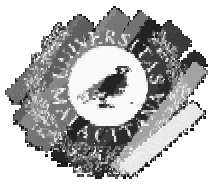


**Departamento de Lenguajes y Ciencias de la Computación**
**Universidad de Málaga**

# TEMA 6: Datos estructurados

## Fundamentos de Informática
## (Ingeniería Técnica Industrial)

**E.U. Politécnica**

**Autores:** M.C. Aranda, A. Fernández, J. Galindo, M. Trella

### *TEMA 6: Datos estructurados: Índice*

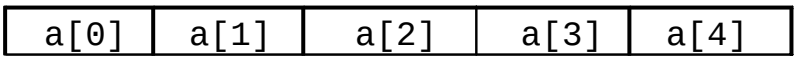
## *Arrays Unidimensionales: Declaración y Acceso*

- Un **array** es una colección de elementos del mismo tipo.
- DECLARACIÓN de un array:

```
tipo_elemento  nombre_array [num_elementos]
```

- ACCESO a cada elemento:
  - Cada elemento se referencia con un ÍNDICE entre corchetes **[ ]**
  - Los índices van de 0 hasta el tamaño del array menos uno.

**EJEMPLO:** `int a[5];`




- VENTAJAS de su uso frente a variables aisladas:
  - Podemos referirnos a todos los elementos a la vez (al declararlos, al pasarlos a una función, etc).
  - Permite resolver problemas de forma sencilla y compacta.



## *Arrays Unidimensionales: Declaración y Acceso*

**EJEMPLOS:**

- ✓ **`char c[5];`** Declara un array de **5 caracteres**, desde **`c[0]`** hasta **`c[4]`**
- ✓ **`float f[20];`** Declara un array de **20 reales**, desde **`f[0]`** hasta **`f[19]`**
- ✓ Programa que carga un array de **10 enteros** con los números del 0 al 9:

```
void main()
{ int x[10], t;
  for(t=0;t<10;t++)  x[t]=t;
}
```

- **¿Cómo se ALMACENAN los arrays en MEMORIA?**
  - Los elementos se almacenan **CONTIGUOS en memoria**.

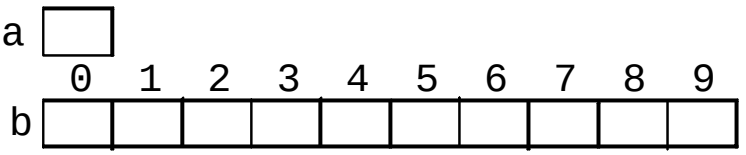
| | `char c[5]` | | | | | `int a[5]` | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Elemento** | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 2 | 3 | 4 |
| **Dirección** | 1000 | 1001 | 1002 | 1003 | 1004 | 1000 | 1002 | 1004 | 1006 | 1008 |

Suponemos que ambos arrays comienzan en la posición 1000, que un entero ocupa 2 bytes y un char ocupa 1 byte (cada número de posición indica 1 byte).

## *Arrays Unidimensionales: Nombres de Arrays*

- Consideremos las siguientes declaraciones:

```
int a;
int b[10];
```




- El tipo del valor de la variable **a** es un entero.
- Cada elemento del array **`b (b[0], ...,b[9])`** es un entero ⇓ puede ser usado en cualquier contexto donde es usado un entero.

### ¿De qué tipo es b (sin índice)?

- NO se refiere al array completo.
- El nombre de un array es un **"puntero constante"** que tiene la dirección del primer elemento del array.
- Por tanto, **b** es un puntero constante a un entero: El puntero **b** apunta al primer elemento del array **b**, o sea, a **`b[0]`** **`(b º &b[0])`**

EXCEPCIÓN: `sizeof(b)` devuelve el tamaño del array y no de un puntero.



## *Arrays Unidimensionales: Inicialización*

- Un array puede ser **inicializado** en su declaración utilizando llaves **`{}`** :

```
int vector[5]={10,20,30,40,50};
```

- Los valores se asignan a los elementos uno a uno.

| **`vector`** | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|

- **Inicialización incompleta:**
  - Se inicializan sólo los 4 **primeros** elementos:
    `int vector[5]={1,2,3,4};`
  - Si hay más valores, da ERROR:
    `int vector[5]={1,2,3,4,5,6};`
- **Tamaño automático de un array:**
  - El compilador cuenta el número de elementos y ese es el tamaño:
    `int vector[]={1,2,3,4,5};`

## *Arrays Unidimensionales: Argumentos en Funciones*

- En **C** TODOS los parámetros de una función se pasan por **VALOR**.
- Si el argumento es un array se pasa **un puntero al primer elemento** del array.
- Por tanto, el paso de un array tiene un comportamiento como si fuera un paso por REFERENCIA.
  - NO se puede modificar la dirección del array.
  - La función SI puede **modificar** los elementos del array.
- Los siguientes PROTOTIPOS de funciones son equivalentes:

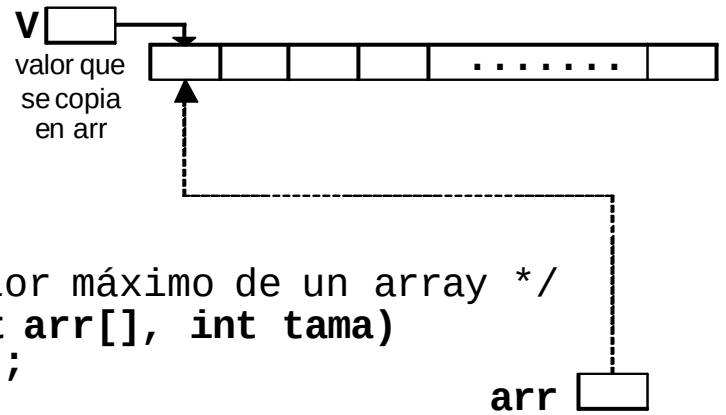
```
float  maximo(float arr[], int tama);
float  maximo(float *arr, int tama);
```

- Es imposible que la función determine el tamaño del array.
  `sizeof(arr)` da el tamaño del puntero, NO del array.
- Si se necesita el tamaño en la función se tiene que pasar como un argumento explícito (como **`tama`** en el ejemplo).



## *Arrays Unidimensionales: Argumentos en Funciones*

```
void main() {
float V[50],m;
      .....
  m=maximo(V,50);
      .....
}
/* Devuelve el valor máximo de un array */
float maximo(float arr[], int tama)
{ float max=arr[0];
  int i;
  for(i=1;i<tama;i++)
    if(max<arr[i])
      max=arr[i];
  return max;
}
```




- No se reserva espacio para el array (**`arr`** en el ejemplo) en la función: el parámetro es un puntero al espacio que fue previamente reservado en otro sitio (**`V`** en el ejemplo).
- Un parámetro array es compatible con arrays de **cualquier tamaño**.

## *Arrays Unidimensionales: Argumentos en Funciones*

```
/* Cálculo de la varianza de N números σ²= E(X²) - E(X)*E(X) donde E() es la esperanza matemática */
#include <stdio.h>
/* Devuelve la media */
float media(float a[],
            int tama)
{
  int i;
  float s=0;
  for(i=0;i<tama;i++)
    s=s+a[i];
  return (s/tama);
}
/* Eleva los elementos del array al cuadrado */
void potencia(float a[],
float a2[], int tama) {
  int i;
  for(i=0;i<tama;i++) {
    a2[i]=a[i]*a[i];
}
```

```
void main() {
  float t[10],t2[10],m1,m2;
  int i,N;
  printf("Tamaño? ");
  scanf("%d", &N);
  for(i=0;i<N;i++) {
    printf("Valor de t[%d]:"
           ,i);
    scanf("%f",&t[i]);
  }
  m1=media(t,N);
  potencia(t,t2,N);
  m2=media(t2,N);
  printf("Varianza: %d\n",
         m2 -(m1*m1));
}
```



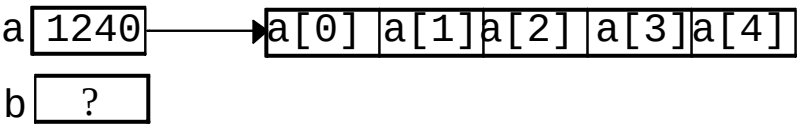
## *Arrays Unidimensionales: Arrays y Punteros*

– Hay una fuerte relación entre punteros y arrays.

– Los arrays y los punteros NO son equivalentes:

A) `int a[5];`

B) `int *b;`




DIFERENCIAS:

A) Declarar un array reserva memoria para TODOS los elementos.

B) Declarar un puntero reserva memoria SÓLO para el puntero.

A) El nombre de un array almacena una dirección fija (**constante**) que apunta al principio de este espacio (al primer elemento del array). NO se puede cambiar el valor de la **constante**.

B) Una variable puntero almacena una dirección de memoria que puede ser modificada. La variable puntero NO está inicializada para apuntar a ningún espacio existente.

## *Arrays Unidimensionales: Arrays y Punteros*

- Se puede acceder a los elementos de un array con **ÍNDICES** o con notación de **PUNTEROS.**
- Así, las siguientes expresiones son **equivalentes**:

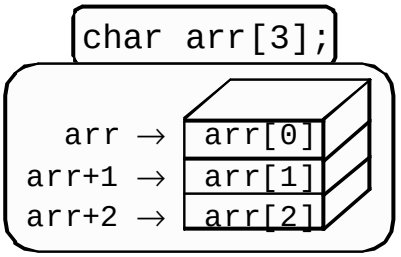
```
float arr[5];
```

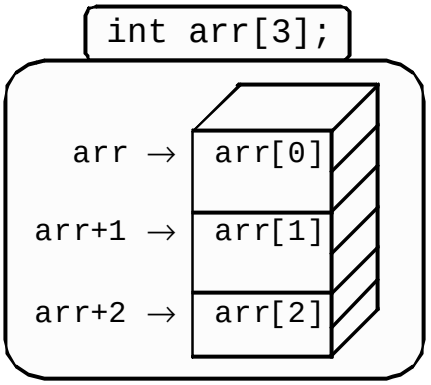
| Usando índices | Usando punteros |
|---|---|
| arr[0] | *(arr) |
| arr[1] | *(arr+1) |
| arr[2] | *(arr+2) |
| arr[3] | *(arr+3) |
| arr[4] | *(arr+4) |
| **arr[índice]** o | ***(arr+índice)** |

Direcciones


- El número de posiciones que se incrementa un puntero depende del tipo de datos del array.

(Suponemos que un `int` ocupa 2 bytes y un `char` 1 byte.)






## *Arrays Multidimensionales: Concepto y Declaración*

Un array es **MULTIDIMENSIONAL** si tiene más de una dimensión.

```
tipo_elemento  nombre_array [a][b][c]...[z];
```

– La "matriz" es un array de 2 dimensiones, es decir un array unidimensional de arrays unidimensionales.

– En general, un array de dimensión **n** es un array unidimensional de arrays de dimensión **n–1**.

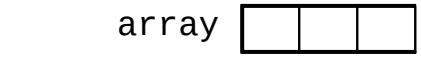
**EJEMPLOS:**

`int m[6][10];` Array bidimensional de 6×10 enteros (matriz)

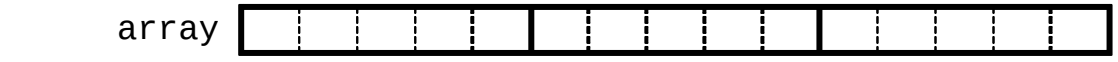
`float arr[3][2][5];` Array tridimensional de 3×2×5 reales (cubo)

**¿Cómo se ALMACENAN los elementos en MEMORIA?**

`int array[3];` Son 3 enteros



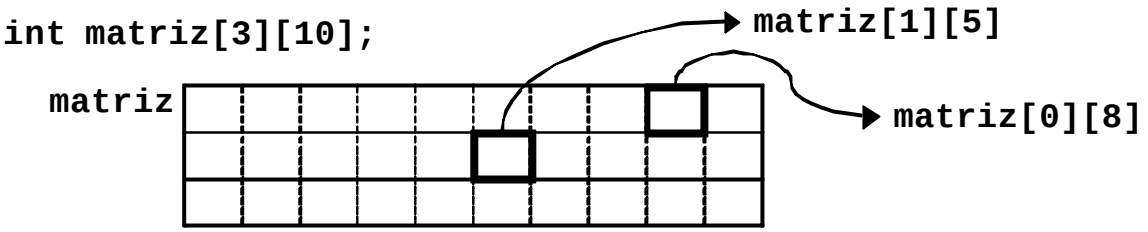

`int array[3][5];` Son 15 enteros




– **Los elementos se almacenan CONTIGUOS en memoria.**

## *Arrays Multidimensionales: Acceso a los Elementos (por Índices)*

- Para identificar un elemento de un array multidimensional, se debe dar **un índice para cada dimensión**, en el mismo orden que en la declaración.
- Cada índice se encierra en sus propios corchetes.




**EJEMPLO**: Cargar un array bidimensional con los números del 1 al 12

```
void main(){
 int num[3][4], i,j;
 for(i=0;i<3;i++)
    for(j=0;j<4;j++)
       num[i][j]=(i*4)+j+1;
}
```

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 5 | 6 | 7 | 8 |
| 2 | 9 | 10 | 11 | 12 |



## *Arrays Multidimensionales: Acceso a los Elementos. Ejemplo*

```
/* Programa que  muestra los valores almacenados en
   un arraybidimensional 2 por 3 */

#include <stdio.h>
#define FILAS    2
#define COLUMNAS  3
void main() {
 float matriz[FILAS][COLUMNAS]=
  {{1,2,3},{4,5,6}};
 int fil,col;
 for(fil=0;fil<FILAS;fil++)
   for(col=0;col<COLUMNAS;col++)
     printf("El valor de [%d][%d] es %d\n",
            fil,col,matriz[fil][col]);
}
```

**SALIDA EN PANTALLA**

```
El valor de [0][0] es 1
El valor de [0][1] es 2
El valor de [0][2] es 3
El valor de [1][0] es 4
El valor de [1][1] es 5
El valor de [1][2] es 6
```

## *Arrays Multidimensionales: Inicialización*

- Hay dos modos de escribir la lista de inicializaciones:

**1. Todos los valores seguidos:**

```
int   matriz[2][3]={0,1,2,10,11,12};
```

**2. Por partes:**

```
int   matriz[2][3]={
      { 0, 1, 2 },
      { 10, 11, 12 }
};
```

- El segundo modo tiene varias ventajas:
  - La organización: es más fácil de leer.
  - Son útiles para las inicializaciones incompletas.



## *Arrays Multidimensionales: Argumentos en Funciones*

- Un array **multidimensional** puede pasarse como argumento a una función.
- Se pasa un PUNTERO AL PRIMER ELEMENTO del array.
  - Recuerde que ese primer elemento es un array (de 1 dimensión menos).
- Se define la longitud de todas las dimensiones excepto la primera.
- Se puede usar el siguiente PROTOTIPO:

  **`void func(int mat[][10]);`** Acepta una matriz en la cual cada fila tiene 10 enteros y con cualquier número de filas.
- También es CORRECTO: **`void func(int (*mat)[10]);`**
- Es INCORRECTO: **`void func1(int **mat);`**

**EJEMPLO:**

- Función que devuelve el valor máximo de un array tridimensional de reales de dimensión **`tx3x5`** (**`t`** indica el tamaño de la primera dimensión).

```
float max(float d[][3][5],int t);
```

Observe que esa función sirve para arrays de diversos tamaños: 4x3x5, 29x3x5 .....

## *Arrays Multidimensionales: Argumentos en Funciones. Ejemplo*

```
/* Suma de matrices 10x10*/
#include <stdio.h>
/* Función que captura los
valores de una matriz 10x10 */
void captura(float m[][10]) {
 int i,j;
 for(i=0;i<10;i++)
   for(j=0;j<10;j++) {
     printf("Valor [%d,%d]: ",
             i,j);
     scanf("%f", &a[i][j]);
   }
}
/* Suma dos matrices 10x10 */
void suma(float m1[][10],
m2[][10], r[][10]) {
 int i,j;
 for(i=0;i<10;i++)
   for(j=0;j<10;j++) {
     r[i][j]=m1[i][j]+m2[i][j];
}
```

```
/* Muestra una matriz 10x10*/
void imprime(float m[][10]) {
 int i,j;
 for(i=0;i<10;i++) {
   for(j=0;j<10;j++)
     printf(" %.2f", m[i][j]);
   printf("\n");
  }
}
void main() {
  float a[10][10],b[10][10],
        c[10][10];
  captura(a);  captura(b);
  suma(a,b,c);
  imprime(c);
}
```

NOTA: Los arrays siempre se pasan por *referencia*. Por tanto, si se cambia el valor de los elementos dentro de una función, se refleja en la llamada (como ocurre en las funciones captura y suma).



## *Cadenas de Caracteres: Declaración*

- Una cadena de caracteres es un **array unidimensional de caracteres**.
- VENTAJA: Existen librerías con funciones para realizar la mayor parte de las operaciones básicas sobre cadenas.
- El último carácter visible de la cadena debe estar seguido del carácter nulo que se representa por **'\0'** (código ASCII 0).
- Este carácter marca el final de la cadena de caracteres.

DECLARACIÓN:

```
char cadena[20];
char *cadena="Hola";
char cadena[]="Adios";
```

✓ En las dos últimas declaraciones el tamaño del array será el número de caracteres dado en la inicialización más 1 (que guarda el carácter '\0').

## *Cadenas de Caracteres: Lectura*

- Una cadena de caracteres puede ser leída y asignada a un array utilizando **`scanf()`** con el modificador **`%s`**, o con **`gets()`** .
  - ✓ Con `scanf` se lee hasta el primer espacio, tabulador o retorno de carro.
  - ✓ Si se pone un tamaño `T` en el especificador de formado (`%Ts`) (ej. %5s), se lee como máximo ese número de caracteres.
  - ✓ Con `gets` se lee hasta el primer retorno de carro.
  - ✓ Para leer una frase (con espacios) de un tamaño concreto, se puede usar la función **`fgets()`**, cuyo prototipo es:

    **`char *fgets(char *s, int n, stdin);`** Lee caracteres hasta que lee n-1 o hasta que lee el carácter RETURN '\n', añadiendo el carácter '\0' al final. Si hay más de n-1 caracteres, quedan sin leer y podrán leerse después.
  - ✓ En cualquiera de los casos, es necesario haber especificado previamente el tamaño de la cadena.

EJEMPLO:

```
char nombre[20];
printf("Introduzca el nombre "); scanf("%s", nombre);
```



## *Cadenas de Caracteres: Lectura y Escritura*

- Se escribe con **`printf()`** con el modificador **`%s`** o con **`puts()`**.
  - ✓ La cadena debe contener el carácter '\0' para ser escrita correctamente.

EJEMPLOS:

```
char nombre[10],apellidos[20];
char direc []= "Avda. América, 35";
printf("Nombre? ");
scanf("%10s", nombre); /* Sólo lee una palabra.
            Observe que no hay que usar & ya que
            nombre es un puntero */
printf("Apellidos? ");
fgets(apellidos, 20, stdin);
printf("Nombre y apellidos: %s %s",nombre,apellidos);
printf("Dirección: %s", direc);
```

## *Cadenas de Caracteres: Ejemplo*

```
/* Calcula las veces que aparece el carácter ch en una
cadena str */
#include <stdio.h>
void main() {
   char str[50],ch;
   int veces=0,i=0;
   printf("Escribe un texto: ");
   gets(str);
   printf("Escribe el carácter para buscar: ");
   ch=getchar();
   while(str[i]!='\0') {
      if(str[i]==ch) veces++;
      i++;
   }
   printf("El número de ocurrencias es %d\n", veces);
}
```



## *Cadenas de Caracteres: Funciones Relacionadas*

- Algunas funciones de manejo de cadenas (**`string.h`**)

**`int strlen(char *str)`**

- ✓ Determina la longitud de una cadena.

**`int strcmp(char *str1,char *str2)`**

- ✓ Compara dos cadenas.
- ✓ Devuelve 0 si son iguales, un número negativo si **`str1`** es menor que **`str2`** o un número positivo si **`str1`** es mayor que **`str2`**.

**`char *strcpy(char *str1,char *str2)`**

- ✓ Copia **`str2`** en **`str1`** (carácter a carácter).

**`char *strcat(char *str1,char *str2)`**

- ✓ Añade **`str2`** al final de **`str1`** (carácter a carácter).

**NOTA**: **`strcpy()`** y **`strcat()`** devuelven un puntero a la cadena resultante. No comprueban si el resultado cabe en la cadena final.

## Cadenas de caracteres. Ejemplo

```
#include <stdio.h>
#include <string.h>
void main() {
  char cad1[10], cad2[10];
  strcpy(cad1, "Hola "); /* Se guardan en cad1 6
        caracteres (incluido el '\0')*/
  strcpy(cad2, "y adios"); /* Se guardan en cad2 8
        caracteres (incluido el '\0') */
  strcat(cad1, cad2); /* Se concatena la cadena cad2 al
        final de cad1. Observe que se intentan guardar
        en cad1 más caracteres que la longitud de la
        cadena. El compilador no da error y se escribe
        a continuación de cad1, en memoria que no
        pertenece a cad1. */
  puts(cad1); /* Se escribe cad1 hasta que encuentre un
        '\0'. El resultado es impredecible: puede
        escribir "Hola y  adios" a pesar de que en total
        son más de 10 caracteres, puede escribir otra
        cosa o puede quedarse colgado el ordenador. */
}
```

**Posible solución:** comprobar si se puede concatenar antes de hacerlo:

```
if(length(cad1)+length(cad2) < 10) strcat(cad1,cad2);
```



## Cadenas de Caracteres: Ejemplo

```
/* Comparación de dos cadenas usando strcmp() */
#include <stdio.h>
#include <string.h>
void main() {
   int n;
   char nombre1[20], nombre2[20];
   printf("De el primer nombre: ");
   scanf("%s", nombre1);
   printf("De el segundo nombre: ");
   scanf("%s", nombre2);
   n= strcmp(nombre1,nombre2);
   if(n==0)
      printf("Nombres Iguales\n");
   else if (n>0)
      printf("Primer nombre mayor que el segundo\n");
   else printf("Primer nombre menor que el segundo\n");
}
```

## *Cadenas de Caracteres: Funciones Relacionadas*

- Algunas funciones de conversión (**`stdlib.h`**)

**`double atof(char *s)`**

✓ Convierte la cadena **`s`** a **`double`**.

**`int atoi(char *s)`**

✓ Convierte la cadena **`s`** a entero.

**`long atol(char *s)`**

✓ Convierte la cadena **`s`** a **`long int`**.

**`char *itoa(int valor,char *s,int base)`**

✓ Convierte un entero a cadena.

✓ Almacena el resultado en **`s.`**

✓ **`base`** es un número entre 2 y 36 y es la base en la que queremos que esté el resultado.

✓ Devuelve un puntero a la cadena resultante.

**`int sprintf(char *s,const char *f [,argumento,... ])`**

✓ Escribe en **`s`** la cadena de caracteres especificada en **`f`** (formato).

✓ El resto de parámetros es exactamente igual que en `printf()`.



## *Estructuras: Definición*

- Una **estructura** es un grupo de información relacionada que se usa para almacenar datos acerca de un tema o actividad.
- Las estructuras están divididas en **CAMPOS**.
- Un campo es una variable de un tipo determinado.

SINTAXIS:

```
struct <nombre> {    /* Define una estructura de nombre <nombre>. */
     <campo1>;       /* Los campos se definen como una variable,
     <campo2>; ...       indicando  el tipo del campo y su nombre. */
     <campoN>;
}
struct <nombre>            /* Declara variables de tipo struct <nombre>.*/
     <var1>, ..., <varN>;  /* La estructura debe haber sido definida. */

struct <nombre>{      /* Mezcla de las dos declaraciones anteriores. */
     <campos>;
} <var1>,...,<varN>;
```

## *Estructuras: Acceso a los Campos*

- Para acceder a cada campo se utiliza el operador punto (.)

EJEMPLO:

```
struct libro {
   char titulo[20]; /* título del libro */
   char autor[20]; /* autor del libro */
   int num; /* número del libro */
};
struct libro lib;

lib.titulo /* accede al título */
lib.titulo[i] /* accede al caracter (i+1) del título */
lib.autor  /* accede al autor */
lib.num    /* accede al número del libro */
```



## *Estructuras: Ejemplo*

```
#include <stdio.h>
#include <string.h>
void main() {
  struct c {
    float coste;
    int codigo;
    char nombre[50];
  } componente;
  componente.coste=1.5;
  componente.codigo=32201;
  strcpy(componente.nombre,"resistencia");
    . . . .
  printf("Nombre %s, código %d, coste %f\n",
         componente.nombre,componente.codigo,
         componente.coste);
}
```

## *Estructuras: Inicialización y Asignación*

- Los campos de una estructura se pueden inicializar a la vez que se declara la variable (igual que los arrays):
  EJEMPLO:

```
struct libro l={"Rebeca", "D. du Maurier", 124};
```

- Se puede asignar una estructura a otra utilizando el operador de asignación '=' (con arrays no se puede hacer):
  EJEMPLO:

```
struct persona {
    char nombre[20];
    char direc[30];
    int edad;
    float altura, peso;
};
struct persona p1;
struct persona p2;
p1=p2;
```

- La operación de comparación '**==**' no está permitida.



## *Estructuras: Ejemplo*

```
/* Escribe cuántos números son mayores que un valor. */
#include <stdio.h>
void main() {
  int i,N=0; float valor;
  struct lista{    /* estructura con un entero y un
     int n;          array. El entero indica cuántos
     float a[20];    elementos válidos tiene el array */
  } l;
  printf("Cuántos elementos? "); scanf("%d", &l.n);
  for(i=0;i<l.n;i++) {  /* lectura de los valores */
     printf("Valor %d: ",i+1); scanf("%f",&l.a[i]);
  }
  printf("Valor? "); scanf("%f", &valor);
  for(i=0;i<l.n;i++) if(l.a[i]>valor) N++;
  printf("Hay %d números mayores que %f\n", N,valor);
}
```

## *Estructuras: Punteros a Estructuras*

- Un modo alternativo de acceder a una estructura es a través de **PUNTEROS**.

EJEMPLO:

✓ Se declara un puntero a la estructura persona:

```
struct persona *ptrPers;
```

Sólo se reserva memoria para el puntero, NO para la estructura.

✓ Se declara una variable de tipo persona:

```
struct persona p;
```

Reserva memoria para la estructura completa.

✓ Se asigna al puntero **ptrPers** la dirección de **p**:

```
ptrPers=&p;
```

✓ Ahora se accede a los campos utilizando el **operador flecha ->**

```
ptrPers->edad=45;  /* ó (*ptrPers).edad=45 */
ptrPers->altura=1.75 /* (*ptrPers).altura=1.75 */
```



## *Estructuras en Funciones: Paso de Estructuras por Valor*

✓ A la función se le pasa una COPIA de la estructura.

✓ La función sólo puede modificar la copia y no la variable estructura original.

✓ Cuando acaba la función los cambios NO se reflejan en la estructura original.

EJEMPLO:

```
void VisualizarDatos(struct persona x) {
   printf("Nombre: %s\n", x.nombre);
   printf("Dirección: %s\n", x.direc);
   printf("La edad es: %d\n", x.edad);
   printf("La altura es: %f\n", x.altura);
   printf("El peso es: %f\n", x.peso);
}
void main() {
   struct persona p1;
   ...
   VisualizarDatos(p1); /* se pasa una copia de p1 */
}
```

## *Funciones que Devuelven Estructuras*

```
struct persona LeerPersona() {
  struct persona temp; /* estructura local */
  printf("Nombre? ");gets(temp.nombre);
  printf("Dirección? "); gets(temp.direc);
  printf("Edad? "); scanf("%d", &temp.edad);
  printf("Altura? "); scanf("%f", &temp.altura);
  printf("Peso? "); scanf("%f", &temp.peso);
  return(temp); /* devuelve una copia de temp */
}
void main() {
  struct persona p1;
  p1=LeerPersona(); /* guarda la estructura
                       devuelta por la función en p1 */
  VisualizarDatos(p1); /* Visualiza la
                          estructura p1 */
}
```



## *Estructuras en Funciones: Paso de Estructuras por Referencia*

- ✓ Se pasa la dirección de la estructrura.
- ✓ Para acceder a cada campo de la estructura se utiliza el **operador flecha ->**

  **variable -> campo** ó **(*variable).campo**

EJEMPLO:

```
void LeerPersona2(struct persona *ptr) {
   printf("Datos? ");
   gets(ptr->nombre); /* todos los valores se guardan
   gets(ptr->direc);   directamente en la estructura
   scanf("%d",&(ptr->edad));   original p1 a través
   scanf("%f",&(ptr->altura));   del puntero ptr */
   scanf("%f",&(ptr->peso));
}
void main() {
   struct persona p1;
   LeerPersona2(&p1); /* pasa la direccion de p1 */
   VisualizarDatos(p1);
}
```

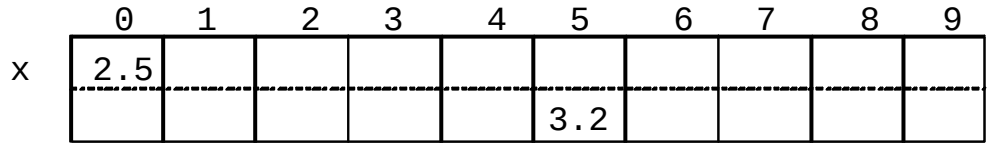
## *Arrays de Estructuras*

- **Se pueden construir arrays en los que cada elemento sea una estructura.**

```
struct complex {
    double real, imag;
};
struct complex x[10]; /*array unidimensional
   de 10 elementos, cada uno de ellos es una
   estructura complex */
```

- Acceso a cada campo:

```
x[0].real=2.5
x[5].imag=3.2;
```






## *Arrays de Estructuras: Ejemplo*

```
#include <stdio.h>
void main() { /*Listado de aprobados y suspensos*/
  struct alumno {
      char nombre[30];
      float notaT,notaP;
  };
  struct alumno A[50]; /* array de alumnos */
  int N;
  printf("Cuántos alumnos? "); scanf("%d",&N);
  ..........
  for(i=0;i<N;i++) {
     printf("%s\t%s\n", A[i].nombre,
          ((0.6*A[i].notaT+0.4*A[i].notaP)<5.0)?
           "Suspenso":"Aprobado");
  }
}
```

## *Creación de Tipos de Datos: Definición y Ejemplos*

Se crean con la sentencia **typedef**.

```
typedef  definicion_tipo  nombre_tipo;
```

EJEMPLOS:

```
typedef int tablero[8][8]; /* define un nuevo tipo de
                              datos llamado tablero que
                              es una matriz 8x8 */
tablero t1,t2;   /* declaración de variables de ese
                    tipo */
typedef struct {   /* Se crea un tipo llamado
   double real;       complejo */
   double imag;
} complejo;
complejo c; /* Se declara una variable de ese tipo */
            /* no hay que poner struct */
complejo x[10]; /* array unidimensional de 10 complejos */
```



## *Creación de Tipos de Datos: Ejemplo*

```
#include <stdio.h>
typedef struct {
   double real, imag;
} complejo;
void pedir_complejo(complejo *c);
void escribir_complejo (complejo c);
complejo suma_complejo(complejo c1, complejo c2);
void main(){
  complejo a,b,s;
  pedir_complejo(&a); pedir_complejo(&b);
  s=suma_complejo(a,b);
  escribir_complejo(s);
}
void pedir_complejo(complejo *c) {
  printf("Parte real? "); scanf("%lf", &(c->real));
  printf("Parte imaginaria? "); scanf("%lf", &(c->imag));
}
......
```

## *Algoritmos Básicos de BÚSQUEDA*

- **Objetivo**: Buscar un elemento determinado en una lista de objetos.
- **Características**: Para cualquier **tipo de datos.**
  - Para cada tipo de datos deberá usarse un comparador adecuado.
- **Supondremos**:
  - **Número de Elementos: `N`**
  - **Tipo de los Elementos: cualquiera**
  - **Array (vector) con los `N` Elementos: `V`**
    Desde el Elemento **`V[0]`** hasta el **`V[N-1]`**
  - **Elemento a buscar: `x`**



## *Búsqueda Secuencial o Lineal*

- Se usa cuando no existe información adicional sobre los datos.
- Se hace un recorrido secuencial a través del array hasta que se encuentra el elemento, o bien hasta que se llega al final (si el elemento no se ha encontrado):

```
ind=0;
while(ind<N && V[ind]!=x) {
  ind++;
}
if (ind>=N)
  puts("No encontrado.");
else
  printf("Encontrado en la pos:
  %d\n.",ind);
```

## *Búsqueda Binaria*

- El array debe estar ordenado.
- Se divide sucesivamente el array en dos partes hasta que se encuentra el elemento o bien hasta que no queda ningún sitio donde buscar:

```
izda=0; dcha=N-1; encontrado=0;
while(izda<=dcha && !encontrado) {
  m=(izda+dcha)/2;
  if (V[m]==x)
    encontrado=1;
  else if(V[m]<x)
         izda=m+1;
       else
         dcha=m-1;
}
```



## *Búsqueda de Subcadenas*

- OBJETIVO: Localizar una subcadena de longitud **M** (array **P**) dentro de otra cadena de longitud **N** (array **T**), con **0<M<=N**

```
i=0; j=0;
while(i<N && j<M) {
  if (T[i]==P[j]) {
     i++;
     j++;
  }
  else {
     i=i-j+1;
     j=0;
  }
}
if (j==M)
  printf("Cadena incluida a partir de la \
          posición %d\n", i-M);
else
  printf("Cadena no incluida.");
```

## *Algoritmos Simples de* ORDENACIÓN

- **Objetivo**: Ordenar una lista de objetos de acuerdo a un criterio ⇨ Debe existir una **Relación de Orden** "<" entre los objetos.
- **Características**: Para cualquier **tipo de datos** y tanto para ordenación **ascendente** como **descendente**.
- **Tipos**:
  - **Interna**: En memoria principal del ordenador. Con acceso directo a los objetos a ordenar.
  - **Externa**: En memoria secundaria (disco). Usada cuando el número de objetos es muy grande.
- **Aplicaciones Principales:**
  - **BUSCAR** un registro (o dato) concreto en una lista (o fichero):
    - Para:
      - **CONSULTAR** sus datos o ver si existe.
      - Buscar elementos **DUPLICADOS**.
    - La **Búsqueda Binaria** es más eficiente que la **Búsqueda Secuencial**, pero requiere que la lista esté ordenada.
  - **EMPAREJAR** datos de dos o más listas: Si las listas están ordenadas el emparejamiento es más rápido.



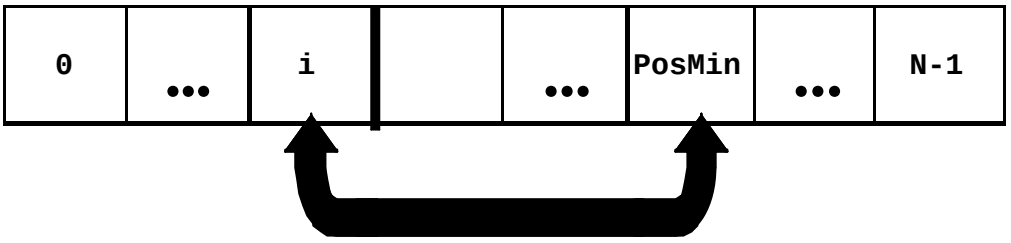
## *Algoritmos Simples de Ordenación*

- Supondremos:
  - **Número de Elementos: `N`**
  - **Tipo de los Elementos: `int`**
  - **Array (vector) con los `N` Elementos: `V`**
    Desde el Elemento **`V[0]`** hasta el **`V[N-1]`**
  - **Función para Intercambiar Elementos: `Swap()`**

```
void Swap (int *x, int *y){
   int temp=*x;
   *x=*y;
   *y=temp;
 }
```

## *Ordenación por "Selección"*

- En la iteración de la posición `i` se supone que están ordenados las posiciones hasta `i–1`.
- Para cada posición `i`: **Seleccionar** el elemento más pequeño (en la posición `PosMin`) entre las posiciones `i` y `N–1`, y se intercambia con el elemento que está en la posición `i`.



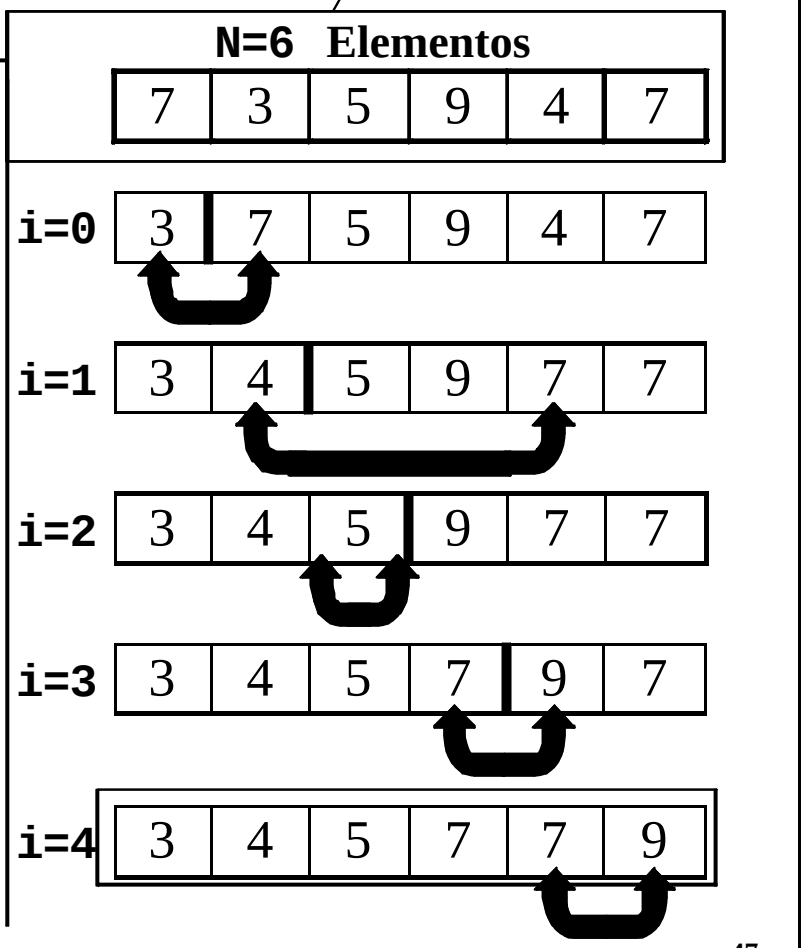



## *Ordenación por "Selección"*

```
/* N-1 iteraciones: Desde i=0 hasta i=N-2 */
for (i=0;i<N-1;i++){
  PosMin=i; /* Posición del elemento mínimo */

  /* Buscar posición del elemento mínimo entre
     las posiciones i+1 y N-1: PosMin */
  for (j=i+1;j<N;j++)
    if (V[j]<V[PosMin])
       PosMin=j;

  /* Intercambiar el elemento mínimo con el
     situado en la posición i (pueden
     coincidir) */
  Swap(&V[PosMin],&V[i]);
}
```

## *Ordenación por "Selección"*

```
for (i=0;i<N-1;i++){
  PosMin=i;

  for (j=i+1;j<N;j++)
    if (V[j]<V[PosMin])
       PosMin=j;

  Swap(&V[PosMin],
       &V[i]);
}
```






## *Modificaciones al de "Selección"*

- Seleccionar el mayor elemento (en vez del menor) en cada iteración:
  - Si suponemos que tenemos una función que devuelve la POSICIÓN del mayor valor en un array `V`, que suponemos de `N` elementos:

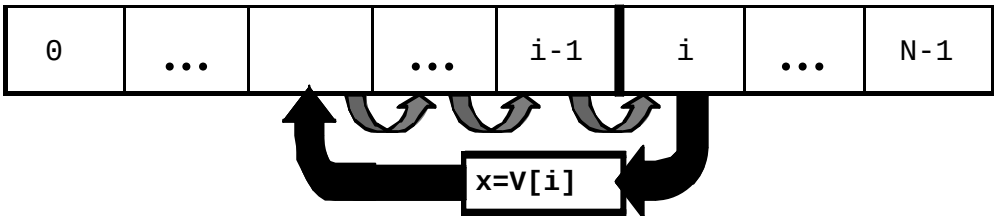
    ```
    unsigned PosMayor(int V[], unsigned N);
    ```

  - Así, la ordenación puede quedar como:

    ```
    for (i=N;i>1;i--)
      Swap ( &(V[i-1]),&(V[ PosMayor(V,i) ]) );
    ```

## *Ordenación por "Inserción"*

- En cada iteración `i`, se suponen ordenados los primeros `i-1` elementos y hay que **insertar** el elemento `i`-ésimo en esos `i-1` primeros elementos.
- Comenzar con el segundo elemento (`i=1`).
- Problema: ¿Cómo buscar donde realizar la inserción del elemento?
- Para buscar donde realizar la inserción, lo más simple es la **búsqueda secuencial**:
  - Desplazar hacia la derecha los elementos mayores al que queremos insertar.
  - Al encontrar un elemento menor o igual que el que queremos insertar, NO lo desplazamos y en la siguiente posición insertamos el elemento.



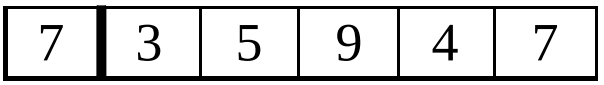



## *Ordenación por "Inserción"*

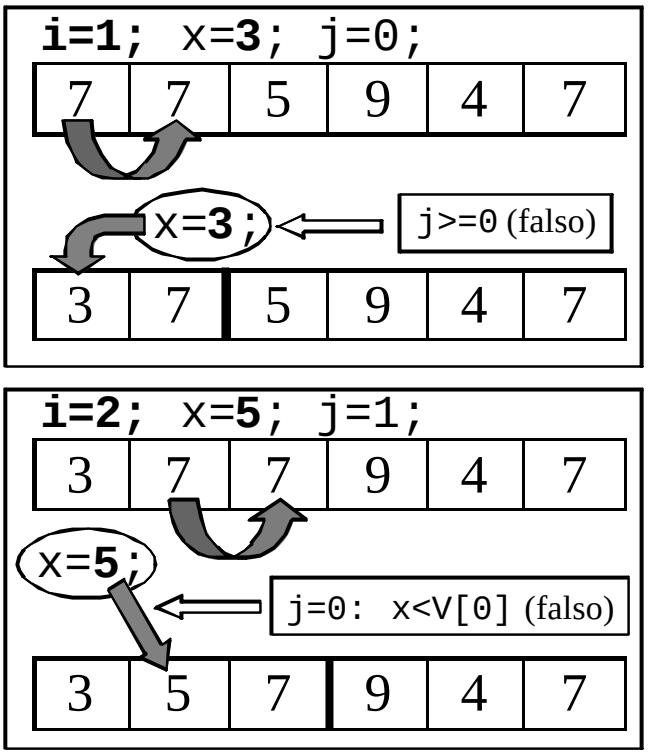
```
/* N-1 iteraciones: Desde i=1 hasta i=N-1 */
for (i=1;i<N;i++){
  x=V[i]; /* x es el elemento a INSERTAR */
  j=i-1;  /* j es el índice con el que buscar */
 /* Buscar posición donde se deba INSERTAR x */
  while (j>=0 && x<V[j]){
    /* Se desplazan a la derecha */
    /* los elementos mayores a x */
    V[j+1]=V[j];
    j--; /* Puede valer -1 (si x es el menor) */
  }

  V[j+1]=x; /* INSERTAR elemento x */
}
```

## Ordenación por "Inserción"

**N=6 Elementos**

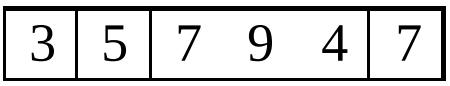



```
for (i=1;i<N;i++){
  x=V[i];
  j=i-1;

  while (j>=0 && x<V[j]){
    V[j+1]=V[j];
    j--;
  }

  V[j+1]=x;
}
```

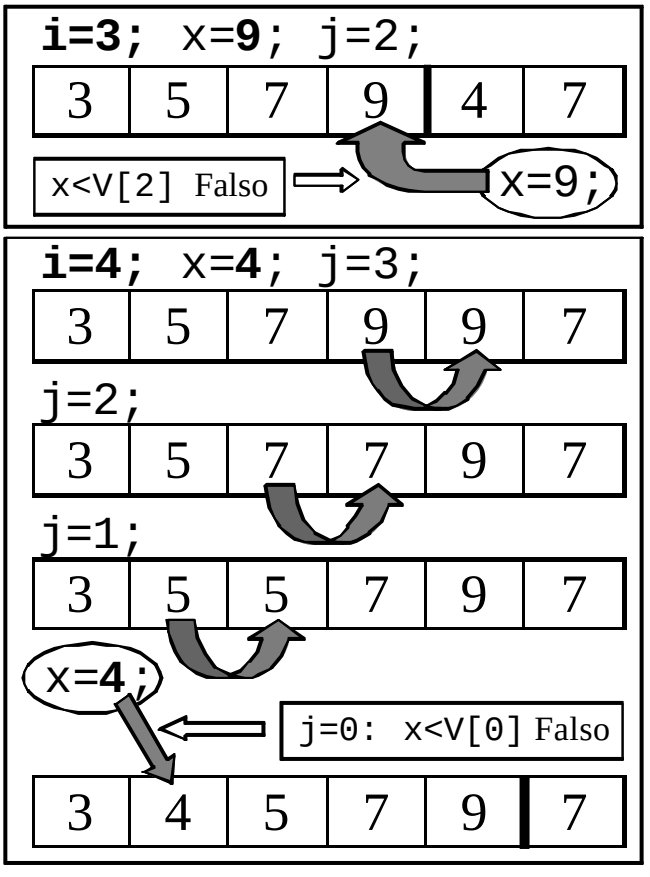





## Ordenación por "Inserción"

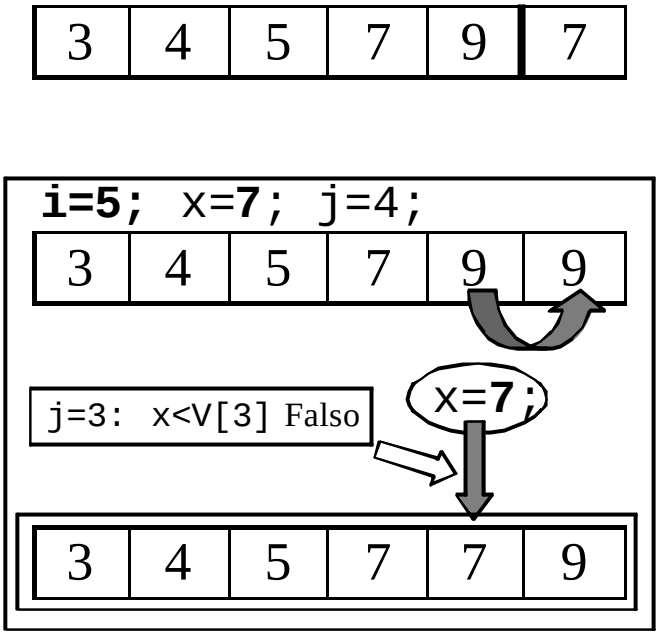



```
for (i=1;i<N;i++){
  x=V[i];
  j=i-1;

  while (j>=0 && x<V[j]){
    V[j+1]=V[j];
    j--;
  }

  V[j+1]=x;
}
```



## *Ordenación por "Inserción"*

```
for (i=1;i<N;i++){
  x=V[i];
  j=i-1;

  while (j>=0 && x<V[j]){
    V[j+1]=V[j];
    j--;
  }

  V[j+1]=x;
}
```






## *Modificaciones al de "Inserción"*

- Puede optimizarse el método para buscar la posición donde se insertará cada elemento, utilizando una **búsqueda binaria:**
  - La mejora **es** en el número de comparaciones y NO en la cantidad de transferencias realizadas.
  - Esta búsqueda binaria es aconsejable si el número de elementos **N** es suficientemente grande.

## *Modificaciones al de "Inserción"*

```
for (i=1;i<N;i++){
  x=V[i];   /* x es el elemento a INSERTAR */
  Izda=0;   /* Izda apunta al primer elemento */
  Dcha=i-1; /* Dcha al último elemento ordenado */

  /* Buscar posición (binaria) */
  while (Izda<=Dcha) {
    m=(Izda+Dcha) / 2;
    if (x<V[m]) Dcha=m-1; /* Puede valer -1 */
    else        Izda=m+1;
  }

  for (j=i-1;j>=Izda;j++) /* Desplazar elementos */
    V[j+1]=V[j];

  V[Izda]=x; /* INSERTAR elemento x */
}
```



## *Ordenación por "Intercambio"*
## *(Algoritmo"Burbuja", "BubbleSort")*

- Comparar los elementos de dos en dos e **intercambiarlos** si están en orden incorrecto.
- Así, los elementos más grandes suben a sus posiciones correctas antes que los más pequeños.
- Iteraciones (del bucle principal):

  **1ª.** Compara elementos en posiciones: (0,1), (1,2), (2,3), ... y (N–2, N–1). Elemento más grande se coloca en la última posición, la N–1, que es su posición correcta.

  **2ª.** Compara posiciones: (0,1), (1,2), (2,3), ... y (N–3, N–2). Siguiente elemento más grande se coloca en la posición N–2, que es su posición correcta.

  • • •

  **Última iteración**: Sólo compara los elementos en las posiciones (0,1), intercambiándolos si procede.

## *Ordenación por "Intercambio"*
### *(Algoritmo"Burbuja", "BubbleSort")*

```
/* Hay que dar N-1 iteraciones:
   Desde i=1 hasta i=N-1 */
for (i=1;i<N;i++)

  /* Comparar elemento j-ésimo con el siguiente,
     empezando desde el primer elemento (j=0) y
     terminando con el último elemento no ordenado */

  for (j=0;j<N-i;j++)

    /* Si los elementos en las posiciones j y j+1
       están desordenados: Intercambiarlos. */
    if (V[j]>V[j+1])
      Swap(&V[j],&V[j+1]);
```

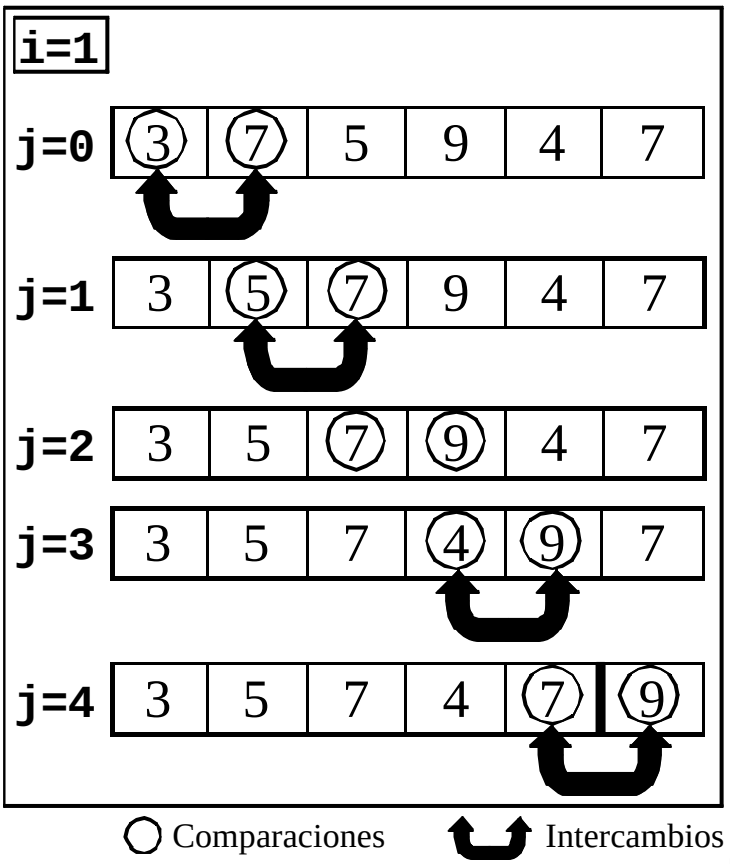


## *Ordenación por "Intercambio": "Burbuja"*

**N=6 Elementos**

| 7 | 3 | 5 | 9 | 4 | 7 |
|---|---|---|---|---|---|

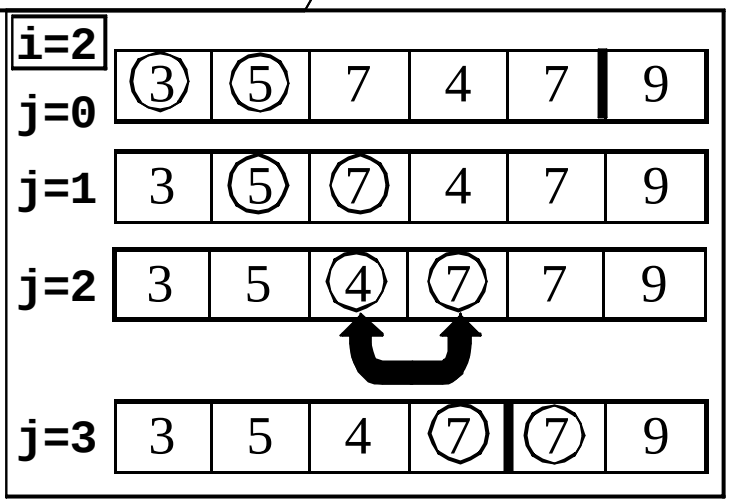
```
for (i=1;i<N;i++)

 for (j=0;j<N-i;j++)

   if (V[j]>V[j+1])
     Swap(&V[j],&V[j+1]);
```

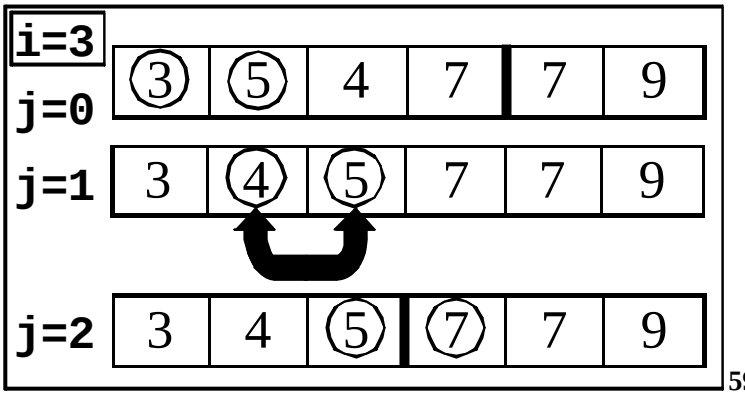


## *Ordenación por "Intercambio": "Burbuja"*

| 3 | 5 | 7 | 4 | 7 | 9 |
|---|---|---|---|---|---|

```
for (i=1;i<N;i++)

 for (j=0;j<N-i;j++)

  if (V[j]>V[j+1])
    Swap(&V[j],&V[j+1]);
```



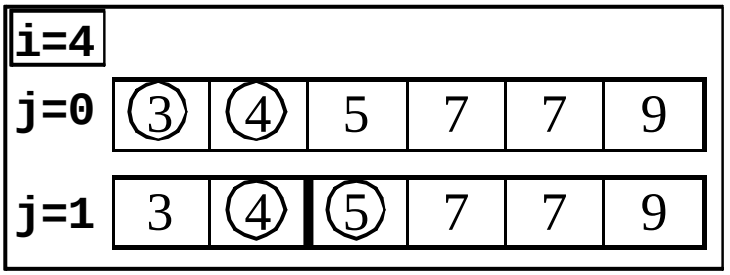



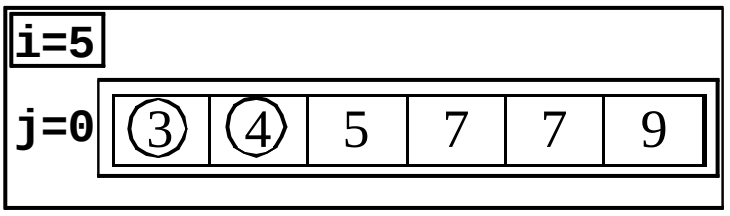



## *Ordenación por "Intercambio": "Burbuja"*

| 3 | 4 | 5 | 7 | 7 | 9 |
|---|---|---|---|---|---|

```
for (i=1;i<N;i++)

  for (j=0;j<N-i;j++)

    if (V[j]>V[j+1])
       Swap(&V[j],&V[j+1]);
```



## *Modificaciones al "Burbuja"*

- Que los números pequeños se "hundan" antes que los grandes:

```
for(i=1;i<=N;i++)

  for(j=N-1;j>=i;i--)

    if(V[j-1]>V[j])
      Swap(&V[j-1],&V[j]);
```

- Detener la ordenación en cuanto se consiga:

```
Vuelta=1;
do {
  NoOrdenado=0;

  for(j=0;j<N-Vuelta;j++)
    if(V[j]>V[j+1]){
       Swap(&V[j],&V[j+1]);
       NoOrdenado=1;
    }
  Vuelta++;
}while(NoOrdenado);
```