# Facultad de Informática

# PROGRAMACIÓN I
## Sesiones teóricas

Departamento de Ciencias de la Computación
y Tecnologías de la Información

# ESTRUCTURAS SIMPLES DE DATOS
## Tipos Estructurados

### Clasificación de tipos de datos

- **Según sean sus constituyentes**
  - Simples
  - Compuestos
    - Estructurados
    - No estructurados
- **Según quien los haya definido**
  - Predeterminados
  - Definidos por el usuario
- **Según la relación entre los valores de referencia**
  - Ordinales
  - No ordinales

# ESTRUCTURAS SIMPLES DE DATOS

## Tipos Estructurados

- Las estructuras de datos pueden ser:
  - **Estáticas:** estructuras de datos cuyo **tamaño** se conoce en tiempo de **compilación**
    - **ARRAYS, REGISTROS**
  - **Dinámicas:** estructuras de datos cuyo **tamaño cambia** en tiempo de **ejecución** (punteros)
    - **FICHEROS, LISTAS, PILAS, COLAS**
    - **ÁRBOLES, GRAFOS**

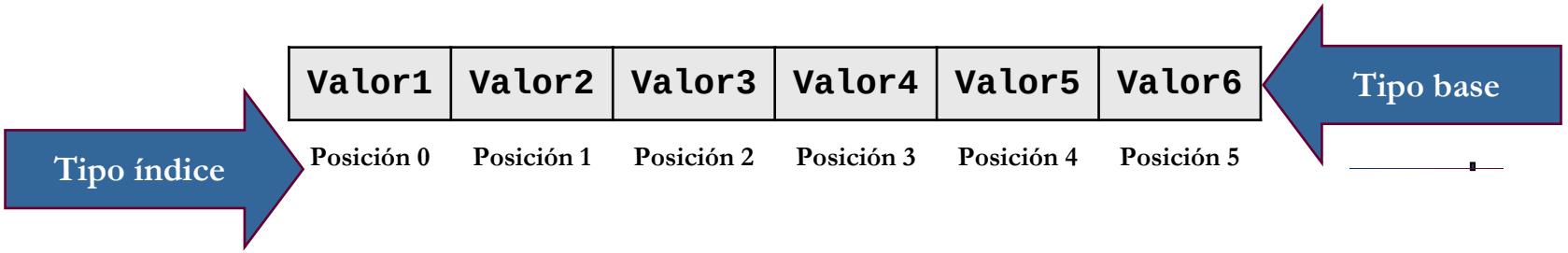
# ARRAYS

# ESTRUCTURAS SIMPLES DE DATOS

## ARRAYS. Tipo de dato Array

- Un **array** es un tipo de datos **estructurado** que está formado por un **número finito** de elementos todos del **mismo tipo** que están situados en **posiciones consecutivas** en memoria y que se asocian con un **único identificador**

  - Todos los elementos comparten un tipo común: **el tipo base** del array
  - La posición que cada elemento ocupa en el grupo de datos se indica mediante **el tipo índice**



# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS

Para declarar un array, debemos especificar el tipo de los elementos del array y el número de elementos:

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

Los elementos pueden ser de cualquier tipo; la longitud del array puede ser cualquier expresión constante (entera).

Usar una macro para definir la longitud de una array es algo habitual:

```
#define N 10
...
int a [N];
```

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS

- Para acceder a un elemento del array, se escribe el nombre del array seguido de un valor entero entre corchetes. Esto se conoce como subíndice o indexación del array.
- Los elementos de un array de longitud **n** se indexan de **0** a **n - 1**.
- Si **a** es una matriz de longitud 10, sus elementos están designados por un [0], un [1], ..., un [9]:
- Las expresiones de la forma a[i] son valores del tipo correspondiente, por lo que se pueden usar de la misma manera que las variables ordinarias:

```
a[0] = 1;
printf ("% d \ n", a[5]);
++a[i];
```

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS

- Los arrays son una estructura de **acceso aleatorio**, se caracterizan porque se puede **acceder** de **forma directa**, y con el **mismo esfuerzo**, a cada uno de sus elementos utilizando el **nombre de la variable** de tipo array **seguida de un índice**

**Vector1[4] indicaría el elemento situado en la posición 5(0-5) del array denominado Vector1, es decir, Valor5**

**Vector1**

| Valor1 | Valor2 | Valor3 | Valor4 | Valor5 | Valor6 |
|---|---|---|---|---|---|
| Posición 0 | Posición 1 | Posición 2 | Posición 3 | Posición 4 | Posición 5 |

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS

- Los arrays se caracterizan porque se puede **acceder**, de **forma directa**, y con el **mismo esfuerzo**, a cada uno de sus elementos utilizando el **nombre de la variable** de tipo array **seguida de un índice**

**¿Cómo se almacenarían las notas de 80 alumnos?**

**Se consumiría demasiado tiempo en declararlas y en acceder a cada una de ellas!!!!**

**Vector1**

| Valor1 | Valor2 | Valor3 | Valor4 | Valor5 | Valor6 |
|---|---|---|---|---|---|
| Posición 0 | Posición 1 | Posición 2 | Posición 3 | Posición 4 | Posición 5 |

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS

- Los arrays se caracterizan porque se puede **acceder**, de **forma directa**, y con el **mismo esfuerzo**, a cada uno de sus elementos utilizando el **nombre de la variable** de tipo array **seguida de un índice**

**¿Cómo se almacenarían las notas de 80 alumnos?**

**en un array de 80 elementos….todos de tipo float**

**VectorNotas**

| Nota1 | Nota2 | Nota3 | ..... | ..... | ..... | No |
|---|---|---|---|---|---|---|
| Posición 0 | Posición 1 | Posición 2 | Posición 3 | Posición 4 | ..... | Posición 79 |

Tipo índice: 0..79

Tipo base: float

# ESTRUCTURAS SIMPLES DE DATOS
# ARRAYS

Muchos programas contienen bucles cuyo trabajo consiste en realizar alguna operación en cada elemento de una matriz.

Ejemplos de operaciones típicas en una matriz de longitud N:

```
for (i = 0; i <N; i ++)
    a[i] = 0;

for (i = 0; i <N; i ++)
    scanf ("% d", &a[i]); / * lectura * /

for (i = 0; i <N; i ++)
    suma + = a[i]; / * suma los elementos de a * /
```

# ESTRUCTURAS SIMPLES DE DATOS
# ARRAYS

El compilador* de C no realiza la comprobación de los límites de los subíndices; si un subíndice se sale del rango, el comportamiento del programa no está definido.

- **Un error común**: olvidar que una matriz con **n** elementos se **indexa** de **0** a **n - 1**, no de 1 a n:

```
int a[10], i;
for (i = 1; i <= 10; i ++)
a[i] = 0;
```

- Un subíndice de la matriz puede ser cualquier expresión entera:

```
a[i+j * 10] = 0;
```

**Depende del diseño e implementación*

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Ejemplo: InvertirNúmeros

**EJEMPLO:** **InvertirNúmeros.c**

```
/ * Invierte una serie de números * /
#include <stdio.h>
#define N 10
int main (void)
    {
       int a[N], i;
       printf ("Introduzca % d números:", N);
       for (i = 0; i <N; i ++)
       scanf ("% d", & a[i]);
       printf ("en el orden inverso son:");
       for (i = N-1; i> = 0; i--)
       printf ("% d", a[i]);
       printf ("\ n");
       return 0;
    }
```

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Inicializar

Una matriz, como cualquier otra variable, puede recibir un valor inicial en el momento en que se declara. La forma más común es una lista de expresiones constantes entre llaves y separadas por comas:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Si la inicialización es más corta que la matriz, a los elementos restantes de la matriz se les asigna el valor 0:

```
int a [10] = {1, 2, 3, 4, 5, 6};
/* el valor inicial de a es{1, 2, 3, 4, 5, 6, 0, 0, 0, 0} */
```

Usando esta característica, podemos inicializar fácilmente una matriz con todas las posiciones a cero: `int a[10] ={0};`

En la inicialización **longitud de la matriz se puede omitir**:

```
int a[ ] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Inicializar

A menudo solo unos pocos elementos de una matriz deben inicializarse explícitamente, los otros elementos pueden tener valores por defecto.

```
int a[15] ={0, 0, 29, 0, 0, 0, 0, 0, 0, 7, 0, 0, 0, 0, 48};
```

En el compilador C99 se pueden usar la definición siguiente.

```
int a[15] = { [2]=29, [9]=7, [14]=48 };
```

OJO! [2]=29 no es necesario poner a[2]=29

Los restantes se inicializan a cero.

El orden de inicialización no se tiene en cuenta.

```
int a[15] = { [14]=48, [9]=7, [2]=29};
```

OJO! La siguiente matriz tendrá 24 elementos:

```
intb[ ] = {[5]=10, [23]=13, [11]=36, [15]=29};
```

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS: función sizeof

El función **sizeof** permite (como se ha visto) determinar el tamaño de un array (en bytes).

Si **a** es una matriz de 10 enteros, entonces **sizeof**(a) suele ser 40 (asumiendo que cada entero requiere cuatro bytes).

Al dividir el tamaño de la matriz por el tamaño del elemento, se obtiene la longitud de la matriz:

```
sizeof (a) / sizeof (a[0])
```

Se puede escribir:

```
for(i=0; i <sizeof (a)/sizeof (a[0]); i ++)
    a[i] = 0;
```

*Algunos compiladores producen un mensaje de advertencia para la expresión*

```
i <sizeof (a)/sizeof (a[0]).
```

¿Por qué?

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Mas de una dimensión

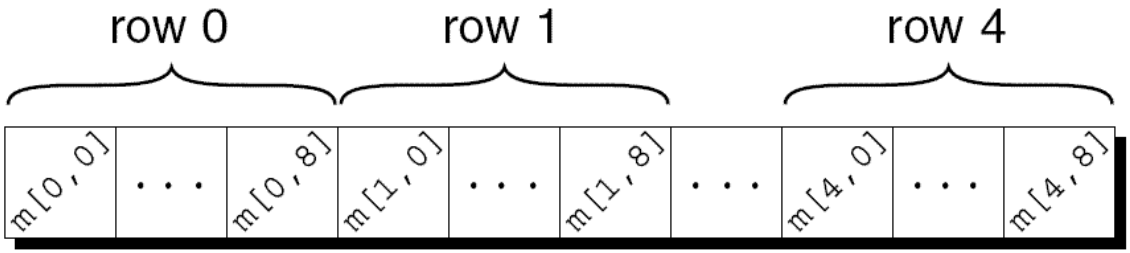
Una matriz puede tener cualquier número de dimensiones.

La siguiente declaración crea una matriz bidimensional:

```
int m[5][9];
```

m tiene 5 filas y 9 columnas. Ambas filas y columnas están indexadas desde 0:

C almacena las matrices como valores consecutivos en orden de fila mayor, esto es, con la fila 0 primero, luego la fila 1, y así sucesivamente.



# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Mas de una dimensión

Consideremos como inicializar una matriz I para usarla como una matriz de identidad.

```
…
#define N 10
…
…
double I[N][N];
int fila, col;

for (fila = 0; fila < N; fila++)
  for (col = 0; col < N; col++)
    if (fila == col)
      I[fila][col] = 1.0;
    else
      I[fila][col] = 0.0;
 …
```

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Mas de una dimensión

Podemos inicializar una matriz bidimensional anidando inicializadores unidimensionales:

```
int m[5][9]={{1, 1, 1, 1, 1, 0, 1, 1, 1},
             {0, 1, 0, 1, 0, 1, 0, 1, 0},
             {0, 1, 0, 1, 1, 0, 0, 1, 0},
             {1, 1, 0, 1, 0, 0, 0, 1, 0},
             {1, 1, 0, 1, 0, 0, 1, 1, 1}};
```

Para matrices de dimensiones superiores se construye de manera similar.

Al igual que una dimensión podemos escribir:

```
int m [5] [9] = {{1, 1, 1, 1, 1, 0, 1, 1, 1},
                 {0, 1, 0, 1, 0, 1, 0, 1, 0},
                 {0, 1, 0, 1, 1, 0, 0, 1, 0}};
```

Las dos últimas filas contendrán ceros.

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Mas de una dimensión

Cómo crear una matriz de identidad 2 × 2:

```
doble  I[2] [2] = {[0] [0]=1.0,  [1] [1]=1.0};
```

Como hemos visto, todos los elementos para los que no se especifica ningún valor se establecerán en cero.

Una matriz se puede hacer "constante" al comenzar su declaración con la palabra const:

```
const char n[ ] ={'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A B C D E F'};
```

Una matriz que se haya declarado constante no debe ser modificada por el programa.

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Arrays en funciones

Cuando un parámetro de función es una matriz unidimensional, la longitud de la matriz se puede dejar sin especificar:

```
int funcion (int a[ ]) / * sin longitud especificada * /
```

C no proporciona ninguna forma fácil para que una función determine la longitud de una matriz que se le pasa. En su lugar, tendremos que proporcionar la longitud como un argumento.

```
int suma_array(int a[ ], int n)
  {
    int i,sum=0;
    for (i = 0; i < n; i++)
    sum += a[i];
    return sum;
  }
```

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Arrays en funciones.

Cuando se llama suma_array, el primer argumento puede ser el nombre de la matriz, y el segundo su longitud:

```
# define LEN 100
int main ( )
{
    int b[LEN], total;
     ...
     total = suma_array (b, LEN);
     ...
  }
```

No ponemos corchetes después del nombre de la matriz cuando se lo pasamos a una función:

```
total = sum_array (b[ ], LEN); /*** INCORRECTO ***/
```

# ESTRUCTURAS SIMPLES DE DATOS
# ARRAYS. Arrays en funciones. Ejemplo

Si un parámetro es una matriz multidimensional, solo se puede omitir la longitud de la primera dimensión.

Si revisamos suma_array para que **a** sea una matriz bidimensional, debemos especificar el número de columnas que tiene la matriz **a**:

```
# define LEN 10

int suma_array_2 (int a[ ][LEN], int n)
 {
     int i, j, suma = 0;
     for (i = 0; i <n; i ++)
     for (j = 0; j <LEN; j ++)
     suma+= a[i][j];
     return suma;
}
```

# ESTRUCTURAS SIMPLES DE DATOS
# ARRAYS. Arrays en funciones.

C99 permite el uso de matrices de longitud variable como parámetros.

Permite escribir:

```
int suma_array (int a[ ], int n)
  {
     ...
  }
```

Aquí, no hay un enlace directo entre **n** y la longitud de la matriz **a**.

Aunque el cuerpo de la función trata **n** como la longitud de a, la longitud real de la matriz podría ser mayor o menor que n.

Posibilidades!:

```
int funcion(int a[ ])
int funcion(int n, int a[n])
int funcion(int a[ ], int n)
 int funcion(int, int [ ])
```

# ESTRUCTURAS SIMPLES DE DATOS
## ARRAYS. Arrays en funciones.

En el caso de mas dimensiones:

```
int fun_array2D(int n, int m, int a[n][m]);
int fun_array2D (int n, int m, int a[*][*]);
int fun_array2D (int n, int m, int a[ ][m]);
int fun_array2D (int n, int m, int a[ ][*]);
```

# Ejemplo

El programa **max_min.c** lee 10 números en una matriz, lo pasa a la función *max_min()* e imprimirá los resultados:

```
Entrada 10 números: 34 82 49 102 7 94 23 11 50 31
Máximo: 102
Mínimo: 7
```

# ✓Ejemplo

## maxmin.c

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

void max_min(int a[], int n, int *max, int *min);

int main(void)
{
  int b[N], i, Max, Min;

  printf("Entroduzca %d números: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &b[i]);
  max_min(b, N, &Max, &Min);
  printf(“El máximo es: %d\n", Max);
  printf(“El mínimo es: %d\n", Min);

  return 0;
}
```

# ✓Ejemplo

```
void max_min(int a[], int n, int *max, int *min)
{
  int i;

  *max = *min = a[0];
  for (i = 1; i < n; i++) {
    if (a[i] > *max)
      *max = a[i];
    else if (a[i] < *min)
      *min = a[i];
  }
}
```

# Cadena de caracteres

# STRINGS

# ESTRUCTURAS SIMPLES DE DATOS

## Cadenas

- En lenguaje C, una cadena de caracteres se representa como un **array** de caracteres.
- El valor de una cadena se escribe entre dobles comillas:

  "Hola mundo"

- Todas las cadenas en C terminan con el carácter nulo ´\0´

  "Hola" se corresponde con: |´H´|´o´|´l´|´a´|´\0´|

- Una cadena de caracteres se puede definir como:

```
char cadena[ ]="Hola";
char * cadena = "Hola"; /* definición a partir de punteros */
char cadena[5]="Hola";  ¿por qué 5?
```

# ESTRUCTURAS SIMPLES DE DATOS

## Cadenas

Se puede acceder a cualquier elemento de una cadena como un array.

char cadena[ ]="Hola mundo";

si hacemos

printf("%c", cadena[4]);

la salida sería: ´ a´

La escritura y lectura de cadenas se puede realizar con **printf( )** y **scanf( )**, usando el especificador de conversión: **%s**

printf("%s\n", cadena); / solo se pone el nombre */

**Pero:** scanf( ) deja de buscar cuando encuentra un **espacio en blanco**

Si hacemos *scanf("%s", cadena)* e introducimos: Hola mundo, cadena "recoge" solo Hola

# ESTRUCTURAS SIMPLES DE DATOS

## Cadenas

El lenguaje C no permite utilizar funciones específicas de lectura y escritura de strings:

**gets**( )

**puts**( )

- gets( ) lee una línea completa, incluyendo espacios en blanco, hasta que encuentra un salto de línea. Devuelve NULL si ha habido errores.

- puts( ) escribe la cadena de caracteres junto con el carácter retorno de línea

```
printf("Esto se ve como una línea\n");
puts("Esto se ve como una línea también");
```

# ESTRUCTURAS SIMPLES DE DATOS

## Cadenas

El lenguaje C también tiene definidas funciones específicas de lectura y escritura de caracteres:

**getchar**( )

**putchar**( )

- getchar( ) lee caracteres hasta que encuentra el final de fichero EOF (EOF un macro, un #define, en stdio.h)

- putchar ( ) escribe el carácter del argumento

```
printf("Esto se ve como una línea\n");
puts("Esto se ve como una línea también");
```

# ESTRUCTURAS SIMPLES DE DATOS

## Ejemplo 1

```
/* el programa cuenta el número de veces que aparece la letra asignada a "buscamos" en una cadena */
#include <stdio.h>
 int main()
 {
    char cadena[256];
    int veces=0, i=0;
    char control='\0' ;
    char buscamos='a';
    printf("Introduzca una cadena de caracteres (máximo 256):");
    scanf("%s", cadena);
    while (cadena[i] != control){
        if (cadena[i]==buscamos) veces++;
        i++;
   }
   printf("Ha escrito %c , %d veces \n", buscamos, veces);
   return 0;
}
```

# ESTRUCTURAS SIMPLES DE DATOS
## Ejemplo 2

**Long_mensaje1.c**

```
/* Determina la longitude de un mensaje */

#include <stdio.h>

int main(void)
{
  char ch;
  int longitud = 0;

  printf("Entroduzca un mensaje: ");
  ch = getchar();
  while (ch != '\n') {
    longitud++;
    ch = getchar();
  }
  printf(“Su mensaje tiene %d characteres.\n", longitud);

  return 0;
}
```

# ESTRUCTURAS SIMPLES DE DATOS
## Ejemplo 3

**Long_mensaje2.c**

```
/* Otra forma de implementación */

#include <stdio.h>

int main()
{
  int longitud = 0;

  printf("Entroduzca el mensaje: ");
  while (getchar()!= '\n') {
    longitud++;
  }
  printf(“Su mensaje tiene %d characteres.\n", longitud);

  return 0;
}
```

# ✓**Ejemplos.** Conversión de mayúsculas a minúsculas

```
#include<stdio.h>

void MinToMay(char string[]); /* cabecera de la funcion */

void main()
{
   char cadena[80];

   printf("Introduce una cadena:");
   gets(cadena);
   MinToMay(cadena); /* Llama a la funcion */
   printf ("En Mayusculas: %s\n", cadena);

}
```

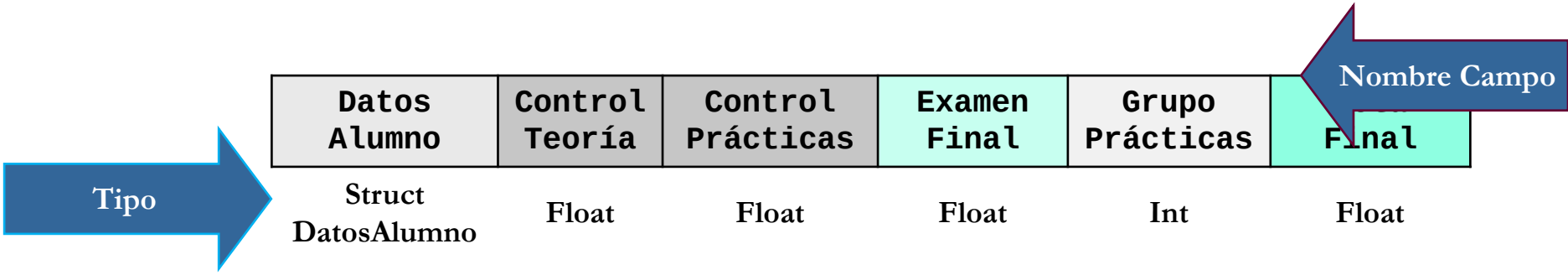
# ✓Ejemplos. Conversión de mayúsculas  aminúsculas

```
/* Función que convierte minúsculas a mayúsculas */

void MinToMay(char string[])
{
   int i=0;
   int desp='a'-'A';
   for (i=0;string[i]!='\0';++i)
   {
      if(string[i]>='a‘ && string[i]<='z')
      {
         string[i]=string[i]-desp;
      }
   }
}
```

# REGISTROS

# ESTRUCTURAS SIMPLES DE DATOS
## REGISTROS. STRUCT

- El STRUCT es un tipo de datos **estructurado** que está formado por un **número finito** de elementos que pueden ser de **distinto tipo** y que se asocian con un **único identificador**
  - Los distintos componentes de un **struct** (registro, estructura) se denominan habitualmente **CAMPOS**
  - En la **declaración** de un struct se deben especificar el **nombre** y el **tipo** de cada campo



# ESTRUCTURAS SIMPLES DE DATOS
## REGISTROS. STRUCT

Las propiedades de una estructura son diferentes de las de un array.

- No se requiere que los elementos de una estructura (sus miembros) tengan el mismo tipo.
- Los miembros de una estructura tienen nombres.
- Para seleccionar un miembro en particular, especificamos su nombre, no su posición.
- En algunos lenguajes, las estructuras se denominan registros y los miembros se conocen como “campos”.

# ESTRUCTURAS SIMPLES DE DATOS

## REGISTROS. STRUCT

Una estructura es una opción lógica para almacenar una colección de elementos de datos relacionados.

Una declaración de dos variables de estructura que almacenan información sobre piezas en un almacén:

```
struct{
        int numero;
        char nombre[LongNbre];
        int disponibles;
        } part1, part2;
```

Para inicializar una estructura se siguen reglas similares a las de los arrays.

No se tienen que inicializar todos los campos.

Cualquier miembro "sobrante" recibe 0 como su valor inicial.

# ESTRUCTURAS SIMPLES DE DATOS
## STRUCT

Inicializamos:

```
struct{
        int numero;
        char nombre[LongNbre];
        int disponibles;
} part1 = {528, "Disco duro", 10},  part2 = {914, "Cable red", 5};
```

Para acceder a un miembro dentro de una estructura, primero escribimos el nombre de la estructura, luego un punto y luego el nombre del campo.

```
printf("numero de pieza: %d\n", part1.numero);
printf("nombre: %s\n", part1.nombre);
printf("unidades disponibles: %d\n", part1.disponibles);
```

# ESTRUCTURAS SIMPLES DE DATOS
## STRUCT

- Los campos de una estructura actúan como valores con sus respectivos tipos.
- Pueden aparecer en una asignación o como el operando en una expresión…:

```
part1.numero = 258;
part1.disponible ++;
```

El punto utilizado para acceder a un miembro de la estructura es en realidad un operador C. Tiene prioridad sobre casi todos los demás operadores.

Ejemplo:

```
scanf ("% d", & part1.disponible);
```

El . como operador tiene prioridad sobre el operador &, por lo que & se aplica a part1.disponible.

# ESTRUCTURAS SIMPLES DE DATOS
## STRUCT

Otra operación importante de las estructuras es la asignación:

part2 = part1;

El efecto de esta declaración es copiar part1.numero en part2.numero, part1.nombre en part2.nombre y así sucesivamente.

El operador **=** solo se puede utilizar con estructuras de tipos compatibles.

Dos estructuras declaradas al mismo tiempo (como part1 y part2) son compatibles. Si están definidas con el mismo tipo también ( *typedef* ).

Aparte de la asignación, C **no** proporciona operaciones en estructuras completas. En particular, los operadores **==** y **!=** No pueden utilizarse con estructuras

# STRUCT

Existen dos formas de nombrar una estructura:

- Declarar una "etiqueta de estructura", como hemos visto.
- Utilizar **typedef** para definir un nombre de tipo (*un tipo nuevo*).

La declaración de una estructura llamada **part** sería:

```
struct part {
  int numero;
  char nombre[LongNbre];
  int disponibles;
};
```

La etiqueta de **part** se puede utilizar para declarar tipo de variables:

```
part part1, part2;
```

# STRUCT

Las funciones pueden tener estructuras como argumentos y valores de retorno.

Una función con un argumento de estructura:

```
void print_part(struct part p)
  {
    printf("Part numero: %d\n", p.numero);
    printf("Part nombre: %s\n", p.nombre);
     printf(“Cantidad disponibles : %d\n", p.disponibles);
  }
```

Una llamada a la función sería: print_part(part1);

# STRUCT

Un función que devuelve una estructura sería

```
struct part pedido( int numero,  const char *nombre, int disponibles)
    {
     struct part p;
     p.numero= numero;
     strcpy(p.nombre, nombre); /* función C que mencionamos que copia strings */
     p.disponibles = disponibles;
     return p;
    }
```

Una llamada sería:

```
pedido(34, "disco", 345);
```

# STRUCT

Podemos incluir como “campo” de una estructura otra estructura:

```
struct estudiante{
  struct nombre_persona {
    nombre;
    primer_apellido;
    segundo_apellido;
   }
  int identificador, edad;
  char sexo;
} estudiante1, estudiante2;
```

Para acceder al nombre o a los apellidos se requieren dos aplicaciones del operador ‘.’ :

```
strcpy(studiante1.nombre, “Pepe");
```

# AGORITMOS de BÚSQUEDA y ORDENACIÓN

# Búsqueda Secuencial

- La **búsqueda secuencial** consiste en **comparar secuencialmente** el **elemento** buscado con los valores contenidos en las posiciones **min..max** del array hasta que, o bien se le **encuentre** en la **posición índice i**, o bien se llegue al **final** del array **sin encontrarlo**, concluyendo por tanto que no está en él

```
#include <stdio.h>
int main()
 {
    int A[20]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
    int buscado, n=20, indice=0;  /* si ponemos n=20, usaremos n-1 */
    printf("introduzca el dato a buscar");
    scanf("%d", &buscado);
    while (buscado!=A[indice] && indice< n)indice ++;
    if( indice<= n-1) printf("El número buscado %d está en la posición %d\n", buscado, indice+1);
   else printf("\n el dato buscado %d no está en el array\n",buscado);
   return 0;
}
```

# Búsqueda Secuencial con centinela

- Al algoritmo de **búsqueda secuencial** le añadimos en la **posición n+1** el **valor buscado** (**la centinela**), de esta forma en la condición de terminación del bucle se ahorra una comparación.

```
#include<stdio.h>
#define N  21   OJO /* array de n+1 elementos, el último índice 20  contendrá el valor del centinela */


int main()
 {
    int A[N]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
    int buscado, indice=0;
    printf("introduzca el dato a buscar\n");
    scanf("%d", &buscado);
    A[N-1]=buscado; / A[20]= buscado, es la "centinela“
    while (buscado!=A[indice])indice ++;
    if( indice< N-1) printf("El número buscado %d está en la posición %d\n", buscado, indice+1);
    else printf("\n el dato buscado %d no está en el array\n",buscado);
    return 0;
}
```

# Búsqueda Binaria

La **búsqueda binaria** se basa en la comparación del elemento buscado con el elemento central del array

Si el elemento central del array es el elemento buscado, la búsqueda ha finalizado

En caso contrario, según sea buscado menor o mayor que ese elemento central se buscará en la primera o segunda mitad del array, respectivamente

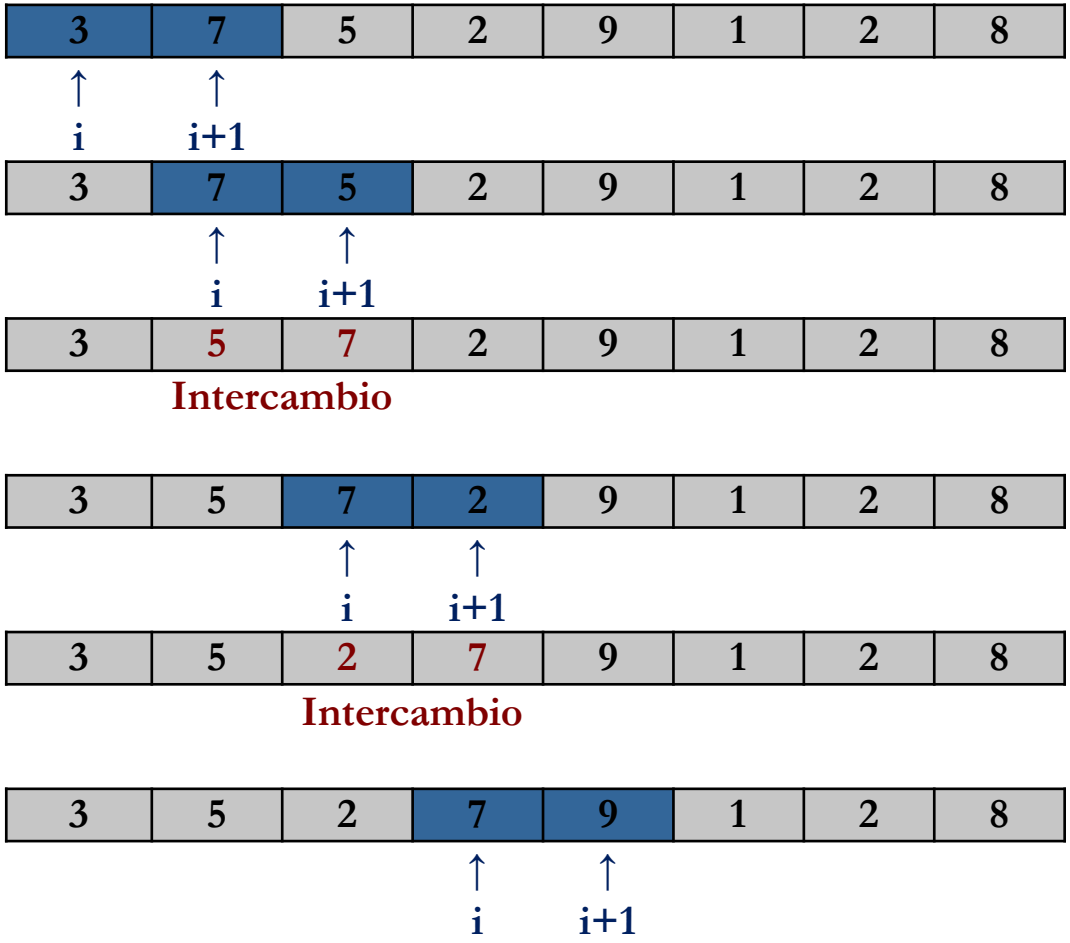
De nuevo se comparará el elemento buscado con el elemento central del subarray seleccionado y así sucesivamente hasta que o bien se encuentre el valor buscado o bien se pueda concluir que el elemento buscado no está en el array (porque el subarray de búsqueda ha quedado vacío)

# Búsqueda Binaria

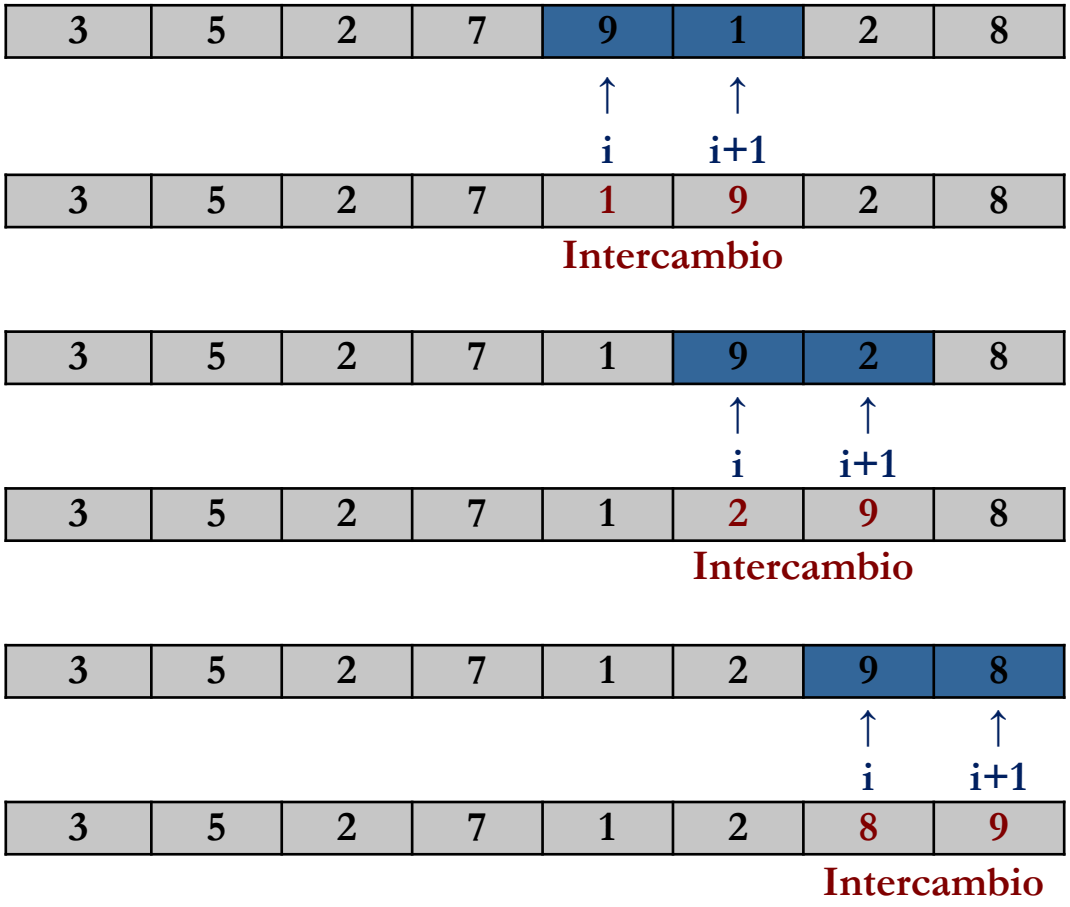
```
#include <stdio.h>
#include <stdlib.h>

int main()
 {
   int A[20]={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
   int izq,der,mit,buscado,n=20;

   printf("introduzca el dato a buscar");
   scanf("%d",&buscado);
   izq=0; der=n-1; mit=(izq+der)/2;   /* array de 0 a n-1 */
   while (izq<=der && buscado!=A[mit])
    {
      if (A[mit]<buscado) izq=mit+1; else der=mit-1;
      mit=(izq+der)/2;
    }
   if(izq<=der)printf("\nEl número %d está en la posición %d\n",buscado,mit+1);
   else printf("\n el número buscado %d no está en el array\n",buscado);
   return 0;
}
```

# Algoritmo de ordenación: Burbuja

- El método de ordenación de la **"burbuja"** se basa en la **comparación** e **intercambio** de **posiciones consecutivas**



# Burbuja


3 5 2 7 9 1 2 8
i
i+1
3 5 2 7 1 9 2 8
Intercambio
3 5 2 7 1 9 2 8
i
i+1
3 5 2 7 1 2 9 8
Intercambio
3 5 2 7 1 2 9 8
i
i+1
3 5 2 7 1 2 8 9
Intercambio

# Burbuja

```
#include <stdio.h>
#define TAM 6
int main()
{
  int lista[TAM]={12,10,5,6,1,3};     //Declaración e Inicialización del  array
  int i,j,temp=0;

  printf("La lista DESORDENADA es: \n");
  for (i=0;i<TAM;i++)
  printf("%3d",lista[i]);
   for (i=1;i<TAM;i++)
    {
     for (j=0;j<TAM-1;j++)
      {
       if (lista[j] > lista[j+1])
        {
          temp = lista[j];
          lista[j]=lista[j+1];  // Intercambio
          lista[j+1]=temp;
        }
      }
   }
  printf("\nLos valores ORDENADOS de lista son: \n");
  for(i=0;i<TAM;i++) printf("%3d",lista[i]);
  return 0;
}
```

# Algoritmo de Ordenación: Quicksort

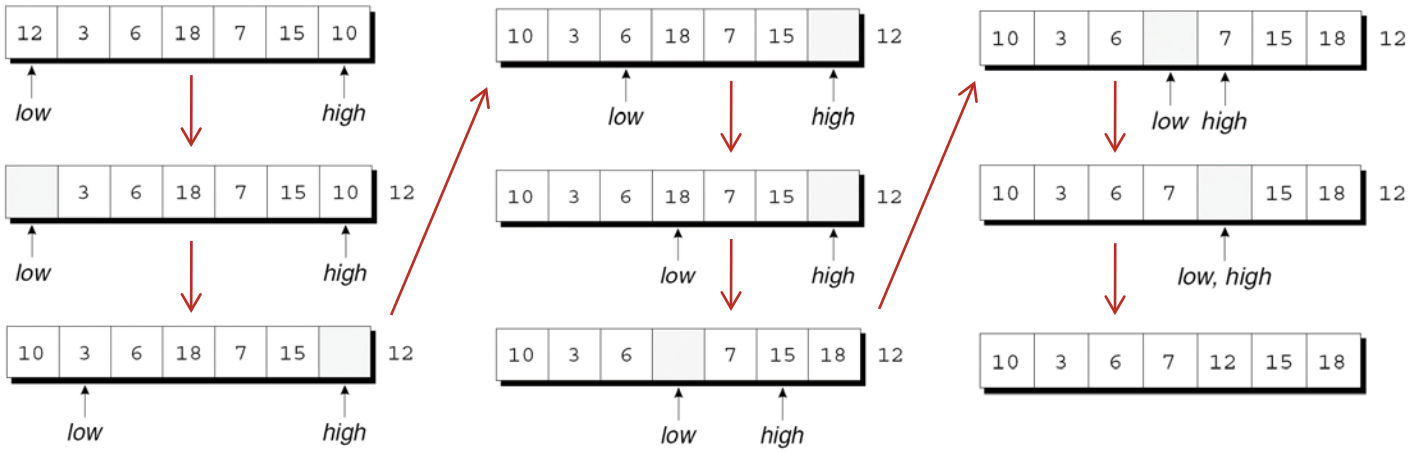
Algoritmo de ordenación rápida, Quicksort, se describe como:

1. Elija un elemento de matriz **e** (el "elemento de partición"), luego reorganice la matriz de modo que los elementos **1**, ..., **i-1** sean menores o iguales que **e**, el elemento i contiene e, y los elementos **i+1**, ..., **n** son mayores o iguales a e.

2. Ordene los elementos **1**,…, **i-1** utilizando Quicksort de forma recursiva.

3. Ordene los elementos **i+1**,…, **n** utilizando Quicksort de forma recursiva.

# Quicksort


12 3 6 18 7 15 10
low
high
12
3 6 18 7 15 10
low
high
12
10 3 6 18 7 15
low
high
10 3 6 18 7 15
low
high
12
10 3 6 18 7 15
low
high
12
10 3 6
7 15 18
low
high
12
10 3 6 7
15 18
low, high
12
10 3 6 7 12 15 18

# Quicksort

```
#include <stdio.h>

#define N 10

void quicksort(int a[], int low, int high);
int split(int a[], int low, int high);

int main(void)
{
  int a[N], i;

  printf("Introduzca los números que desea ordenar: ", N);
  for (i = 0; i < N; i++)
    scanf("%d", &a[i]);
  quicksort(a, 0, N - 1);

  printf("Los números ordenados quedan: ");
  for (i = 0; i < N; i++)
    printf("%d ", a[i]);
  printf("\n");

  return 0;
}
```

# Quicksort

La función quicksort es:

```
void quicksort(int a[], int low, int high)
{
  int middle;

  if (low >= high) return;
  middle = split(a, low, high);
  quicksort(a, low, middle - 1);
  quicksort(a, middle + 1, high);
}
```