAX_ALUMNOS];
```

Si sólo queremos usar parte de él, nos preguntamos dónde debemos colocar los huecos. Tenemos varias posibilidades:

► *Todos los huecos a la derecha*

Dejamos todos los huecos juntos (normalmente en la zona de índices altos, es decir, a la derecha) En este caso, basta usar una variable entera, `util` que indique el número de componentes usadas.

A partir de ahora  $X$  denotará una componente utilizada con un valor concreto. El caso anterior sería por tanto del tipo:

```
util_vector = 4
```

X	X	X	X	?	?	...	?
---	---	---	---	---	---	-----	---

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

```
for (int i=0; i < util_vector ; i++){  
    .....  
}
```

En el ejemplo de las notas:

```
util_notas = 3;  
  
for (int i = 0; i < util_notas; i++)  
    suma = suma + notas[i];
```

### ► Huecos en cualquier sitio

?	9.5	4.6	?	?	7.8	...	?
---	-----	-----	---	---	-----	-----	---

?	X	X	?	?	X	...	?
---	---	---	---	---	---	-----	---

¿Cómo recorreremos el vector? En el ejemplo de las notas:

```
for (int i = 0; i < MAX_ALUMNOS; i++)  
    suma = suma + notas[i];
```

Este bucle no nos vale ya que también procesa las componentes no asignadas (con un valor indeterminado)

Podemos usar otro vector de `bool` que nos indique si la componente está asignada o no:

```
bool utilizado_en_notas[MAX_ALUMNOS];
```

false	true	true	false	false	true	...	false
-------	------	------	-------	-------	------	-----	-------

```
for (int i = 0; i < MAX_ALUMNOS; i++){  
    if (utilizado_en_notas[i])  
        suma = suma + notas[i];  
}
```

Siempre que podamos, elegiremos la primera opción (todos los huecos a la derecha), para evitar trabajar con dos vectores dependientes entre sí (el vector que contiene los datos y el que indica los elementos utilizados).

Supongamos que podemos elegir un valor especial *nulo* ( $N$ ) del tipo de dato de las componentes para indicar que la componente no está siendo usada. Por ejemplo, en un vector de `string`  $N$  podría ser la cadena "", o -1 en un vector de positivos.

Entonces tendríamos las siguientes opciones:

► *Huecos en cualquier sitio y marcados con un valor especial*

$N$	$X$	$X$	$N$	$N$	$X$	...	$N$
-----	-----	-----	-----	-----	-----	-----	-----

El recorrido será del tipo:

```
for (int i = 0; i < MAX; i++){  
    if (vector[i] !=  $N$ ){  
        .....  
    }  
}
```

En el ejemplo de las notas:

```
const double NULO = -1.0;  
  
for (int i = 0; i < MAX_ALUMNOS; i++)  
    if (notas[i] != NULO)  
        suma = suma + notas[i];
```

► *Todos los huecos a la derecha. El primero, marcado con un valor especial*

Segunda opción. Los huecos se dejan al final (o al principio). Ponemos  $N$  en la última componente utilizada (debemos reservar siempre una componente para  $N$ ).

$X$	$X$	$X$	$X$	$N$	?	...	?
-----	-----	-----	-----	-----	---	-----	---

El recorrido será del tipo:

```
for (int i=0; vector[i] !=  $N$  ; i++){  
    .....  
}
```

En el ejemplo de las notas:

```
const double NULO = -1.0;  
i = 0;  
  
while (notas[i] != NULO){  
    suma = suma + notas[i];  
    i++;  
}
```



**Ejemplo.** Retomamos el ejemplo de las notas y leemos desde teclado el número de alumnos que vamos a procesar. Calculamos el número de alumnos que superan la media aritmética de las notas.

Optamos por una representación en la que todas las componentes usadas estén correlativas. Alternativas:

- ▷ Usar como valor nulo el -1, ya que no es una nota posible.

8.2	9.1	3.5	-1	?	?	...	?
-----	-----	-----	----	---	---	-----	---

- ▷ Usar una variable adicional que indique el número de alumnos actuales.

util_notas = 3							
8.2	9.1	3.5	?	?	?	...	?

Optamos por esta última.

```
int main(){
    const int MAX_ALUMNOS = 100;
    double notas[MAX_ALUMNOS];
    int util_notas;
    int superan_media;
    double media;

    cout << "Introduzca el número de alumnos (entre 1 y " +
            to_string(MAX_ALUMNOS) + "): ";

    do{
        cin >> util_notas;
    }while (util_notas <= 0 || util_notas > MAX_ALUMNOS);

    for (int i=0; i < util_notas; i++){
        cout << "nota[" << i << "] --> ";
```

```
        cin >> notas[i];
    }
    media = 0;

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

    media = media / util_notas;
    superan_media = 0;

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

    cout << "\n" << superan_media << " alumnos han superado la media\n";
}
```

[http://decsai.ugr.es/jccubero/FP/III\\_notas.cpp](http://decsai.ugr.es/jccubero/FP/III_notas.cpp)

**Ejemplo.** Representemos la ocupación de un autobús con un vector de cadenas de caracteres.

- ▷ Los asientos se numeran a partir de 1. El asiento 0 corresponde al conductor.
- ▷ Se permite que haya asientos no ocupados entre los pasajeros y permitimos acceder a cualquier componente (asiento) en cualquier momento. Para identificar un asiento vacío usamos  $N = ""$

"Pedro"	"Juan"	""	"Carlos"	""	""	...	"María"
---------	--------	----	----------	----	----	-----	---------

X	X	N	X	N	N	...	X
---	---	---	---	---	---	-----	---

```
int main(){
    const string TERMINADOR = "-";
    const string VACIO = "";
    const int MAX_PLAZAS = 50;
    string pasajeros[MAX_PLAZAS];      // C++ inicializa los string a ""
    string conductor, nombre_pasajero;
    int    asiento;

    for (int i=0; i < MAX_PLAZAS; i++)
        pasajeros[i] = VACIO;

    cout << "Autobús.\n";
    cout << "\nIntroduzca nombre del conductor: ";
    cin >> conductor;

    pasajeros[0] = conductor;

    cout << "\nIntroduzca los nombres de los pasajeros y su asiento."
        << "Termine con " << TERMINADOR << "\n";
    cout << "\nNombre: ";
```

```
cin >> nombre_pasajero;

while (nombre_pasajero != TERMINADOR){
    cout << "Asiento: ";
    cin >> asiento;

    pasajeros[asiento] = nombre_pasajero;

    cout << "\nNombre: ";
    cin >> nombre_pasajero;
}

cout << "\n\nConductor: " << pasajeros[0];

for (int i=1; i < MAX_PLAZAS; i++){
    cout << "\nAsiento número: " << i;

    if (pasajeros[i] != VACIO)
        cout << " Pasajero: " << pasajeros[i];
    else
        cout << " Vacío";
}
}
```

---

**Nota:**

Al ejecutar el programa, deben introducirse cadenas de caracteres sin espacios en blanco. Posteriormente veremos cómo leer cadenas sin esta restricción.

---

[http://decsai.ugr.es/jccubero/FP/III\\_autobus.cpp](http://decsai.ugr.es/jccubero/FP/III_autobus.cpp)

En resumen, los tipos más comunes de gestión de un vector son:

▷ Dejando huecos entre las componentes:

- Si podemos elegir un valor especial  $N$  que represente componente no utilizada.

$N$	$X$	$X$	$N$	$N$	$X$	...	$N$
-----	-----	-----	-----	-----	-----	-----	-----

Los recorridos deben comprobar si la componente actual es  $N$

- Si no podemos elegir dicho valor  $N$ , necesitaremos identificar las componentes no utilizadas usando, por ejemplo, un vector de `bool`.

?	$X$	$X$	?	?	$X$	...	?
---	-----	-----	---	---	-----	-----	---

false	true	true	false	false	true	...	false
-------	------	------	-------	-------	------	-----	-------

Los recorridos deben comprobar si la componente actual está utilizada viendo el valor correspondiente en el vector de `bool`

▷ Sin dejar huecos (se colocan todas al principio, por ejemplo):

- Si podemos elegir un valor especial  $N$  que represente componente no utilizada.

Ponemos  $N$  en la última componente utilizada (debemos reservar siempre una componente para  $N$ ).

$X$	$X$	$X$	$X$	$N$	?	...	?
-----	-----	-----	-----	-----	---	-----	---

- Si no podemos elegir dicho valor  $N$ , usamos una variable que marque el final:

util = 4							
$X$	$X$	$X$	$X$	?	?	...	?

---

**Nota:**

Una forma mixta de procesar los datos es trabajar con una representación con huecos y cada cierto tiempo, compactarlos para eliminar dichos huecos.

---

### III.1.3.3. El tipo string como secuencia de caracteres

El tipo de dato `string` es algo más complejo de lo que se ha visto hasta ahora (se verá que, realmente, es una clase con métodos asociados)

Una particularidad es que podemos acceder a las componentes del `string` con la notación corchete de los vectores. Cada componente es de tipo `char`:

No debemos modificar componentes que no estén ya incluidas en el `string`:

```
#include <iostream>
#include <string>
using namespace std;

int main(){
    string cadena;

    cadena = "FundamentAs de Programación";
    cout << cadena[0]; // Imprime F
    cadena[9] = 'o';    // Fundamentos de Programación
}
```

Aunque no entendamos por ahora la sintaxis, introducimos algunas operaciones útiles con datos de tipo `string`. Supongamos la siguiente declaración:

```
string cadena;  
char character;
```

▷ `cadena.size()` devuelve el tamaño actual de la cadena.

Por defecto, C++ inicializa un dato de tipo `string` a la **cadena vacía** (*empty string*) que es el literal `""`. El tamaño es cero.

▷ `cadena.push_back(character)` añade un carácter al final de la cadena.

▷ `+` es un operador binario que concatena dos cadenas.

```
#include <iostream>  
#include <string>  
using namespace std;
```

```
int main(){  
    string cadena;  
  
    cadena = "Fundamento";  
    cout << cadena.size();    // 10  
    cadena.push_back('s');  
    cout << cadena;           // Fundamentos  
    cout << cadena.size();    // 11  
    cadena = cadena + " de Programación";  
    cout << cadena;           // Fundamentos de Programación  
    cadena[50] = 'o';         // Comportamiento indeterminado
```



**No *añada* caracteres a un `string` mediante el acceso directo a la componente con la notación corchete. Use `push_back` en su lugar.**

## III.2. Recorridos sobre vectores

En lo que sigue, asumiremos un vector sin huecos y sin destacar ninguna componente como valor nulo.

util = 4

X	X	X	X	?	?	...	?
---	---	---	---	---	---	-----	---

Sin pérdida de generalidad, trabajaremos sobre un vector de `char`.

### III.2.1. Algoritmos de búsqueda

Los algoritmos de búsqueda tienen como finalidad localizar la posición de un elemento específico en un vector.

- ▷ **Búsqueda secuencial (Linear/Sequential search)** . Técnica más sencilla.
- ▷ **Búsqueda binaria (Binary search)** . Técnica más eficiente, aunque requiere que el vector esté ordenado.

En los siguientes algoritmos usaremos una variable `pos_encontrado`. Si el valor se encuentra, le asignaremos la posición en el vector en la que ha sido encontrado. En caso contrario, le asignaremos un valor imposible de posición, como por ejemplo -1.



### III.2.1.1. Búsqueda Secuencial

#### **Algoritmo: Primera Ocurrencia**

Ir recorriendo las componentes del vector

- mientras no se encuentre el elemento buscado Y
- mientras no lleguemos al final del mismo

```
const int TAMANIO = 50;
char vector[TAMANIO];
.....
i = 0;
pos_encontrado = -1;
encontrado = false;

while (i < total_utilizados && !encontrado){
    if (vector[i] == buscado){
        encontrado = true;
        pos_encontrado = i;
    }
    else
        i++;
}

if (encontrado)
    cout << "\nEncontrado en la posición " << pos_encontrado;
else
    cout << "\nNo encontrado";
```

[http://decsai.ugr.es/jccubero/FP/III\\_vector\\_busqueda\\_secuencial.cpp](http://decsai.ugr.es/jccubero/FP/III_vector_busqueda_secuencial.cpp)



### **Verificación (pruebas de unidad):**

- ▷ 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.

### **III.2.1.2. Búsqueda Binaria**

Se aplica sobre un vector ordenado.

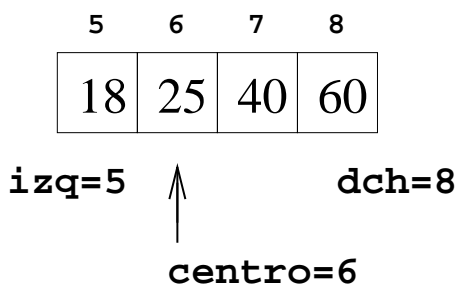
#### **Algoritmo: Búsqueda Binaria**

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.

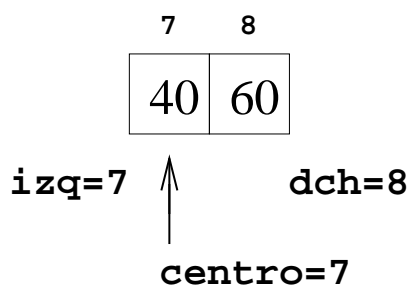
0	1	2	3	4	5	6	7	8
-8	4	5	9	12	18	25	40	60

**izq=0**
**centro=4**
**dch=8**

40 > 12



$$40 > 25$$



Encontrado

```
int main(){
    const int TAMANIO = 50;
    char vector[TAMANIO];
    .....
    izda = 0;
    dcha = total_utilizados - 1;
    encontrado = false;

    while (izda <= dcha && !encontrado){
        centro = (izda + dcha) / 2;

        if (vector[centro] == buscado)
            encontrado = true;
        else if (buscado < vector[centro])
            dcha = centro - 1;
        else
            izda = centro + 1;
    }

    if (encontrado)
        pos_encontrado = centro;
    else
        pos_encontrado = -1;
    .....
}
```

[http://decsai.ugr.es/jccubero/FP/III\\_vector\\_busqueda\\_binaria.cpp](http://decsai.ugr.es/jccubero/FP/III_vector_busqueda_binaria.cpp)

---

**Nota:**

En el caso de que el valor a buscar estuviese repetido, no se garantiza que se encuentre la primera posición.

---

### III.2.1.3. Otras búsquedas

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

$v = (h, b, c, f, d, ?, ?, ?)$

#### **Algoritmo:** Encontrar el mínimo

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

```
if (total_utilizados > 0){
    minimo = vector[0];
    posicion_minimo = 0;

    for (int i = 1; i < total_utilizados ; i++){
        if (vector[i] < minimo){
            minimo = vector[i];
            posicion_minimo = i;
        }
    }
}
else
    posicion_minimo = -1;
```

---

#### **Nota:**

También podríamos usar `vector[posicion_minimo]`: en vez de `minimo`

---

[http://decsai.ugr.es/jccubero/FP/III\\_vector\\_minimo.cpp](http://decsai.ugr.es/jccubero/FP/III_vector_minimo.cpp)

**Ejemplo.** Encuentre un vector dentro de otro.

```
principal = (h,b,a,a,a,b,f,d,?,?,?)  
a_buscar = (a,a,b,?,?,?, ?, ?, ?, ?)  
pos_encontrado = 3
```

**Algoritmo: Buscar un vector dentro de otro**

Si `a_buscar` cabe en `principal`

Recorrer `principal` -inicio- hasta que:

- `a_buscar` no quepa en lo que queda de `principal`
- se haya encontrado `a_buscar`

Recorrer las componentes de `a_buscar` -i- comparando  
`a_buscar[i]` con `principal[inicio + i]` hasta:

- llegar al final de `a_buscar` (se ha encontrado)
- encontrar dos caracteres distintos

```
for (int inicio = 0;
    inicio + total_utilizados_a_buscar <= total_utilizados
    &&
    !encontrado;
    inicio++){

    va_coincidiendo = true;

    for (int i = 0; i < total_utilizados_a_buscar && va_coincidiendo; i++)
        va_coincidiendo = principal[inicio + i] == a_buscar[i];

    if (va_coincidiendo){
        encontrado = true;
        pos_encontrado = inicio;
    }
}
```

[http://decsai.ugr.es/jccubero/FP/III\\_vector\\_busqueda\\_subvector.cpp](http://decsai.ugr.es/jccubero/FP/III_vector_busqueda_subvector.cpp)

## III.2.2. Recorridos que modifican componentes

### III.2.2.1. Inserción de un valor

El objetivo es insertar una componente nueva dentro del vector. Debemos desplazar una posición todas las componentes que hay a su derecha.

```
v = (t,e,c,a,?,?,?)    total_utilizados = 4
pos_insercion = 2      valor_nuevo = r
v = (t,e,r,c,a,?,?)    total_utilizados = 5
```

#### **Algoritmo: Insertar un valor**

```
Recorrer las componentes desde el final
hasta llegar a pos_insercion
    Asignarle a cada componente la anterior.
Colocar la nueva
```

```
for (int i = total_utilizados ; i > pos_insercion ; i--)
    vector[i] = vector[i-1];

vector[pos_insercion] = valor_nuevo;
total_utilizados++;
```

#### **Verificación (pruebas de unidad):**

- ▷ 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 número de componentes utilizadas sea igual al tamaño





### III.2.2.2. Eliminación de un valor

¿Qué significa eliminar/borrar una componente? Recuerde que la memoria reservada para todo el vector siempre es la misma.

Tipos de borrado:

▷ Borrado lógico

La componente se **marca** como borrada y habrá que tenerlo en cuenta en el procesamiento posterior.

Es el tipo de borrado usual cuando la ocupación es del tipo:

N	X	X	N	N	X	...	N
---	---	---	---	---	---	-----	---

Obviamente, la ventaja es la eficiencia. De hecho, si se prevé trabajar con datos en los que hay que hacer continuas operaciones de borrado, esta representación con huecos puede ser la recomendada.

▷ Borrado físico

Todas las componentes que hay a la derecha se desplazan una posición a la izquierda.

Es el tipo de borrado usual cuando la ocupación es del tipo que nos ocupa:

util = 4							
X	X	X	X	?	?	...	?

El problema es la ineficiencia. Habrá que tener especial cuidado de no realizar este tipo de borrados de forma continua.

```
v = (t,e,r,c,a,?,?,?)    total_utilizados = 5
pos_a_eliminar = 2
v = (t,e,c,a,?,?,?,?)    total_utilizados = 4
```

**Algoritmo: Eliminar un valor**

Recorrer las componentes desde la posición a eliminar hasta el final

Asignarle a cada componente la siguiente.

Actualizar total\_utilizados

```
if (posicion >= 0 && posicion < total_utilizados){
    int tope = total_utilizados-1;

    for (int i = posicion ; i < tope ; i++)
        vector[i] = vector[i+1];

    total_utilizados--;
}
```

**Los casos de prueba para la verificación serían los mismos que los del algoritmo de inserción.**

### III.2.3. 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.

▷ **Aplicaciones:**

Mostrar los ficheros de un directorio ordenados alfabéticamente, ordenar los resultados de una búsqueda en Internet (PageRank es un método usado por Google que asigna un número de *importancia* a una página web), etc.

▷ **Técnicas de ordenación:**

- **Ordenación interna** : Todos los datos están en memoria principal durante el proceso de ordenación.
  - En la asignatura veremos métodos básicos como Inserción, Selección e Intercambio.
  - En el segundo cuatrimestre se verán métodos más avanzados como Quicksort o Mergesort.
- **Ordenación externa** : Parte de los datos a ordenar están en memoria externa mientras que otra parte está en memoria principal siendo ordenada.

▷ **Aproximaciones:**

Supondremos que los datos a ordenar están en un vector.

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

Vamos a ver varios algoritmos de ordenación.

*Idea común a algunos algoritmos de ordenación:*

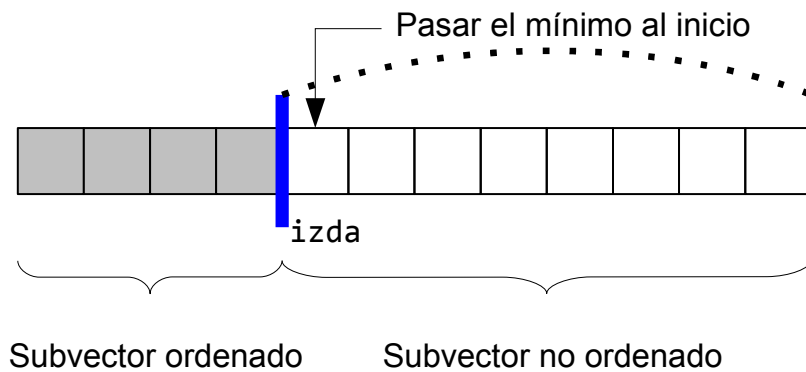
- ▷ El vector se dividirá en dos sub-vectores. El de la izquierda, contendrá componentes ordenadas. Las del sub-vector derecho no están ordenadas.
- ▷ Se irán cogiendo componentes del sub-vector derecho y se colocarán adecuadamente en el sub-vector izquierdo.

**Es muy importante conocer los métodos de ordenación  
para el examen de FP**

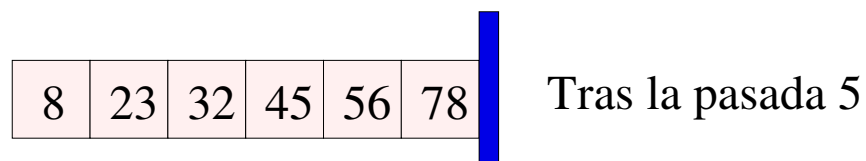
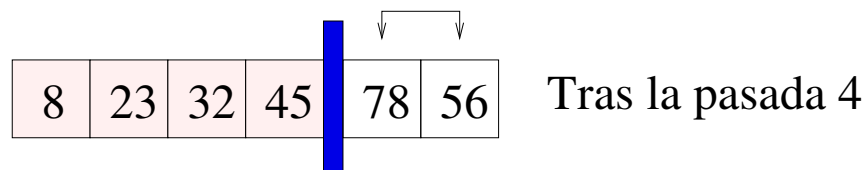
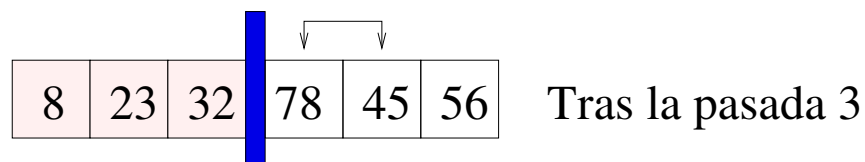
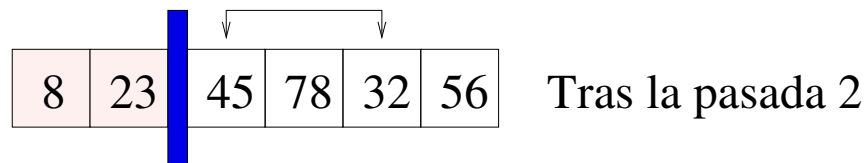
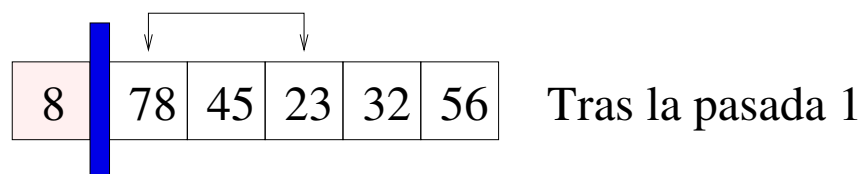
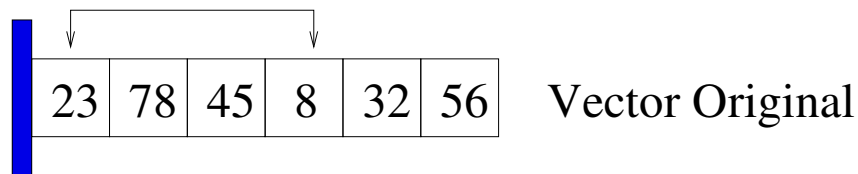
**IMPORTANT**

### III.2.3.1. Ordenación por Selección

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



Para ilustrar los métodos usaremos una secuencia de enteros (en cualquier caso, recuerde que una secuencia de `char` no es sino una secuencia de los enteros que representan los órdenes correspondientes)



### **Algoritmo: Ordenación por Selección**

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

```
for (int izda = 0 ; izda < total_utilizados ; izda++){
    minimo = vector[izda];
    posicion_minimo = izda;

    for (int i = izda + 1; i < total_utilizados ; i++){
        if (vector[i] < minimo){
            minimo = vector[i];
            posicion_minimo = i;
        }
    }

    intercambia = vector[izda];
    vector[izda] = vector[posicion_minimo];
    vector[posicion_minimo] = intercambia;
}
```

---

#### **Nota:**

La última iteración (cuando `izda` es igual a `total_utilizados - 1`) no es necesaria. El bucle podría llegar hasta `total_utilizados - 2` (inclusive)

---

[http://decsai.ugr.es/jccubero/FP/III\\_vector\\_ordenacion.cpp](http://decsai.ugr.es/jccubero/FP/III_vector_ordenacion.cpp)



### ***Verificación (pruebas de unidad):***

- ▷ **Que el vector esté vacío**
- ▷ **Que el vector sólo tenga una componente**
- ▷ **Que tenga un número de componentes par/impar**
- ▷ **Que el vector ya estuviese ordenado**
- ▷ **Que el vector ya estuviese ordenado, pero de mayor a menor**
- ▷ **Que el vector tenga todas las componentes iguales**
- ▷ **Que tenga dos componentes iguales al principio o al final o en medio**

***Nota.*** Estas pruebas también son aplicables al resto de algoritmos de ordenación

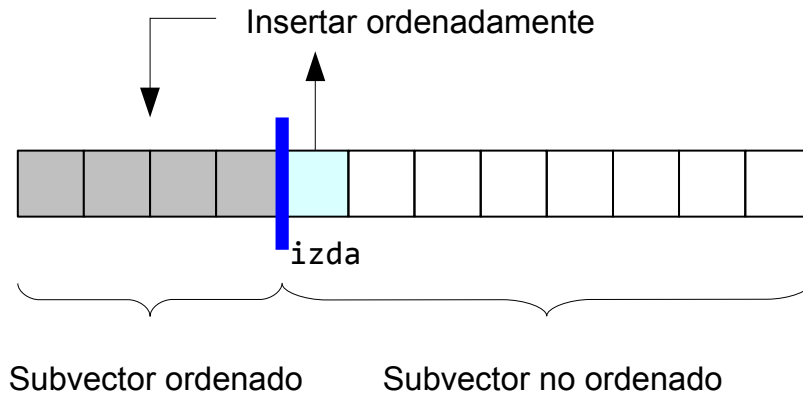
***Applet de demostración del funcionamiento:***

<http://www.sorting-algorithms.com/>

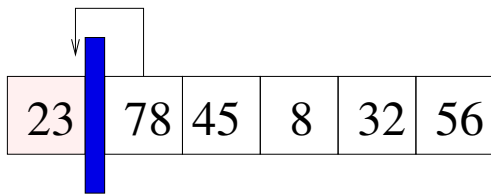


### III.2.3.2. Ordenación por Inserción

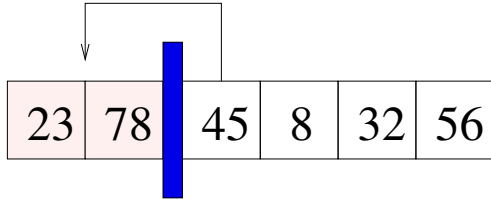
**Ordenación por inserción (*Insertion sort*)** : El vector se divide en dos sub-vectores: el de la izquierda ordenado, y el de la derecha desordenado. Cogemos el primer elemento del subvector desordenado y lo insertamos de forma ordenada en el subvector de la izquierda (el ordenado).



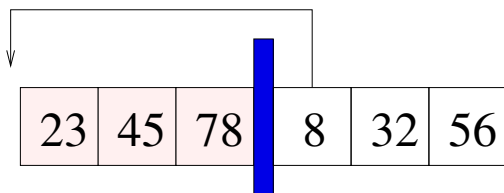
**Nota.** La componente de la posición *izda* (primer elemento del subvector desordenado) será reemplazada por la anterior (después de desplazar)



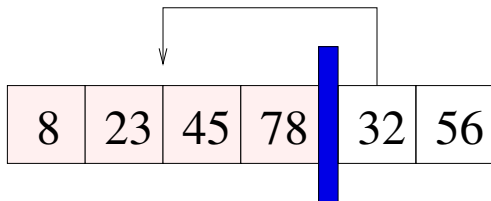
Vector Original



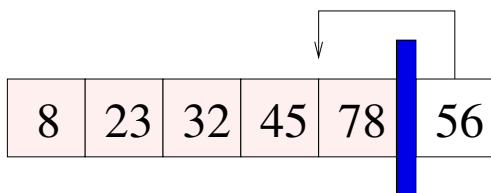
Tras la pasada 1



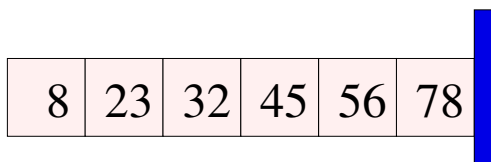
Tras la pasada 2



Tras la pasada 3



Tras la pasada 4



Tras la pasada 5

### **Algoritmo: Ordenación por Inserción**

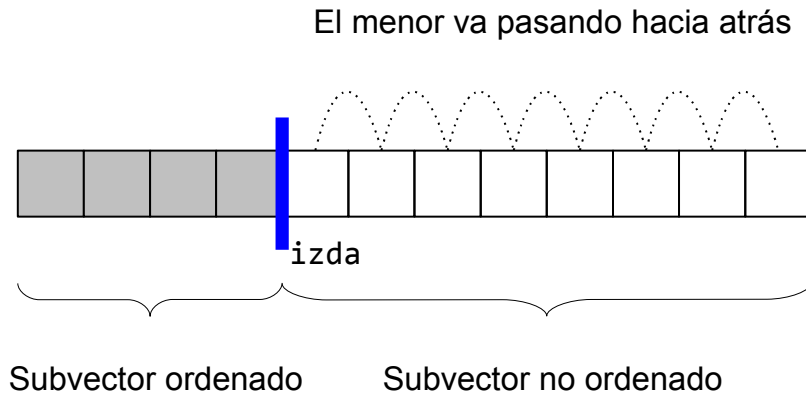
Ir fijando el inicio del subvector derecho  
con un índice izda (desde 1 hasta total\_utilizados - 1)  
Insertar v[izda] de forma ordenada  
en el subvector izquierdo

```
for (int izda = 1; izda < total_utilizados; izda++){  
    a_desplazar = vector[izda];  
  
    for (i = izda; i > 0 && a_desplazar < vector[i-1]; i--)  
        vector[i] = vector[i-1];  
  
    vector[i] = a_desplazar;  
}
```

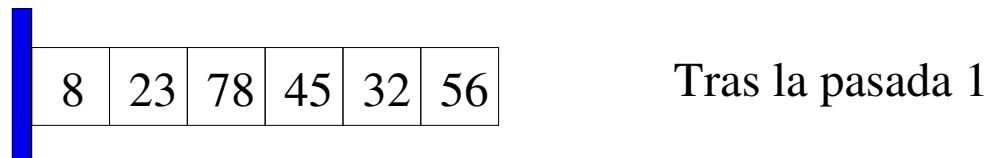
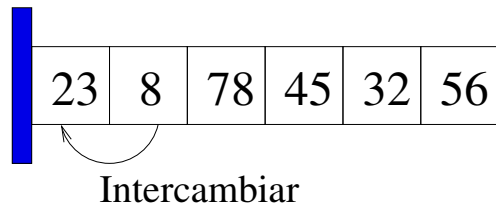
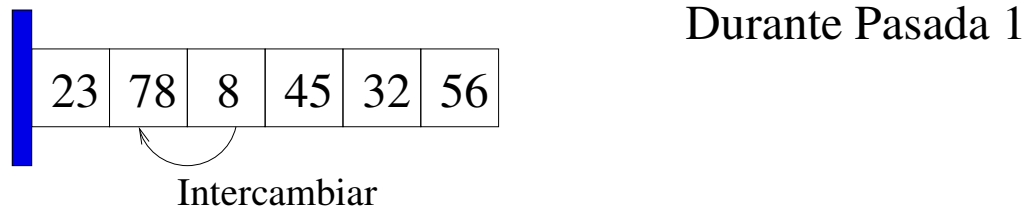
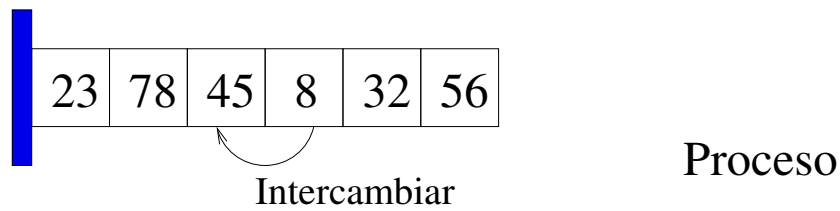
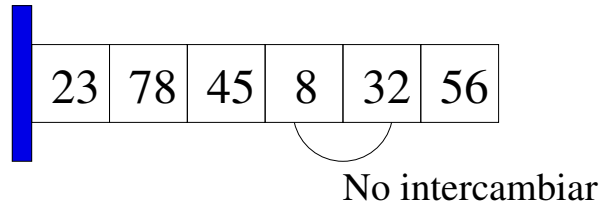
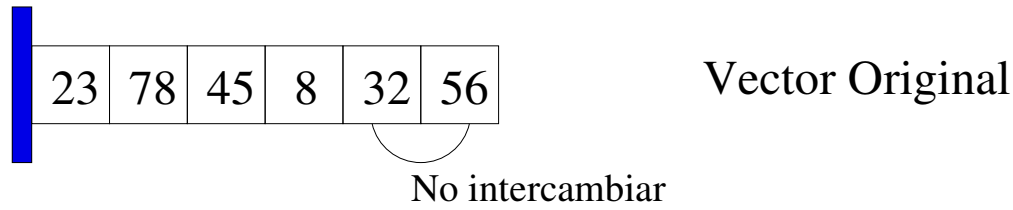
[http://decsai.ugr.es/jccubero/FP/III\\_vector\\_ordenacion.cpp](http://decsai.ugr.es/jccubero/FP/III_vector_ordenacion.cpp)

### III.2.3.3. Ordenación por Intercambio Directo (Método de la Burbuja)


**Ordenación por intercambio directo (burbuja) (Bubble sort)** : 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 (intercambiándolos)



## Primera Pasada




## Resto de Pasadas




23	78	45	8	32	56
----	----	----	---	----	----

 Vector Original


8	23	78	45	32	56
---	----	----	----	----	----

 Tras la pasada 1


8	23	32	78	45	56
---	----	----	----	----	----

 Tras la pasada 2


8	23	32	45	78	56
---	----	----	----	----	----

 Tras la pasada 3

8	23	32	45	56	78
---	----	----	----	----	----

 Tras la pasada 4

8	23	32	45	56	78
---	----	----	----	----	----

 Tras la pasada 5

### **Algoritmo: Ordenación por Burbuja**

```
Ir fijando el inicio del subvector derecho
con un contador izda desde 0 hasta total_utilizados - 1
  Recorrer el subvector de la derecha desde
    el final hasta el principio (izda)
    con un contador i

    Si  $v[i] < v[i-1]$  intercambiarlos
```

#### ► **Primera Aproximación**

```
for (int izda = 0; izda < total_utilizados; izda++){
    for (int i = total_utilizados-1 ; i > izda ; i--){
        if (vector[i] < vector[i-1]){
            intercambia = vector[i];
            vector[i] = vector[i-1];
            vector[i-1] = intercambia;
        }
    }
}
```

[http://decsai.ugr.es/jccubero/FP/III\\_vector\\_ordenacion.cpp](http://decsai.ugr.es/jccubero/FP/III_vector_ordenacion.cpp)

**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

```
cambio = true;

for (int izda = 0; izda < total_utilizados && cambio; izda++){
    cambio = false;

    for (int i = total_utilizados-1 ; i > izda ; i--){
        if (vector[i] < vector[i-1]){
            cambio = true;
            intercambia = vector[i];
            vector[i] = vector[i-1];
            vector[i-1] = intercambia;
        }
    }
}
```

[http://decsai.ugr.es/jccubero/FP/III\\_vector\\_ordenacion.cpp](http://decsai.ugr.es/jccubero/FP/III_vector_ordenacion.cpp)



## III.3. Matrices

### III.3.1. Declaración y operaciones con matrices

#### III.3.1.1. Declaración

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

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.

**Una matriz se declara de la forma siguiente:**

`<tipo> <identificador> [<núm. filas>][<núm. columnas>;`

**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.**

```
int main(){
    const int TAMANIO_FIL = 2;
    const int TAMANIO_COL = 3;

    double parcela[TAMANIO_FIL][TAMANIO_COL];
    .....
```

---

**Otros ejemplos:**

```
.....
int    imagen[MAX_ALTURA][MAX_ANCHURA];
double notas[NUM_ALUMNOS][NUM_NOTAS];
char   crucigrama [MAX_FIL][MAX_COL];
string ventas_diarias[MAX_VENTAS_POR_DIA][MAX_NUM_CODIGOS];
```

### III.3.1.2. Acceso y asignación

`<identificador> [<índice fila>][<índice columna>;`

`<identificador> [<índice fila>][<índice columna>]` 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:

`<identificador> [<índice fila>][<índice columna>] = <expresión>`

`<expresión>` ha de ser del mismo tipo de dato que el tipo base de la matriz.

#### *Ejemplo.*

```
int main(){
    const int TAMANIO_FIL = 2;
    const int TAMANIO_COL = 3;
    double parcela[TAMANIO_FIL][TAMANIO_COL];

    parcela[0][1] = 4.5; // Correcto;
    parcela[2][0] = 7.2; // Error ejecución: Fila 2 no existe.
    parcela[0][3] = 7.2; // Error ejecución: Columna 3 no existe.
    parcela[0][0] = 4;    // Correcto. Casting automático: double = int
```

### III.3.1.3. 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
```

### III.3.1.4. Representación en memoria (Ampliación)

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



m

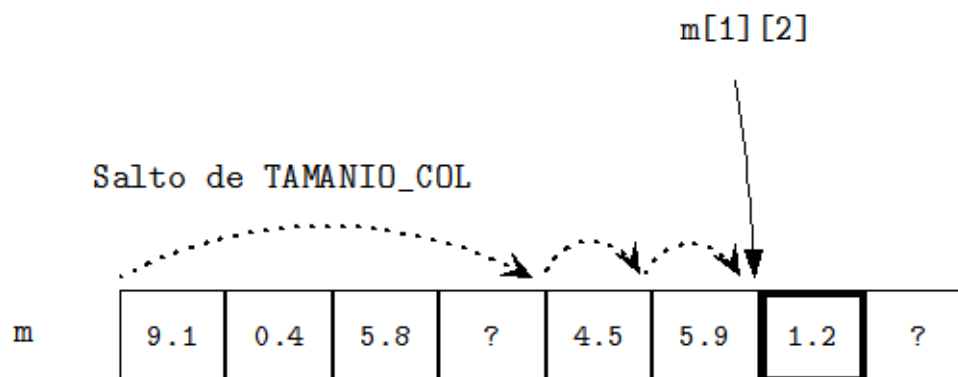
9.1	0.4	5.8	?
4.5	5.9	1.2	?

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 `TAMANIO_COL`, 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 `TAMANIO_COL`, pero no `TAMANIO_FIL`. Para ello, da tantos saltos como indique la expresión  $i * \text{TAMANIO\_COL} + j$

### III.3.1.5. Más de dos dimensiones

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

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

```
const int DIV_HOR = 2,  
        DIV_VERT = 3,  
        TOTAL_CULTIVOS = 5;  
  
double parcela[DIV_HOR][DIV_VERT][TOTAL_CULTIVOS];  
  
// Asignación al primer cultivo de la parcela 1,2  
  
parcela[1][2][0] = 4.5;
```

El tamaño (número de componentes) de una matriz es el producto de los tamaños de cada dimensión. En el ejemplo anterior, se han reservado en memoria un total de  $2 \times 3 \times 5 = 30$  enteros.

## III.3.2. Gestión de componentes útiles con matrices

¿Qué componentes usamos en una matriz? Depende del problema.

En este apartado trabajamos directamente en el `main`. En el próximo tema lo haremos dentro de una clase.

Veamos los tipos de gestión de componentes no utilizadas más usados.

### III.3.2.1. Se usan todas las componentes

Ocupamos todas las componentes reservadas. Las casillas no utilizadas tendrán un valor especial *N* del tipo de dato de la matriz.

`MAX_FIL = 15, MAX_COL = 20`

<i>N</i>	<i>X</i>	<i>N</i>	<i>N</i>	<i>X</i>	<i>N</i>	...	<i>X</i>
<i>X</i>	<i>X</i>	<i>N</i>	<i>N</i>	<i>N</i>	<i>X</i>	...	<i>N</i>
<i>N</i>	<i>N</i>	<i>X</i>	<i>X</i>	<i>X</i>	<i>N</i>	...	<i>N</i>
...	...	...	...	...	...	...	...
<i>X</i>	<i>N</i>	<i>X</i>	<i>N</i>	<i>X</i>	<i>N</i>	...	<i>X</i>

**Ejemplo.** Queremos gestionar un crucigrama que siempre tiene un tamaño fijo 5 x 6 . El carácter # representa un separador entre palabras.

```
A C E R O S
L A S E R #
T I O # D E
A D # N E D
R A Z O N #
```

**Leemos los datos e imprimimos el crucigrama.**

```
#include <iostream>
using namespace std;

int main(){
    const int MAX_FIL = 5, MAX_COL = 6;
    char crucigrama [MAX_FIL][MAX_COL];

    for (int i = 0; i < MAX_FIL; i++)
        for (int j = 0; j < MAX_COL; j++)
            cin >> crucigrama[i][j];

    for (int i = 0; i < MAX_FIL; i++){
        for (int j = 0; j < MAX_COL; j++)
            cout << crucigrama[i][j] << " ";
        cout << "\n";
    }
}
```

**Los datos deben introducirse con todos los caracteres consecutivos:**

```
ACEROSLASER#TIO#DEAD#NEDRAZON#
```

[http://decsai.ugr.es/jccubero/FP/III\\_crucigrama\\_base.cpp](http://decsai.ugr.es/jccubero/FP/III_crucigrama_base.cpp)



### III.3.2.2. Se ocupan todas las columnas pero no todas las filas (o al revés)

Debemos estimar el máximo número de filas que podamos manejar (`MAX_FIL`) para reservar memoria suficiente. Usaremos una variable `filas_utilizadas` para saber cuántas se usan en cada momento de la ejecución del programa.

```
MAX_FIL = 15, MAX_COL = 20
filas_utilizadas = 3
```

X	X	X	X	X	X	...	X
X	X	X	X	X	X	...	X
X	X	X	X	X	X	...	X
?	?	?	?	?	?	...	?
...	...	...	...	...	...	...	...
?	?	?	?	?	?	...	?

**Ejemplo.** Queremos gestionar las notas de los alumnos de una clase. Cada alumno tiene 3 notas de evaluación continua, dos notas de exámenes de prácticas y una nota del examen escrito.

```
MAX_ALUMNOS = 100, NUM_NOTAS = 6
num_alumnos = 2
```

4.0	7.5	8.0	6.5	7.0	4.0
3.0	2.5	4.0	3.0	2.5	1.5
?	?	?	?	?	?
...	...	...	...	...	...
?	?	?	?	?	?

**Calculamos la nota media de cada alumno, aplicando unas ponderaciones.**

```
int main(){
    const int MAX_ALUMNOS = 100, NUM_NOTAS = 6;
    double notas[MAX_ALUMNOS][NUM_NOTAS];
    int num_alumnos;
    double nota_final;
    double ponderacion[NUM_NOTAS] = {0.1/3.0, 0.1/3.0, 0.1/3.0,
                                      0.05, 0.15, 0.7};

    cout.precision(3);

    do{
        cin >> num_alumnos;
    }while (num_alumnos < 0);

    for (int i = 0; i < num_alumnos; i++)
        for (int j = 0; j < NUM_NOTAS; j++)
            cin >> notas[i][j];

    for (int i = 0; i < num_alumnos; i++){
        nota_final = 0;

        for (int j = 0; j < NUM_NOTAS; j++)
            nota_final = nota_final + notas[i][j] * ponderacion[j];

        cout << "\nNota final del alumno número " << i
              << " = " << nota_final;
    }
}
```

[http://decsai.ugr.es/jccubero/FP/III\\_notas\\_base.cpp](http://decsai.ugr.es/jccubero/FP/III_notas_base.cpp)

### III.3.2.3. Se ocupa un bloque rectangular

Este tipo de ocupación es bastante usual.

Ocupamos un bloque completo. Lo normal será la esquina superior izquierda.

Por cada dimensión debemos estimar el máximo número de componentes que podamos almacenar y usaremos tantas variables como dimensiones haya para saber cuántas componentes se usan en cada momento.

```
MAX_FIL = 15, MAX_COL = 20
util_fil = 2, util_col = 4
```

X	X	X	X	?	?	...	?
X	X	X	X	?	?	...	?
?	?	?	?	?	?	...	?
...	...	...	...	...	...	...	...
?	?	?	?	?	?	...	?

**Ejemplo.** Queremos gestionar un crucigrama de cualquier tamaño.

```
A  C  E  R  O  S  ?  ...  ?
L  A  S  E  R  #  ?  ...  ?
T  I  O  #  D  E  ?  ...  ?
A  D  #  N  E  D  ?  ...  ?
R  A  Z  O  N  #  ?  ...  ?
?  ?  ?  ?  ?  ?  ?  ...  ?
...
?  ?  ?  ?  ?  ?  ?  ...  ?
```

## Leemos los datos del crucigrama y los imprimimos.

```
#include <iostream>
using namespace std;

int main(){
    const char SEPARADOR = '#';
    const int MAX_FIL = 5, MAX_COL = 6;
    char crucigrama [MAX_FIL][MAX_COL];
    int util_fil, util_col;

    do{
        cin >> util_fil;
    }while (util_fil > MAX_FIL || util_fil < 0);

    do{
        cin >> util_col;
    }while (util_col > MAX_COL || util_col < 0);

    for (int i = 0; i < util_fil; i++)
        for (int j = 0; j < util_col; j++)
            cin >> crucigrama[i][j];

    for (int i = 0; i < util_fil; i++){
        for (int j = 0; j < util_col; j++)
            cout << crucigrama[i][j] << " ";
        cout << "\n";
    }
}
```

Una palabra viene delimitada por el inicio de la fila, el final de ésta y también por el terminador #. Sólo consideramos palabras en horizontal y de izquierda a derecha.

Vamos a construir un vector con todas las palabras del crucigrama. Cada palabra se almacenará en un `string`.

Lo vamos a ver de dos formas distintas.

### ► *Primera forma*

Utilizamos dos apuntadores `izda` y `dcha` para delimitar el inicio y final de la palabra. `izda` se posicionará al inicio de la palabra y `dcha` irá avanzando hasta el final de la misma. Al llegar al final, construiremos la cadena con los caracteres que haya entre `izda` y `dcha`.

#### **Algoritmo:** Obtener palabras en horizontal de un crucigrama

```
Recorrer todas las filas
  Inicializar izda y dcha a cero

  Recorrer la fila actual
    Si el carácter que hay en la columna izda es un
      separador, avanzar izda y dcha
    si no
      Si el siguiente a dcha es separador o final de fila,
        estamos al final de una palabra
        => Construir la palabra con los caracteres
          entre izda y dcha.
          Avanzar izda y dcha al final de la palabra
      si no
        Mantener izda y avanzar dcha
```

```
string cjto_palabras[MAX_FIL*MAX_COL];
.....
util_cjto_palabras = 0;

for (int fil = 0; fil < util_fil; fil++){
    izda = dcha = 0;
    hay_columnas = util_col != 0;

    while (hay_columnas){
        if (crucigrama[fil][izda] == SEPARADOR){
            izda++;
            dcha++;
        }
        else{
            siguiente = dcha + 1;

            if (siguiente == util_col ||
                crucigrama[fil][siguiente] == SEPARADOR){

                for (int j = izda; j <= dcha; j++)
                    palabra.push_back(crucigrama[fil][j]);

                cjto_palabras[util_cjto_palabras] = palabra;
                util_cjto_palabras++;
                palabra = "";
                izda = dcha = siguiente;
            }
            else
                dcha++;
        }
        hay_columnas = dcha < util_col;
    }
}
```

## ► Segunda forma

Utilizamos un único apuntador y detectamos las distintas situaciones que se pueden producir.

```
a indica que hay una letra distinta de separador
# indica que hay un separador
... indica que hay cualquier cosa (separador o letra)
! indica la posición actual
| indica el final de la fila
```

### Casos a tener en cuenta:

Casos en los que debo ir construyendo la palabra  
pero no la he terminado: Cuando el actual no es separador  
y no he llegado al final

```
!
...a...|
```

Casos en los que he terminado una palabra y la añado:  
Cuando el actual es un separador o final de fila y  
en cualquier caso, el anterior es una letra

```
!
...a#...|
!
...a|
```

En cualquier otro caso, no tengo que hacer nada más.

Independientemente del caso, avanzo la columna.

### **Algoritmo: Obtener palabras en horizontal de un crucigrama (2)**

Usaremos dos variables:

    actual\_es\_separador

    anterior\_es\_separador (el anterior al actual)

Recorrer todas las filas

    Recorrer la fila actual

        Si no hemos rebasado el final y el actual no es separador

            Concatenar el carácter actual en la palabra  
            que se está formando.

        si no

            Si el actual es un separador o final de fila y  
            en cualquier caso, el anterior es una letra

                Añadir la palabra que se estaba formando



```
util_cjto_palabras = 0;
anterior_es_separador = false;
actual_es_separador = false;

for (int fil = 0; fil < util_fil; fil++){
    col = 0;
    anterior_es_separador = true;

    do{
        hay_columnas = col < util_col;

        if (hay_columnas)
            actual_es_separador = crucigrama[fil][col] == SEPARADOR;

        if (hay_columnas && ! actual_es_separador){
            palabra = palabra + crucigrama[fil][col];
            anterior_es_separador = false;
        }
        else{
            if (! anterior_es_separador &&
                (!hay_columnas || actual_es_separador)){

                cjto_palabras[util_cjto_palabras] = palabra;
                util_cjto_palabras++;
                palabra = "";
                anterior_es_separador = true;
            }
        }
        col++;
    }while (hay_columnas);
}
```

[http://decsai.ugr.es/jccubero/FP/III\\_crucigrama\\_extrae\\_palabras.cpp](http://decsai.ugr.es/jccubero/FP/III_crucigrama_extrae_palabras.cpp)

**Ejemplo.** Queremos gestionar una sopa de letras de cualquier tamaño (no hay carácter # separador de palabras como en un crucigrama).

```
A C E R O S ? ... ?
L A S E R I ? ... ?
T I O B D E ? ... ?
A D I N E D ? ... ?
R A Z O N P ? ... ?
? ? ? ? ? ? ? ... ?
...
? ? ? ? ? ? ? ... ?
```

Supongamos que queremos buscar una palabra (en horizontal y de izquierda a derecha). Usamos como base el algoritmo de búsqueda de un vector dentro de otro visto en la página 319

**Algoritmo: Buscar una palabra en una sopa de letras**

Recorrer todas las filas -fila- hasta terminarlas  
o hasta encontrar la palabra

Con una fila fija, recorrer sus columnas -col\_inicio-  
hasta que:

- a\_buscar no quepa en lo que queda de fila
- se haya encontrado a\_buscar

Recorrer las componentes de a\_buscar -i- comparando  
a\_buscar[i] con sopa[fila][col\_inicio + i] hasta:

- llegar al final de a\_buscar (se ha encontrado)
- encontrar dos caracteres distintos

```
#include <iostream>
using namespace std;

int main(){
    const int MAX_FIL = 30, MAX_COL = 40;
    char sopa [MAX_FIL][MAX_COL];
    int util_fil, util_col;

    char a_buscar [MAX_COL];
    int  tamaño_a_buscar;
    bool encontrado, va_coincidiendo;
    int  fil_encontrado, col_encontrado;

    do{
        cin >> util_fil;
    }while (util_fil > MAX_FIL || util_fil < 0);

    do{
        cin >> util_col;
    }while (util_col > MAX_COL || util_col < 0);

    for (int i = 0; i < util_fil; i++)
        for (int j = 0; j < util_col; j++)
            cin >> sopa[i][j];

    cin >> tamaño_a_buscar;

    for (int i = 0; i < tamaño_a_buscar; i++)
        cin >> a_buscar[i];

    encontrado = false;
    fil_encontrado = col_encontrado = -1;
```

```
for (int fil = 0; fil < util_fil && !encontrado; fil++){

    for (int col_inicio = 0;
        col_inicio + tamaño_a_buscar <= util_col && !encontrado;
        col_inicio++){

        va_coincidiendo = true;

        for (int i = 0; i < tamaño_a_buscar && va_coincidiendo; i++)
            va_coincidiendo = sopa[fil][col_inicio + i] == a_buscar[i];

        if (va_coincidiendo){
            encontrado = true;
            fil_encontrado = fil;
            col_encontrado = col_inicio;
        }
    }
}

if (encontrado)
    cout << "\nEncontrado en " << fil_encontrado << "," << col_encontrado;
else
    cout << "\nNo encontrado";
}
```

[http://decsai.ugr.es/jccubero/FP/III\\_sopa.cpp](http://decsai.ugr.es/jccubero/FP/III_sopa.cpp)

### III.3.2.4. Se ocupan las primeras filas, pero con tamaños distintos

Debemos usar un vector para almacenar el tamaño actual de cada fila.

MAX\_FIL = 15, MAX\_COL = 20 , util\_fil = 3, util\_col = {5, 2, 4}

X	X	X	X	X	?	...	?
X	X	?	?	?	?	...	?
X	X	X	X	?	?	...	?
?	?	?	?	?	?	...	?
...	...	...	...	...	...	...	...
?	?	?	?	?	?	...	?

**Ejemplo.** Almacenamos las notas de evaluación continua de los alumnos. Cada alumno se corresponde con una fila y el número de notas puede variar entre los distintos alumnos, pero no serán más de 8.

MAX\_ALUMNOS = 100, MAX\_NUM\_NOTAS = 8

num\_alumnos = 2 , num\_notas\_por\_alumno = {4, 3, ?, ..., ?}

9.5	8.3	9.1	6.7	?	?	...	?
3.4	4.1	2.3	?	?	?	...	?
?	?	?	?	?	?	...	?
...	...	...	...	...	...	...	...
?	?	?	?	?	?	...	?

```
int main(){
    const int MAX_ALUMNOS = 100, MAX_NUM_NOTAS = 8;
    double notas_ev_continua [MAX_ALUMNOS][MAX_NUM_NOTAS];
    int num_notas_por_alumno[MAX_ALUMNOS];          int num_alumnos;
```

Indiquemos como nota final que en matrices de gran tamaño, se hace necesario usar otras técnicas de almacenamiento de datos con *huecos*, ya que el número de datos *desperdiciados* puede ser demasiado elevado.

La solución pasará por usar *memoria dinámica (dynamic memory)* . Se verá en el segundo cuatrimestre,

---

**Bibliografía recomendada para este tema:**

**Una referencia bastante adecuada es Wikipedia (pero la versión en inglés)**

[http://en.wikipedia.org/wiki/Category:Search\\_algorithms](http://en.wikipedia.org/wiki/Category:Search_algorithms)

[http://en.wikipedia.org/wiki/Category:Sorting\\_algorithms](http://en.wikipedia.org/wiki/Category:Sorting_algorithms)