



Programación

UD5: Vectores y Matrices. Algoritmos de Ordenación.





Parte I: Vectores

- ① Definición
- ② Declaración y representación en memoria
- ③ Operaciones con vectores
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- ④ Sobre el tamaño de los vectores
- ⑤ Modularización y vectores
 - Construcción de vectores en funciones
 - Trabajando con vectores locales
- ⑥ Cadenas de caracteres



Parte II: Algoritmos de búsqueda y ordenación

7 Algoritmos de búsqueda

- Búsqueda secuencial
- Búsqueda binaria

8 Algoritmos de ordenación

- Ordenación por selección
- Ordenación por inserción
- Ordenación por intercambio directo (Método de la burbuja)



Parte III: Matrices

- 9 Declaración e inicialización de matrices de 2 dimensiones
- 10 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
- 11 Sobre el tamaño de las matrices
- 12 Matrices de más de 2 dimensiones
- 13 Modularización con matrices
- 14 Gestión de filas de una matriz como vectores



Motivación

Motivación

Pensar en un programa para calcular la varianza de 500 datos introducidos por el usuario.

Problema

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



Parte I

Vectores



Definición

Definición

Tipo de dato compuesto

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.

Vector

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**.



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.
 - El número de componentes debe conocerse a priori y no es posible alterarlo durante la ejecución del programa.
 - Pueden usarse literales o constantes enteras pero **nunca una variable**¹.
- Las posiciones ocupadas por el vector están contiguas en memoria.

double notas[3] ;

notas =

?	?	?
---	---	---

¹El estándar C99 permite usar una variable pero **C++ estándar no lo admite**.
g++ lo admite como extensión propia.



Declaración y representación en memoria

Consejo

Usar constantes para especificar el tamaño de los vectores.

```
1 int main(){
2     const int NUM_REACTORES=20;
3     const int TOTAL_ALUMNOS = 100;
4     int longitud=50;
5
6     double notas[TOTAL_ALUMNOS];
7     int TemperaturasReactores[NUM_REACTORES];
8     // Otros ejemplos
9     bool casados[40];
10    char NIF[9];
11    bool Vector[longitud]; // Error en compilación.
12    .....
```



Acceso a las componentes

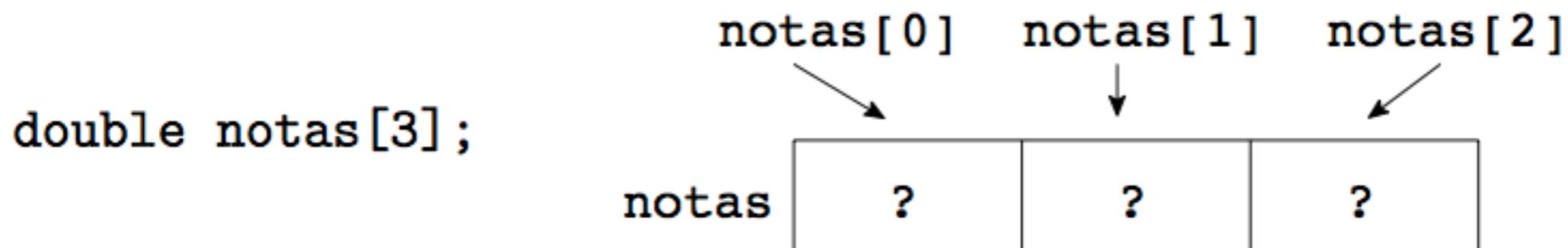
Dada la declaración

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

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 <N.Componentes>-1.



notas[9], notas['1'] o notas[1.5] no son correctas.

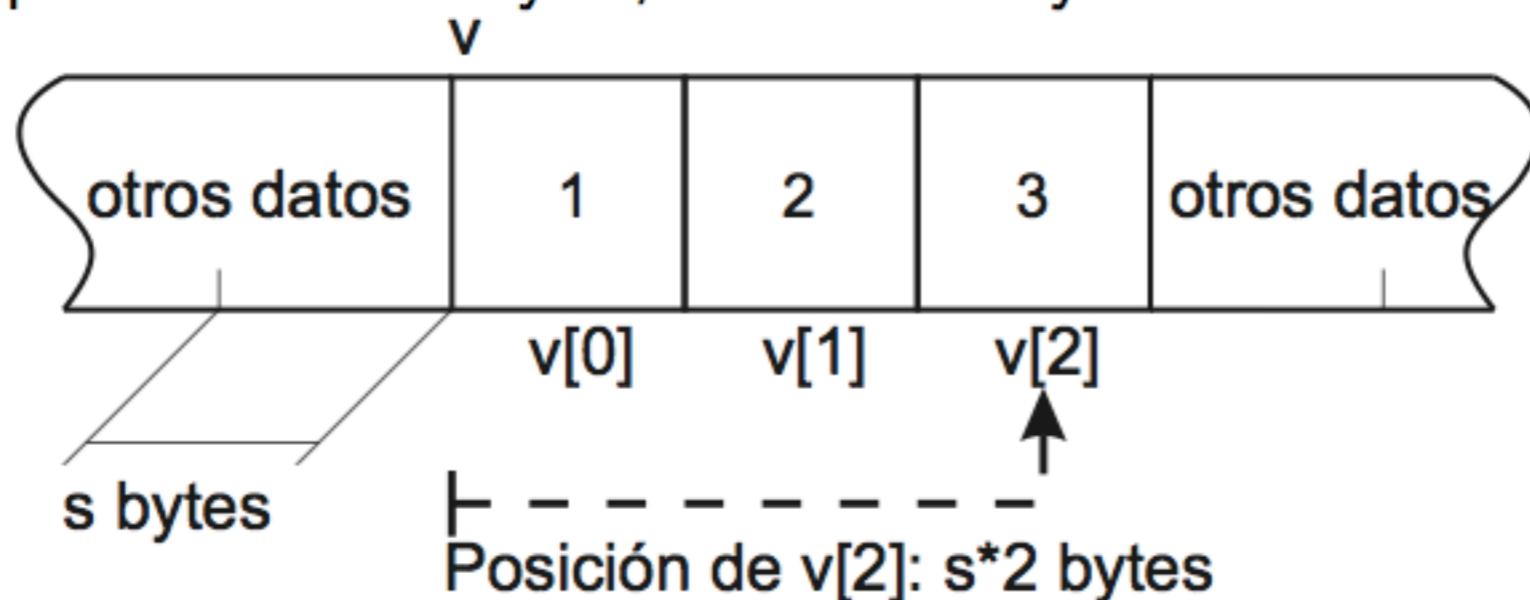
- Cada componente es una variable más del programa, del tipo indicado en la declaración del vector.



Acceso a las componentes

- Para acceder a la componente i , el compilador se debe desplazar i posiciones desde el comienzo del vector.
- Cada posición tiene un número de bytes determinado por el tipo de dato base.

Para acceder a $v[2]$ el compilador saltará 2 posiciones desde el comienzo de v . Si cada posición tiene s bytes, saltará $s*2$ bytes.





Inicialización

<tipo> <identificador> [<N.Componentes>] = {<valores separados por coma>};

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

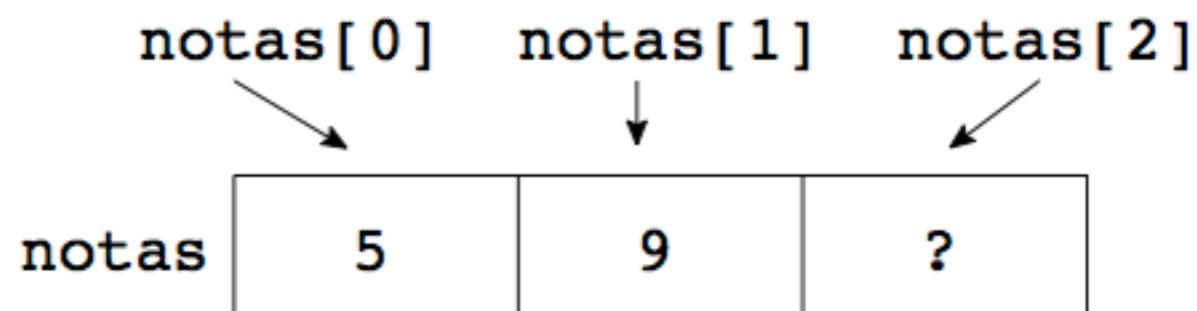
automáticamente el compilador asume `int vector[3]`



Asignación

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

```
1 int main(){  
2     double notas[3];  
3  
4     notas[0] = 5;  
5     notas[1] = 9;
```



¡Cuidado!

No se permite la asignación global a todos los elementos del vector.

¡Cuidado!

El compilador no comprueba el acceso a las componentes: modificar una componente inexistente tiene consecuencias imprevisibles.



Lectura y escritura

La lectura y escritura se realiza componente a componente.

```
1 int main(){
2     const int NUM_NOTAS = 5;
3     double notas[NUM_NOTAS], media;
4
5     for (int i=0; i<NUM_NOTAS; i++){
6         cout << "Nota del alumno " << i << ": ";
7         cin >> notas[i];
8     }
9
10    media = 0;
11    for (int i=0; i<NUM_NOTAS; i++)
12        media += notas[i];
13    media /= NUM_NOTAS;
14
15    cout << "\nMedia = " << media << endl;
16 }
```





Sobre el tamaño de los vectores I

En C++ no es posible declarar un vector de tamaño variable. →

Debemos dar un tamaño *suficientemente grande* a los vectores.

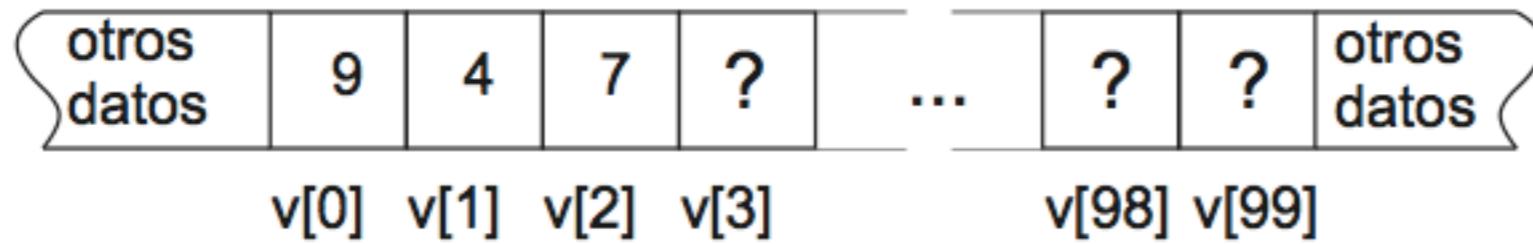
Pero, ¿Cómo sabemos el número de elementos que realmente estamos usando en un momento dado?

Normalmente se dejan libres los elementos con índice más alto.

Habitualmente se usa una variable entera que indique el número de componentes usadas.

Como convención, usaremos identificadores del tipo `util`,
`utilNombreDelVector` o `util_nombre_del_vector`.

```
int v[100] = {9, 4, 7};  
int util_v = 3;  
    v
```





Sobre el tamaño de los vectores II

```
1 int main(){
2     const int DIM_NOTAS = 100;
3     double notas[DIM_NOTAS];
4     int util_notas;
5     double media;
6     do{
7         cout << "Introduzca el número de alumnos (entre 1 y "
8             << DIM_NOTAS << "): ";
9         cin >> util_notas;
10    }while (util_notas<1 ||util_notas>DIM_NOTAS);
11
12    for (int i=0; i<util_notas; i++){
13        cout << "nota["<< i << "]": ";
14        cin >> notas[i];
15    }
16
```





Sobre el tamaño de los vectores III

```
17  
18  
19     media=0;  
20     for (int i=0; i<util_notas; i++)  
21         media += notas[i];  
22     media /= util_notas;  
23     cout << "\nMedia: " << media << endl;  
24 }
```



Sobre el tamaño de los vectores IV

Un método alternativo es usar un elemento *especial* que indique el final del vector (**elemento centinela**).

```
1 int main(){
2     const int DIM_NOTAS = 100;
3     double notas[DIM_NOTAS];
4     double media;
5     int i;
6
7     cout << "nota[0]: (-1 para terminar): ";
8     cin >> notas[0];
9     for(i=1; notas[i-1] != -1 && i < DIM_NOTAS-1; i++){
10         cout << "nota[" << i << "]: (-1 para terminar): ";
11         cin >> notas[i];
12     }
13     notas[i] = -1;
14
15 }
```





Sobre el tamaño de los vectores V

```
16     media=0;
17     for (i=0; notas[i] != -1; i++)
18         media += notas[i];
19
20     if (i == 0)
21         cout << "No se introdujo ninguna nota\n";
22     else{
23         media /= i;
24         cout << "\nMedia: " << media << endl;
25     }
26 }
```



Modularización y vectores

El paso de vectores a funciones se hace mediante un parámetro formal del mismo tipo **exactamente** (no puede ser compatible).

```
1 #include <iostream>
2 using namespace std;
3
4 void imprime_vector (char v[5]){
5     for (int i=0; i<5; i++)
6         cout << v[i] << " ";
7 }
8 int main(){
9     char vocales[5]={'a','e','i','o','u'};
10    imprime_vector(vocales);
11 }
```



Si necesitamos usar el mismo algoritmo para diferentes tipos de dato tendremos que implementar una función para cada tipo.



Modularización y vectores

C++ permite usar un vector sin dimensiones como parámetro formal.
Necesitamos saber el número de componentes usadas.

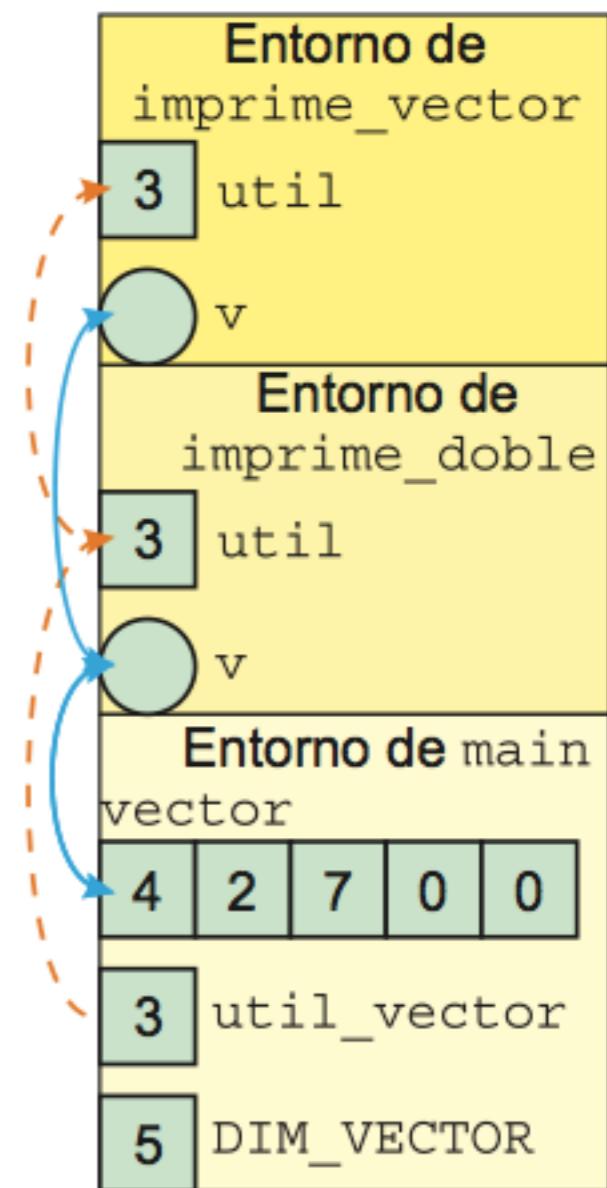
```
1 #include <iostream>
2 using namespace std;
3
4 void imprime_vector(char v[], int util){
5     for (int i=0; i<util; i++)
6         cout << v[i] << " ";
7 }
8 int main(){
9     char vocales[5]={'a','e','i','o','u'};
10    char digitos[10]={'0','1','2','3','4',
11                      '5','6','7','8','9'};
12    imprime_vector(vocales, 5); cout<<endl;
13    imprime_vector(digitos, 10); cout<<endl;
14    imprime_vector(digitos, 5); cout<<endl; // del '0' al '4'
15    imprime_vector(vocales, 100); cout<<endl; // ERROR
16 }
```





Los vectores **SIEMPRE** se pasan por referencia, en el sentido de que podemos modificar las componentes pero **no se pone &**.

```
1 #include <iostream>
2 using namespace std;
3
4 void imprime_vector(int v[], int util){
5     for (int i=0; i<util; i++)
6         cout << v[i] << " ";
7 }
8 void imprime_doble(int v[], int util){
9     for (int i=0; i<util; i++)
10        v[i] *= 2;
11    imprime_vector(v, util);
12 }
13 int main(){
14     const int DIM_VECTOR = 5;
15     int vector[DIM_VECTOR]={4,2,7};
16     int util_vector=3;
17     imprime_doble(vector, util_vector);
18     imprime_vector(vector, util_vector);
19 }
```





Modularización y vectores

Problema

Es imposible pasar un vector por valor a una función.

Opción: Vectores de constantes

Utilizando el calificador `const`.

```
1 void imprime_vector(const int v[], int util){  
2     for (int i=0; i<util; i++)  
3         cout << v[i] << " ";  
4 }  
5 void imprimedoble(const int v[], int util){  
6     for (int i=0; i<util; i++)  
7         v[i] *= 2; // ERROR de compilación  
8     imprime_vector(v, util);  
9 }
```



Modularización y vectores

- Si no se utiliza el calificador `const`, el compilador asume que el vector se va a modificar (aunque no se haga).
- No es posible pasar un vector de constantes a una función cuya cabecera indica que el vector se modifica (aunque la función realmente no modifique el vector)

```
1 #include<iostream>
2 using namespace std;
3
4 void imprime_vector (char v[]){
5     for (int i=0; i<5; i++)
6         cout << v[i] << " ";
7 }
8 int main(){
9     const char vocales[5]={‘a’, ‘e’, ‘i’, ‘o’, ‘u’};
10    imprime_vector(vocales); // ERROR de compilacion
11 }
```



Construcción de vectores en funciones I

Si queremos que una función **devuelva** un vector éste no puede ser local ya que al terminar la función su zona de memoria *desaparece*.

Debemos declarar dicho vector en el **main** (en general, en la función llamadora) y pasarlo como parámetro.

```
1 #include <iostream>
2 using namespace std;
3
4 void imprime_vector(const int v[], int util);
5 void solo_pares(const int v[], int util_v,
6                  int pares[], int &util_pares);
7 int main(){
8     const int DIM=100;
9     int entrada[DIM] = {8,1,3,2,4,3,8},
10        salida[DIM];
11    int util_entrada = 7, util_salida;
12    solo_pares(entrada, util_entrada, salida, util_salida);
13    imprime_vector(salida, util_salida);
```





Construcción de vectores en funciones II

```
14 }
15 void solo_pares(const int v[], int util_v,
16                 int pares[], int &util_pares){
17     util_pares=0;
18     for (int i=0; i<util_v; i++)
19         if (v[i]%2 == 0){
20             pares[util_pares]=v[i];
21             util_pares++;
22         }
23 }
24 void imprime_vector(const int v[], int util){
25     for (int i=0; i<util; i++)
26         cout << v[i] << " ";
27 }
```



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

```
1 #include <iostream>
2 using namespace std;
3
4 void imprime_vector(const char v[], int util);
5 void quita_repes(const char original[], int util_original,
6                   char destino[], int &util_destino);
7 int main(){
8     const int DIM =100;
9     char entrada[DIM]={ 'b','b','i','e','n','n','n'},
10        salida[DIM];
11    int util_entrada = 7, util_salida;
12
13    quita_repes(entrada, util_entrada, salida, util_salida);
14    imprime_vector(salida, util_salida);
15 }
16
```





```
17 void quita_repes(const char original[], int util_original,
18                     char destino[], int &util_destino){
19     destino[0] = original[0];
20     util_destino = 1;
21     for (int i=1; i<util_original;i++)
22         if (original[i] != original[i-1]){
23             destino[util_destino] = original[i];
24             util_destino++;
25         }
26     }
27
28 void imprime_vector(const char v[], int util){
29     for (int i=0; i<util; i++)
30         cout << v[i] << " ";
31 }
```



Trabajando con vectores locales

Comprobar si un vector de caracteres es un palíndromo.

Algoritmo:

- ① Eliminar espacios en blanco
- ② Recorrer el vector desde el principio hasta la mitad
 - ① Comprobar que el elemento en la posición actual desde el inicio es igual al elemento en la posición actual desde el final.

Necesitamos un vector local donde guardar el resultado del paso 1. ¿Cómo lo declaramos?



Trabajando con vectores locales

Usando una constante global

```
const int DIM = 100;

bool palindromo(const char v[], int longitud){
    char sinespacios[DIM];
    .....
}

int main(){
    char entrada[DIM];
```

Es la única solución (de momento).

Inconveniente: no podemos separar las implementación de `palindromo` de la definición de la constante.



```
1 #include <iostream>
2 using namespace std;
3 const int DIM = 100;
4
5 void quita_espacios(const char original[],
6                      int util_original,
7                      char destino[], int &util_destino);
8 void imprimecad(const char v[], int util);
9 bool palindromo(const char v[], int longitud);
10 int main(){
11     char entrada1[DIM]={'a','n','i','l','i','n','a'};
12     int util_entrada1=7;
13     char entrada2[DIM]={'d','a','b','a','l','e','','',
14     'a','r','r','o','z','','','a','','','l','a','','',
15     'z','o','r','r','a','','','e','l','','','a','b','a','d'};
16     int util_entrada2=31;
17
18     imprimecad(entrada1, util_entrada1);
19     if (palindromo(entrada1, util_entrada1))
20         cout << " es palíndromo\n";
```





```
21     else
22         cout << " no es palíndromo\n";
23
24     imprimecad(entrada2, util_entrada2);
25     if (palindromo(entrada2, util_entrada2))
26         cout << " es palíndromo\n";
27     else
28         cout << " no es palíndromo\n";
29 }
30
31 void quita_espacios(const char original[],
32                      int util_original,
33                      char destino[], int &util_destino){
34     util_destino=0;
35     for (int i=0; i<util_original; i++)
36         if (original[i] != ' ')
37             destino[util_destino]=original[i];
38             util_destino++;
39     }
40 }
```



```
41 bool palindromo(const char v[], int longitud){  
42     bool espalindromo = true;  
43     char sinespacios[DIM];  
44     int long_real;  
45  
46     quita_espacios(v, longitud, sinespacios, long_real);  
47  
48     for (int i=0; i< long_real/2 && espalindromo; i++)  
49         if(sinespacios[i] != sinespacios[long_real-1-i])  
50             espalindromo = false;  
51  
52     return espalindromo;  
53 }  
54  
55 void imprimecad(const char v[], int util){  
56     for (int i=0; i<util; i++)  
57         cout << v[i];  
58 }
```



Cadenas de caracteres

Cadena de caracteres

Secuencia ordenada de caracteres de longitud variable.

Permiten trabajar con datos como apellidos, direcciones, etc...

Tipos de cadenas de caracteres en C++

- ① **cstring**: cadena de caracteres heredado de C.
- ② **string**: cadena de caracteres propia de C++ (se estudiarán en MP2).

Cadenas de caracteres de C

Un vector de tipo **char** de un tamaño determinado acabado en un carácter especial, el carácter '**\0**' (carácter nulo), que marca el fin de la cadena (véase ▶ uso del elemento centinela).



Literales de cadena de caracteres

- Un literal de cadena de caracteres es una secuencia de cero o más caracteres encerrados entre comillas dobles.
- Su longitud es el número de caracteres que tiene.
- Su tipo es un vector de **char** con un tamaño igual a su longitud más uno (para el carácter nulo).

"Hola" de tipo **const char[5]**

"Hola mundo" de tipo **const char[11]**

"" de tipo **const char[1]**



Cadenas de caracteres

Cadenas de caracteres: declaración e inicialización

```
char nombre[10] ={'J','a','v','i','e','r','\0'};
```

'J'	'a'	'v'	'i'	'e'	'r'	'\0'	?	?	?
-----	-----	-----	-----	-----	-----	------	---	---	---

```
char nombre[] ={'J','a','v','i','e','r','\0'}; // Asume  
char[7]
```

Equivalente a las anteriores son:

```
char nombre[10] = "Javier";
```

```
char nombre[] = "Javier";
```

¡Cuidado!

```
char cadena[] = "Hola"; //char[5]
```

```
char cadena[] = {'H','o','l','a'}; // char[4]
```



Paso de cadenas a funciones I

El paso de cadenas corresponde al paso de un vector a una función. Como la cadena termina con el carácter nulo, no es necesario especificar su tamaño.

Función que nos diga la longitud de una cadena

```
1 int longitud(const char cadena[]){
2     int i=0;
3     while (cadena[i]!='\0')
4         i++;
5     return i;
6 }
```





Paso de cadenas a funciones II

Función que concatena dos cadenas

```
1 void concatena(const char cad1[], const char cad2[],
2                 char res[]){
3     int pos=0;
4     for (int i=0;cad1[i]!='\0';i++){
5         res[pos]=cad1[i];
6         pos++;
7     }
8     for (int i=0;cad2[i]!='\0';i++){
9         res[pos]=cad2[i];
10        pos++;
11    }
12    res[pos]='\0';
13 }
```





Entrada/salida de cadenas

Para leer y escribir cadenas se pueden usar las operaciones de lectura y escritura ya conocidas.

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5     char nombre[80];
6     cout << "Introduce tu nombre:  ";
7     cin >> nombre;
8     cout << "El nombre introducido es: " << nombre;
9 }
```



Problema

cin se detiene cuando encuentra un separador



Entrada/salida de cadenas

Solución:

`cin.getline(<cadena>, <tamaño>);`

lee hasta que se encuentra un salto de línea o no hay más espacio

```
1  char nombre[80],direccion[120];
2  int edad;
3  cout << "Introduce tu nombre:  ";
4  cin.getline(nombre,80);
5  cout << "El nombre introducido es:  "<< nombre;
6  cout << "\nIntroduce tu edad:  ";
7  cin >> edad;
8  cout << "La edad introducida es:  "<< edad;
9  cout << "\nIntroduce tu direccion:  ";
10 cin.getline(direccion,120);
11 cout << "La direccion introducida es:  "<< direccion;
```





La biblioteca `cstring` II

```
1 #include<iostream>
2 #include<cstring>
3 using namespace std;
4 int main(){
5     const int DIM=100;
6     char c1[DIM]="Hola";
7     char c2[DIM];
8
9     strcpy(c2, "mundo");
10    strcat(c1, " ");
11    strcat(c1, c2);
12    cout <<"Longitudes:"<<strlen(c1)<< " "<<strlen(c2);
13    cout << "\nc1: "<< c1 << " c2: "<< c2;
14    if (strcmp(c1,"adiós mundo cruel") < 0)
15        cout << "\nCuidado con las mayúsculas\n";
16    if (strcmp(c2, "mucho") > 0)
17        cout << "\n\"mundo\" es mayor que \"mucho\"\n";
18 }
```





Parte II

Algoritmos de búsqueda y ordenación



Algoritmos de Búsqueda

Búsqueda

Proceso de encontrar un elemento específico en un vector

Tipos

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



Búsqueda secuencial

- El vector se supone no ordenado.
- El vector no se modifica: se pasa como `const <tipo> v[]`
- Devuelve la posición donde está o -1 si no está

Algoritmo

- ① Situarse al principio del vector
- ② Mientras tenga componentes en el vector y no lo haya encontrado
 - ① Comparar el elemento a buscar con el elemento actual
 - ② Si coinciden, ya lo he encontrado: recordar la posición.
 - ③ En otro caso, avanzar al siguiente elemento



Búsqueda secuencial. Primera aproximación

```
1 int BuscaSec(const double v[], int util_v, double buscado){  
2     int i, posicion;  
3     bool encontrado;  
4  
5     i=0;  
6     encontrado=false;  
7     while ((i<util_v) && !encontrado)  
8         if (v[i] == buscado){  
9             posicion=i;  
10            encontrado=true;  
11        } else  
12            i++;  
13  
14    if (encontrado)  
15        return posicion;  
16    else  
17        return -1;  
18 }
```



Búsqueda secuencial. Usando un bucle for

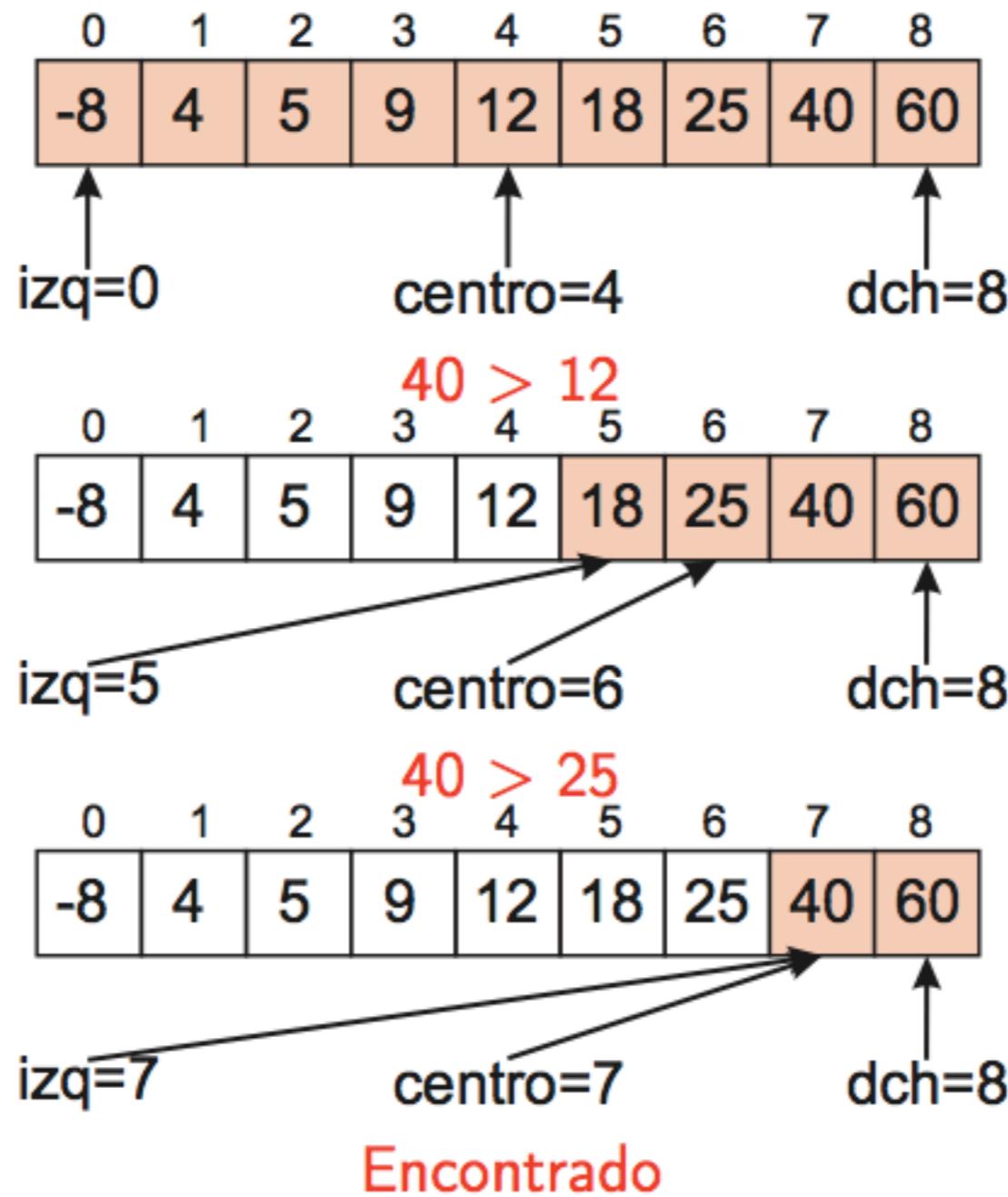
```
1 int BuscaSec(const double v[], int util_v, double buscado){  
2     int i;  
3     bool encontrado = false;  
4  
5     for (i=0; (i<util_v) && !encontrado; i++)  
6         encontrado = (v[i] == buscado);  
7  
8     if (encontrado)  
9         return i-1;  
10    else  
11        return -1;  
12 }
```



```
1 int BuscaBinaria(const double v[], int util_v,
2                      double buscado) {
3     int izq = 0;
4     int dch = util_v-1;
5     int centro = (izq+dch)/2;
6
7     while ((izq<=dch) && (v[centro] != buscado)) {
8         if (buscado < v[centro])
9             dch = centro-1;
10        else
11            izq = centro+1;
12        centro = (izq+dch)/2;
13    }
14
15    if (izq>dch)
16        return -1;
17    else
18        return centro;
19 }
```

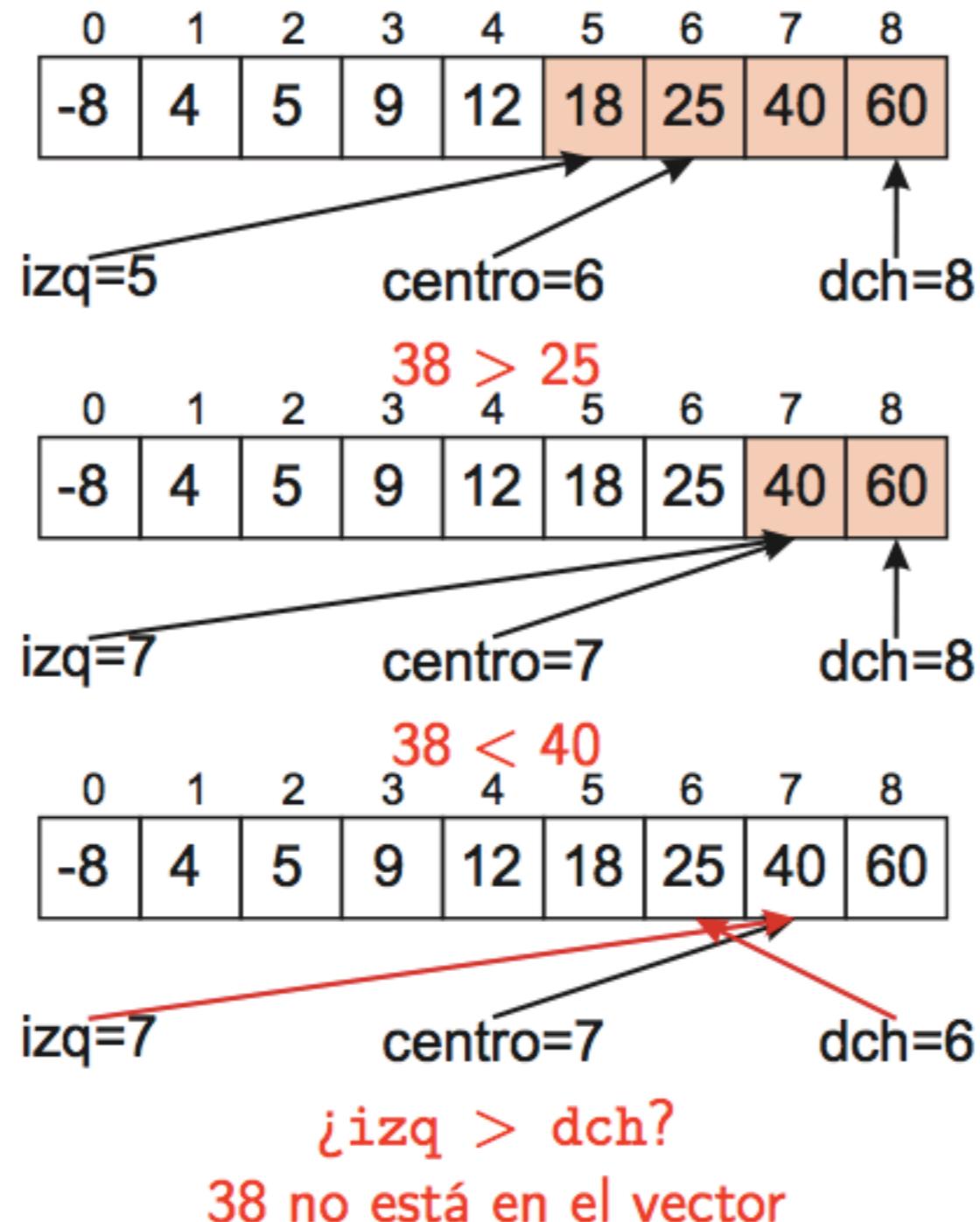


Ejemplo: Buscar el número 40





Ejemplo: Buscar el número 38





Algoritmos de ordenación

Ordenación

Procedimiento mediante el cual se disponen los elementos de un vector en un orden especificado, tal como orden alfabético u orden numérico creciente o decreciente.

Tipos

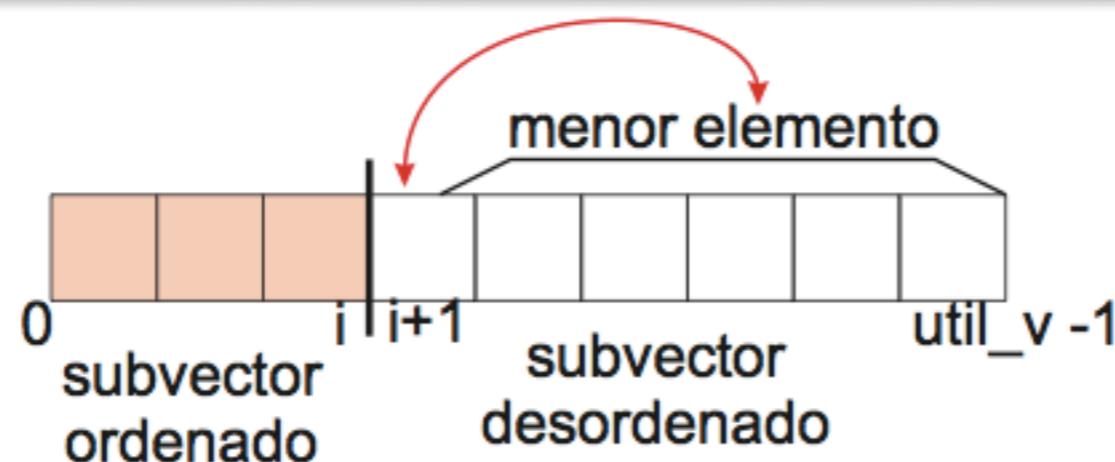
- *Ordenación interna.* Todos los datos están en memoria principal durante el proceso de ordenación.
Estudiaremos los siguientes:
 - Inserción.
 - Selección.
 - Intercambio.
- *Ordenación externa.* Parte de los datos a ordenar están en memoria externa mientras que otra parte está siendo ordenada en memoria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



Ordenación por selección

Idea básica

- ① Seleccionar el menor elemento del vector de tamaño $util_v$ e intercambiarlo con el primero. Así obtenemos un subvector ordenado de tamaño 1 y un subvector desordenado de tamaño $util_v - 1$.
- ② Repetir el proceso de seleccionar el menor elemento del subvector desordenado e intercambiarlo con el primero hasta que el subvector desordenado tenga un único elemento, que forzosamente será el mayor de todos los valores, por lo que el vector estará completamente ordenado.





Ordenación por selección. Usando funciones

```
1 int PosMinimo (const double v[], int izda, int dcha){  
2     double minimo = v[izda];  
3     int posicion_minimo = izda;  
4     for (int i=izda+1 ; i <= dcha ; i++)  
5         if (v[i] < minimo){  
6             minimo = v[i];  
7             posicion_minimo = i;  
8         }  
9     return posicion_minimo;  
10 }  
11  
12 void OrdSeleccion (double v[], int util_v){  
13     int pos_min;  
14     for (int i=0 ; i<util_v-1 ; i++){  
15         pos_min = PosMinimo(v, i, util_v-1);  
16         IntercambiaDouble(v[i],v[pos_min]);  
17     }  
18 }
```

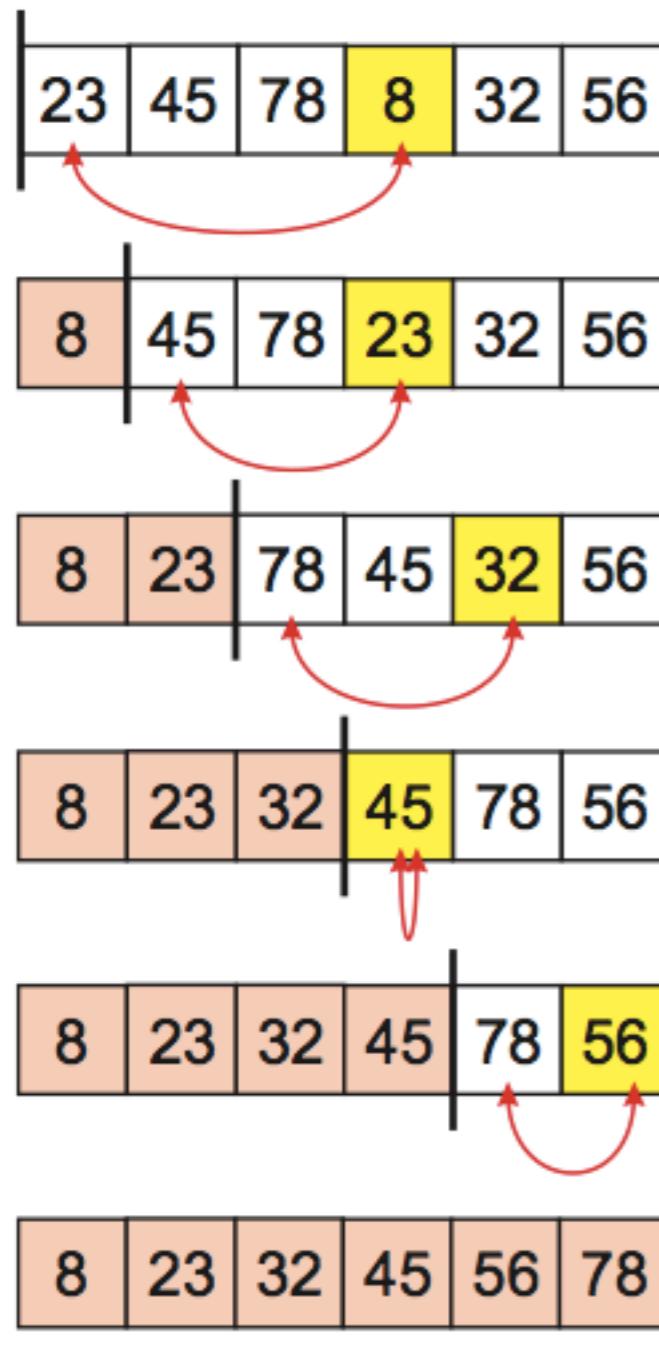


Ordenación por selección. Versión sin funciones

```
1 void OrdSeleccion(double v[], int util_v){  
2     int pos_min;  
3     double aux;  
4  
5     for (int i=0; i<util_v-1; i++){  
6         pos_min=i;  
7         for (int j=i+1; j<util_v; j++)  
8             if (v[j] < v[pos_min])  
9                 pos_min=j;  
10  
11     aux = v[i];  
12     v[i] = v[pos_min];  
13     v[pos_min] = aux;  
14 }  
15 }
```



Ejemplo



Iteración 1

Iteración 2

Iteración 3

Iteración 4

Iteración 5

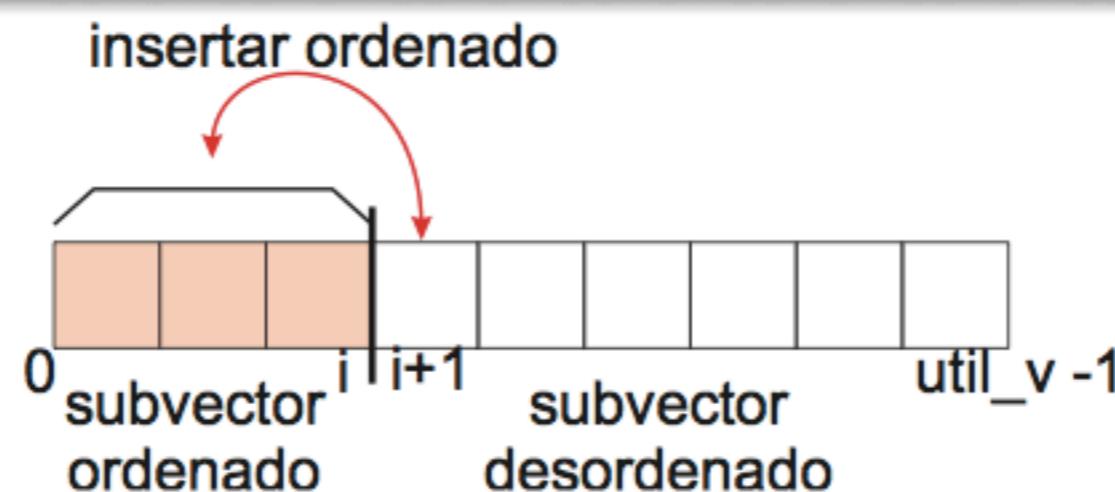
Estado Final



Ordenación por inserción

Idea básica

- ① Partimos de que el vector se divide en un subvector ordenado con un único elemento (el primero) y un vector desordenado de tamaño $util_v - 1$.
- ② Repetir el proceso de seleccionar el primer elemento del subvector desordenado e insertarlo en el vector ordenado de forma que siga estando ordenado. De esta forma el subvector ordenado tiene un elemento más.





Insertar de forma ordenada

Ejemplo: Insertar 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,?,?)

Precondiciones:

- ① El vector debe estar ordenado de forma ascendente.
- ② util < DIM.

```
1 void InsertaOrd(double v[], int &util, int valor){  
2     for (int i=util; i>0 && valor<v[i-1]; i--)  
3         v[i]=v[i-1];  
4  
5     v[i]=valor;  
6     util++;  
7 }
```



Ejemplo: Construir una función que lea enteros desde el teclado hasta leer un -1 y devuelva un vector ordenado. El -1 no debe estar en el vector

```
1 void InsertaOrd(int v[], int &util, int valor){  
2     int i;  
3     for (i=util; i>0 && valor<v[i-1]; i--)  
4         v[i]=v[i-1];  
5     v[i]=valor;  
6     util++;  
7 }  
8 void LeeYOrdena(int v[], int &util){  
9     int valor;  
10    util =0; // v inicialmente vacio  
11    cout << "Introduce un entero (-1 para terminar)";  
12    cin >> valor;  
13    while(valor != -1){  
14        InsertaOrd(v, util, valor);  
15        cout << "Introduce un entero (-1 para terminar)";  
16        cin >> valor;  
17    }  
18 }
```





Ordenación por inserción. Usando funciones

```
1 void InsertaOrd(double v[], int &util, int valor){  
2     for (int i=util; i>0 && valor<v[i-1]; i--)  
3         v[i]=v[i-1];  
4  
5     v[i]=valor;  
6     util++;  
7 }
```

```
1 void OrdInsercion (double v[], int util_v){  
2     int izq=1;  
3     while(izq<util_v)  
4         InsertaOrd(v, izq, v[izq]);  
5 }
```

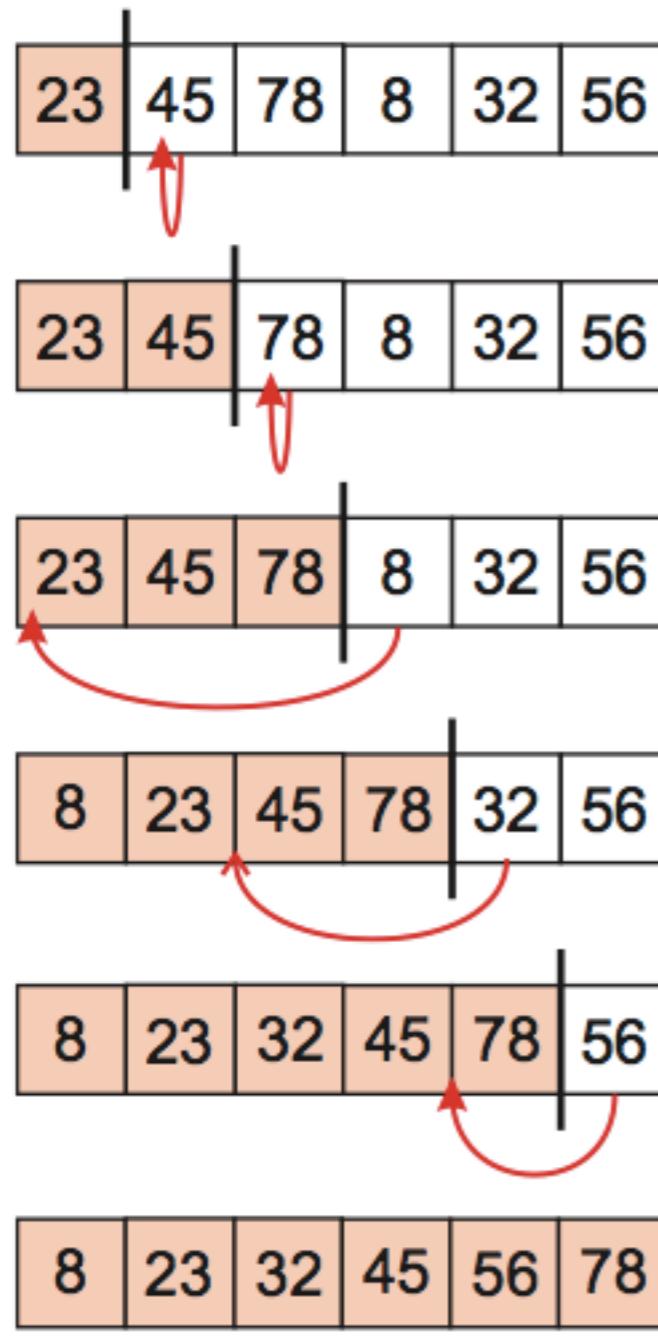


Ordenación por inserción. Versión sin funciones

```
1 void OrdInsercion (double v[], int util_v){  
2     int izda, i;  
3     double valor;  
4  
5     for (izda=1; izda<util_v; izda++){  
6         valor = v[izda];  
7  
8         for (i=izda; i>0 && valor<v[i-1]; i--)  
9             v[i] = v[i-1];  
10  
11         v[i] = valor;  
12     }  
13 }
```



Ejemplo



Iteración 1

Iteración 2

Iteración 3

Iteración 4

Iteración 5

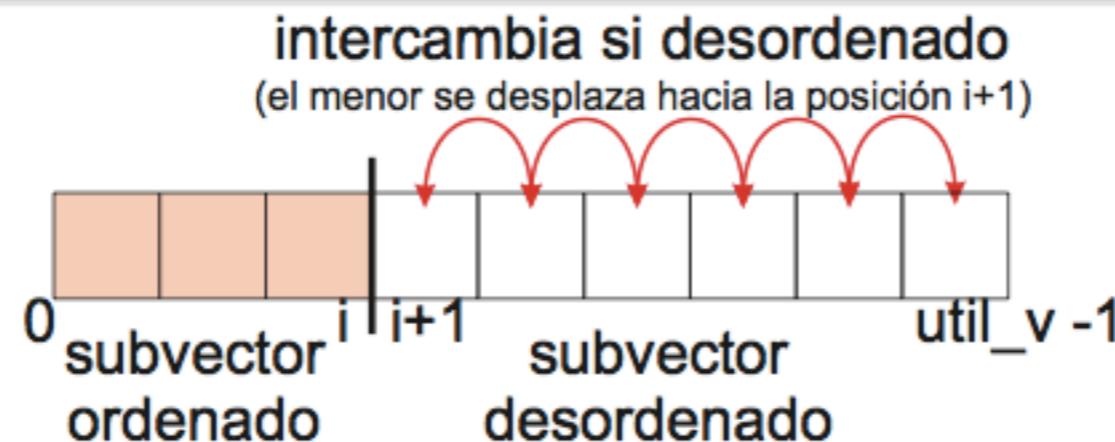
Estado Final



Ordenación por intercambio directo (Método de la burbuja)

Idea básica

- ① Desde el final del vector vamos comparando pares de elementos contiguos y, si no están ordenados, se intercambian. Cuando lleguemos al principio del vector el menor de los elementos estará al principio y ya tendremos un elemento en orden.
- ② Repetir el proceso de recorrer la parte desordenada del vector desde el final comparando e intercambiando pares de elementos contiguos desordenados hasta que el subvector desordenado tenga un único elemento.





Ejemplo (primera pasada)

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

No intercambiar

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

No intercambiar

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

Intercambiar

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

Intercambiar

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

Intercambiar

Iteración 1

Ver como el elemento menor (8) se va a ir desplazando hasta la primera posición



Ejemplo (resto de pasadas)

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

Iteración 2

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

Iteración 3

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

Iteración 4

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

Iteración 5

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

Estado Final

Observad que en las 2 últimas iteraciones no ha habido cambios (puesto que el vector ya estaba ordenado)



Ordenación por intercambio directo (Método de la burbuja). Usando funciones

```
1 void OrdBurbuja (double v[], int util_v){  
2     for (int izda=0; izda<util_v; izda++)  
3         for (int i=util_v-1 ; i>izda ; i--)  
4             if (v[i] < v[i-1])  
5                 IntercambiaDouble(v[i], v[i-1]);  
6 }
```

Mejora: Sabemos que si en una pasada del bucle interior no se produce ningún intercambio, el vector ya está ordenado.

Comprobar en el bucle exterior si no se ha producido intercambio en cada pasada.



Ordenación por intercambio directo (Método de la burbuja). Versión mejorada y sin funciones

```
1 void OrdBurbuja (double v[], int util_v){  
2     bool cambio = true;  
3     for (int izda=0; izda<util_v && cambio; izda++){  
4         cambio = false;  
5         for (int i=util_v-1 ; i>izda ; i--)  
6             if (v[i] < v[i-1]){  
7                 cambio=true;  
8  
9                 double aux = v[i];  
10                v[i] = v[i-1];  
11                v[i-1]=aux;  
12            }  
13        }  
14 }
```



Parte III

Matrices

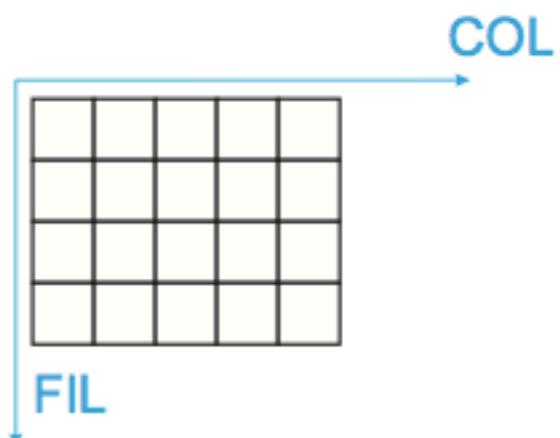


Declaración de matrices de 2 dimensiones

<tipo> <identificador> [DIM_FIL][DIM_COL];

- El tipo base de la matriz es el mismo para todas las componentes.
- Ambas dimensiones han de ser de tipo entero

```
1 int main(){
2     const int DIM_FIL = 2;
3     const int DIM_COL = 3;
4
5     double parcela[DIM_FIL] [DIM_COL];
6 }
```





Inicialización

- “Forma segura”: Poner entre llaves los valores de cada fila.

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

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

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

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

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



La declaración en detalle

- El compilador procesa las matrices como vectores de vectores.
- Es decir, es un vector con un tipo base también vector (cada fila).

- En la declaración

`int m[2] [3]`

`m` es un vector de 2 elementos (`m[2]`) y cada elemento es un vector de 3 `int` (`int xxxx[3]`).

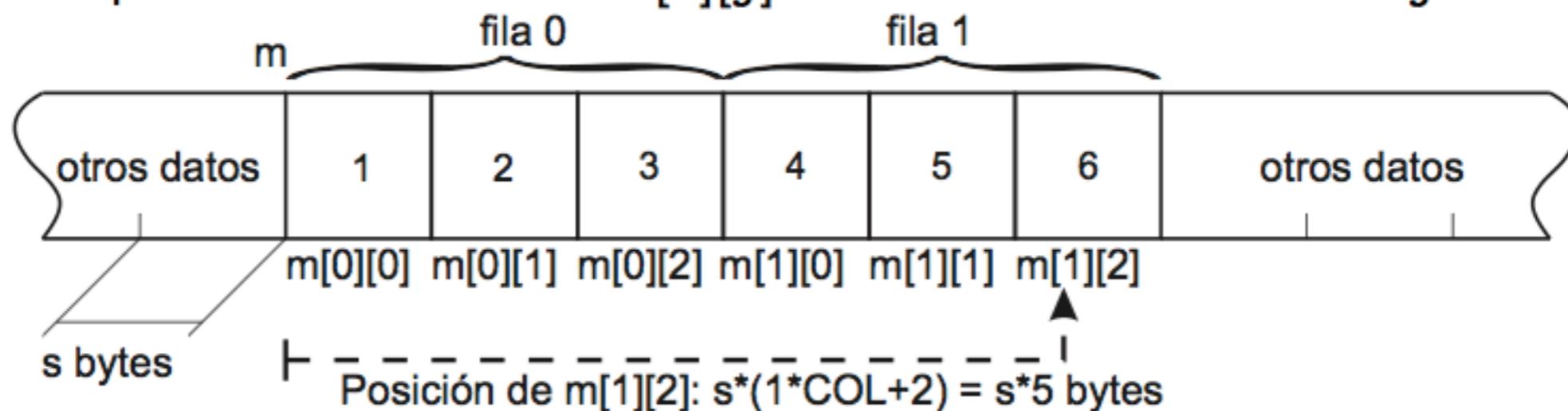
- Observad que la sintaxis de la inicialización es la de un vector de vectores

`int m[2] [3]={ {1,2,3}, {4,5,6} };`



Almacenamiento y límites de matrices

- Todos los elementos de las matrices se almacenan en un bloque contiguo de memoria.
- La organización depende del lenguaje: en C++ se almacenan por filas.
- Para acceder al elemento $m[i][j]$ en una matriz $\text{FIL} \times \text{COL}$ el compilador debe *pasar a la fila i* y desde ahí moverse j elementos
- La posición del elemento $m[i][j]$ se calcula como $i * \text{COL} + j$





Acceso, asignación, lectura y escritura

Acceso

`<identificador> [<ind1>][<ind2>]` (los índices comienzan en cero).
`<identificador> [<ind1>][<ind2>]` es una variable más del programa
y se comporta como cualquier variable del tipo de dato base de la matriz.
¡El compilador no comprueba que los accesos sean correctos!

Asignación

`<identificador> [<ind1>][<ind2>] = <expresión>;`
`<expresión>` ha de ser compatible con el tipo base de la matriz.

Lectura y escritura

`cin >> <identificador> [<ind1>][<ind2>];`
`cout << <identificador> [<ind1>][<ind2>];`



Sobre el tamaño de las matrices I

Para cada dimensión usaremos una variable que indique el número de componentes usadas.

```
1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     const int FIL=20, COL=30;
6     double m[FIL][COL];
7     int fil_enc, col_enc, util_fil,
8         util_col, f, c;
9     double buscado;
10    bool encontrado;
11
12    do{
13        cout << "Introducir el número de filas: ";
14        cin >> util_fil;
15    }while ((util_fil<1) || (util_fil>FIL));
```





Sobre el tamaño de las matrices II

```
16
17 do{
18     cout << "Introducir el número de columnas: ";
19     cin >> util_col;
20 }while ((util_col<1) ||(util_col>COL));
21
22 for (f=0 ; f<util_fil; f++)
23     for (c=0 ; c<util_col ; c++){
24         cout << "Introducir el elemento ("
25             << f << ","<< c << "): ";
26         cin >> m[f][c];
27     }
28
29 cout << "\nIntroduzca elemento a buscar: ";
30 cin >> buscado;
31
32
```



Sobre el tamaño de las matrices III

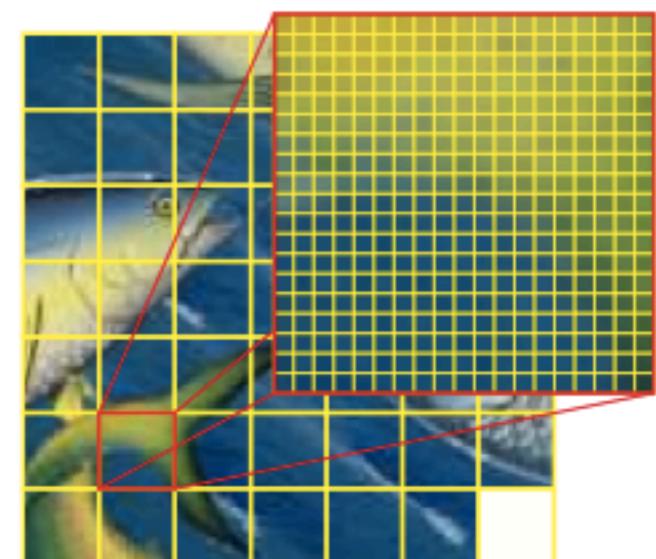
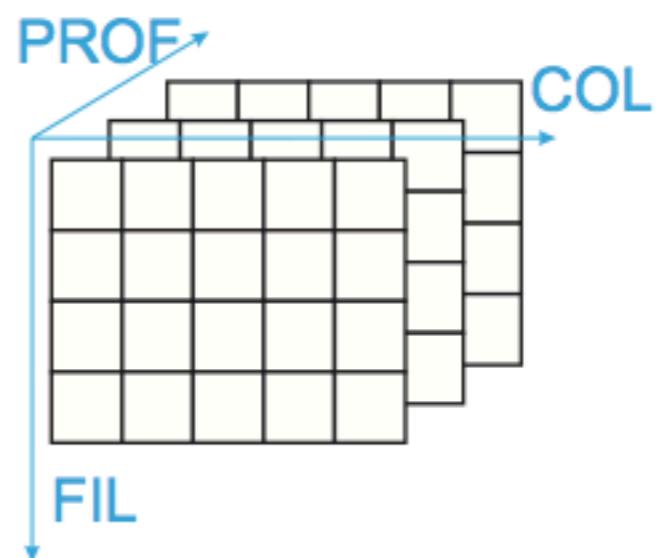
```
33     encontrado=false;
34     for (f=0; !encontrado && (f<util_fil) ; f++)
35         for (c=0; !encontrado && (c<util_col) ; c++)
36             if (m[f][c] == buscado){
37                 encontrado = true;
38                 fil_enc = f; col_enc = c;
39             }
40
41     if (encontrado)
42         cout << "Encontrado en la posición "
43             << fil_enc << "," << col_enc << endl;
44 else
45     cout << "Elemento no encontrado\n";
46
47 return 0;
48 }
```



Matrices de más de 2 dimensiones

Podemos declarar tantas dimensiones como queramos añadiendo más corchetes.

```
1 int main(){
2     const int FIL = 4;
3     const int COL = 5;
4     const int PROF =3;
5     double mat[PROF] [FIL] [COL] ;
6
7     double puzzle[7] [7] [19] [19];
8 }
```





Modularización con matrices

- Para pasar una matriz hay que especificar todas las dimensiones **menos la primera**

```
void lee_matriz(double m[][COL], int util_fil, int  
util_col);
```

- COL no puede ser local a main. Debe ser global

```
1 const int FIL=20, COL=30;  
2  
3 void lee_matriz(double m[] [COL] , int util_fil,  
4                   int util_col);  
5  
6 int main(){  
7     double m[FIL] [COL] ;  
8     int util_fil=7, util_col=12;  
9  
10    lee_matriz(m, util_fil, util_col);
```



Modularización con matrices

```
1 #include <iostream>
2 using namespace std;
3 const int FIL=20, COL=30;
4 void lee_matriz(double m[] [COL],
5                  int util_fil, int util_col){
6     for (int f=0 ; f<util_fil; f++)
7         for (int c=0 ; c<util_col ; c++){
8             cout << "Introducir el elemento (" 
9                 << f << "," << c << "):   ";
10            cin >> m[f] [c];
11        }
12    }
13 int lee_int(const char mensaje[], int min, int max){
14     int aux;
15     do{
16         cout << mensaje;
17         cin >> aux;
18     }while ((aux<min) || (aux>max));
19     return aux;
20 }
```





Modularización con matrices

```
21 void busca_matriz(const double m[][] [COL] , int util_fil,
22                 int util_col, double elemento,
23                 int &fil_encontrado, int &col_encontrado){
24     bool encontrado=false;
25     fil_encontrado = -1; col_encontrado = -1;
26     for (int f=0; !encontrado && (f<util_fil) ; f++)
27         for (int c=0; !encontrado && (c<util_col) ; c++)
28             if (m[f][c] == elemento){
29                 encontrado = true;
30                 fil_encontrado = f;
31                 col_encontrado = c;
32             }
33 }
34
35 int main(){
36     double m[FIL] [COL];
37     int fil_enc, col_enc, util_fil,
38         util_col;
39     double buscado;
40 }
```



Modularización con matrices

```
41 util_fil = lee_int("Introducir el número de filas: ",  
42                         1, FIL);  
43 util_col = lee_int("Introducir el número de columnas: ",  
44                         1, COL);  
45 lee_matriz(m, util_fil, util_col);  
46 cout << "\nIntroduzca elemento a buscar: ";  
47 cin >> buscado;  
48  
49 busca_matriz(m, util_fil, util_col, buscado,  
50                         fil_enc, col_enc);  
51 if (fil_enc != -1)  
52     cout << "Encontrado en la posición "  
53     << fil_enc << "," << col_enc << endl;  
54 else  
55     cout << "Elemento no encontrado\n";  
56  
57 return 0;  
58 }
```



Gestión de filas de una matriz como vectores I

- Dado que los elementos de cada fila están contiguos en memoria podemos gestionar cada fila como si fuese un vector.
- La fila i -ésima de una matriz m es $m[i]$.
- Cada fila $m[i]$ tiene $util_col$ componentes usadas

```
1 void busca_matriz(const double m[][][COL], int util_fil,
2                     int util_col, double elemento,
3                     int &fil_enc, int &col_enc){
4     fil_enc = -1;
5     col_enc = -1;
6     for (int f=0; col_enc == -1 && (f<util_fil); f++)
7         col_enc = busca_sec(m[f], util_col, elemento);
8     if (col_enc != -1)
9         fil_enc = f-1;
10 }
```



Gestión de filas de una matriz como vectores II

- Como toda la matriz está contigua en memoria, si la matriz está completamente llena, podemos hacer

```
1 void busca_matriz(const double m[] [COL], double elto,
2                 int &fil_encontrado, int &col_encontrado){
3     int encontrado = busca_sec(m[0], util_col*util_fil,
4                                 elto);
5     if (encontrado != -1){
6         fil_encontrado = encontrado / COL;
7         col_encontrado = encontrado % COL;
8     } else{
9         fil_encontrado = -1;
10        col_encontrado = -1;
11    }
12 }
```



Bibliografía

[2005] Garrido,A.“Programación en C++”



Jaime Matas Bustos
CFGS Desarrollo de Aplicaciones Multiplataforma
C.E.S. Cristo Rey

