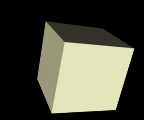
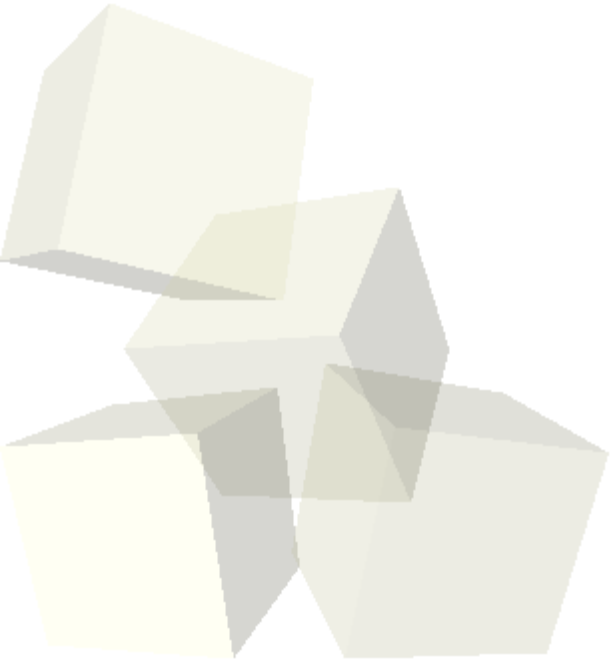


# Tema 4
# Tipos y estructuras de datos



Informática
Grado en Física
Universitat de València

Ariadna.Fuertes@uv.es
Francisco.Grimaldo@uv.es

# Guión del tema

- Introducción:
  - Concepto de dato estructurado
  - Clasificación de las agrupaciones de datos
  - Tipos de agrupaciones
- Estructuras de datos contiguas:
  - vectores,
  - matrices,
  - cadenas de caracteres y,
  - estructuras (o registros)
- Punteros y estructuras de datos dinámicas.
  - Introducción a las estructuras dinámicas:

    listas enlazadas

# Introducción (1/3)

- Los tipos simples de datos: booleanos, caracteres, enteros y reales.
- En general, la información tratada por el ordenador irá agrupada de una forma más o menos coherente en estructuras especiales, compuestas por datos simples. A este tipo de agrupaciones las denominaremos tipos de datos estructurados o tipos compuestos.
- Los tipos de datos estructurados o tipos compuestos se distinguen por los elementos que las componen y por las operaciones que se pueden realizar con ellas o entre sus elementos.
- En ocasiones interesará manejar las estructuras como elementos únicos, y otras nos interesará manejar los elementos que componen los tipos de datos estructurados como elementos separados.

# Introducción (2/3)

- Clasificaciones de las agrupaciones de datos:
  - Según cómo se ubiquen en la memoria del ordenador: Agrupaciones contiguas/Agrupaciones enlazadas
    - Las agrupaciones contiguas son aquellas que ocupan posiciones sucesivas o contiguas de bits en memoria. La posición de un elemento dentro de la agrupación se puede calcular sumando al comienzo de la agrupación una cantidad determinada.
    - Las agrupaciones enlazadas son aquellas que tienen la información dispersa en la memoria, y la manera de acceder desde un elemento a otro es mediante la dirección en memoria dónde está el siguiente elemento.
  - Según los tipos de datos que contengan: Agrupaciones homogéneas/Agrupaciones heterogéneas
    - Las agrupaciones homogéneas son aquellas que guardan grupos de elementos iguales (todos los elementos guardados son enteros, reales,...)
    - Las agrupaciones heterogéneas son las que agrupan elementos (campos) de diferentes tipos.

# Introducción (3/3)

- Según si tienen un tamaño fijo de elementos o puede crecer: Agrupaciones estáticas/Agrupaciones dinámicas
  - Las agrupaciones estáticas son las que guardan un número predeterminado de elementos, que no puede ser modificado en el momento en que se ejecuta el programa. Hay que elegir el número cuando escribimos el programa, antes de compilarlo.
  - Las agrupaciones dinámicas son aquellas que pueden ir modificando el número de elementos que contienen a medida que va siendo necesario guardar más información.

- Tipos de agrupaciones de datos:
  - arrays
    - vectores o arrays unidimensionales,
    - matrices o arrays multidimensionales y
    - cadenas
  - registros o estructuras y
  - listas dinámicas.

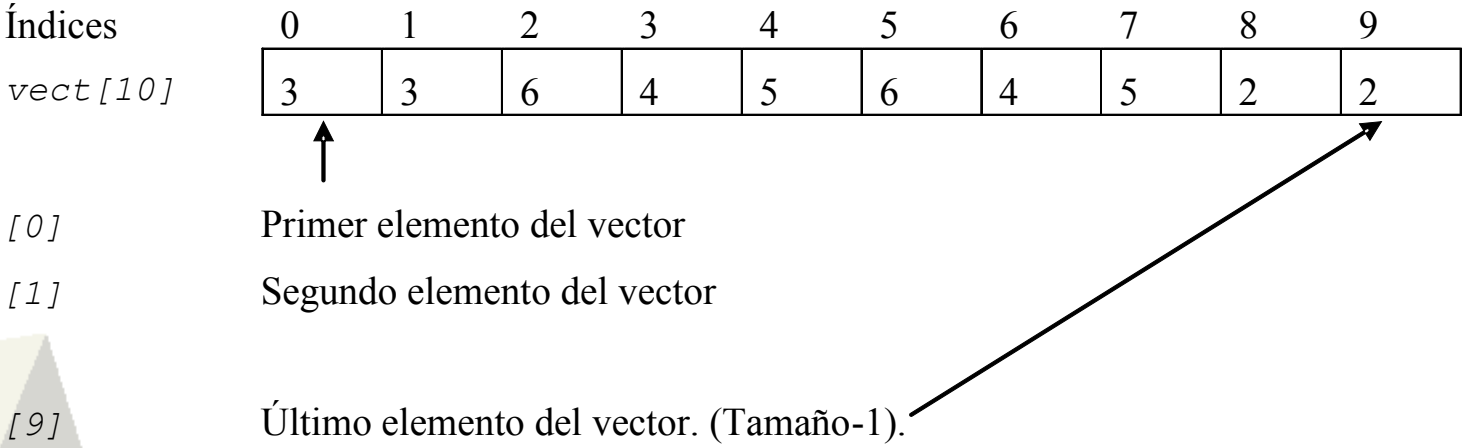
Las agrupaciones, en general, no son exclusivas de manera que podremos encontrar combinaciones de ellas si es necesario.

# Arrays (1/27)

- Los arrays son *agrupaciones homogéneas, contiguas y estáticas*. Los elementos que contienen son todos iguales y el número de elementos que podemos guardar se define cuando escribimos el programa.
- Podemos distinguir entre los siguientes tipos de arrays: vectores (arrays unidimensionales), matrices (arrays bidimensionales) y arrays multidimensionales.
- Vectores (o arrays unidimensionales)
- Son agrupaciones en las que cada elemento tienen asociado un índice (un entero), de manera que se puede acceder a cada uno de los elementos mediante la utilización de ese índice (operación de **indexación**). El índice indica la posición del elemento dentro del vector.
- Este tipo de datos estructurados sirven para agrupar variables de un mismo tipo con un único nombre.
- Supongamos que queremos declarar 10 variables de tipo entero. La única forma de hacerlo hasta ahora sería declararlos como variables individuales. La otra forma de hacerlo es utilizando un vector.

# Arrays: Vectores (2/27)

- La declaración de un vector la realizaremos de la siguiente manera: Nombre[Tamaño] : Tipo
- Donde Tipo es el tipo de los datos guardados en el vector, Nombre el nombre que le vamos a dar al vector y Tamaño el número de elementos que es capaz de guardar el vector.
- Los índices comienzan en 0 y terminan en Tamaño – 1



# Arrays: Vectores - operaciones (3/27)

**Operaciones**:

- **Asignación**
- Se da un valor determinado a algún elemento del vector. La manera de hacerlo es mediante el nombre del vector y el índice que queremos asignar:

  Nombre[indice] ← Valor

- **Ejemplo**: Para asignar el valor 4 al índice (o posición del vector) ocho:

  vect [8] ← 4

- NO se pueden realizar asignaciones entre vectores. NUNCA se debe realizar la siguiente asignación. Sean: a[10], b[10]: enteros

  a ← b ¡¡¡ERROR!!!

# Arrays: Vectores - operaciones (3/27)

- **Recorrido**
- Se pasa por los elementos del vector para realizar una tarea concreta en cada elemento.
- Recorrido completo

```
Desde i ← 0 hasta Tamaño-1 hacer
    Procesar (Nombre [i])
    i ← i+1
Fin_desde
```

- Recorrido parcial

```
i ← 0
Mientras (Nombre[i] cumpla una cierta condición) hacer
    Procesar (Nombre [i])
    i ← i+1
Fin_mientras
```

- Para facilitar la realización de bucles y evitar posibles errores en el recorrido de los vectores se recomienda que el número de elementos que posee un vector TAM se defina previamente como una constante.

# Arrays: Vectores - operaciones (4/27)

- **Ejemplo**: Realizar una función que calcule la media de los elementos contenidos en un vector

```
Funcion MediaVector ( vect[TAM]: entero) : real
Variables
    i : entero
    med : real
Inicio
    med ← 0
    Desde i ← 0 hasta TAM-1 hacer
        med ← med + vect[i]
        i ← i + 1
    Fin_desde

    med ← med/TAM
    MediaVector ← med
Fin_Funcion
```

# Arrays: Vectores - operaciones (5/27)

- **Iniciación**
- En la declaración del vector sólo se reserva espacio. NO se pone ningún valor en el vector.
- Para hacer la iniciación de todos y cada uno de los elementos del vector, habrá que recorrerlo y asignar un valor (o pedir al usuario que de valor a cada uno de los elementos).

- **Ejemplo**: Supongamos que queremos pedir al usuario que nos dé todos los valores del vector:

```
Desde i ← 0 hasta TAM-1 hacer
    leer( vect[i])
    i ← i + 1
Fin_desde
```

- **Ejemplo**: Supongamos que queremos poner todos los elementos del vector a cero:

```
Desde i ← 0 hasta TAM-1 hacer
    vect[i] ← 0
    i ← i + 1
Fin_desde
```

# Arrays: Vectores - operaciones (6/27)

- **Ejemplo**: Realizar un procedimiento que rellene los elementos de un vector con datos introducidos por el usuario

```
Procedimiento RellenaVector (ref vect[TAM]: entero)
Variables
    i : entero
Inicio
    Desde i ← 0 hasta TAM-1 hacer
        leer( vect[i] )
        i ← i + 1
    Fin_desde
Fin_Procedimiento
```

- PREGUNTA: ¿Podría existir la Función RellenaVector?

# Arrays: Vectores - operaciones (7/27)

- **Búsqueda en vectores**
- Para buscar un cierto elemento en un vector, en principio hay que ir recorriendo todos y cada uno de los elementos, en busca del que se busca. En general a esto se le llama *búsqueda secuencial*.

**Ejemplo**: *Realizar una función que diga si un cierto valor 'x' se encuentra o no en un vector 'vect'*:

```
Funcion Buscar1 ( vect[TAM]: entero, x : entero) : booleano
Variables
    i : entero
    encontrado : booleano
Inicio
    encontrado ← falso
    Desde i ← 0 hasta TAM-1 hacer
        Si vect[i] = x entonces
            encontrado ← verdadero
        Fin_si
        i ← i + 1
    Fin_desde
    Buscar1 ← encontrado
Fin_Funcion
```

# Arrays: Vectores - operaciones (8/27)

- Mejora del algoritmo anterior: “Salir de bucle cuando se encuentre el resultado” además de si se llega al final...

```
Funcion Buscar2 ( vect[TAM]: entero, x : entero) : booleano
Variables
    i : entero
    encontrado : booleano
Inicio
    i ← 0
    Mientras i < TAM AND vect[i] ≠ x hacer
        i ← i + 1
    Fin_Mientras
    Si i = TAM entonces
        encontrado ← falso
    Sino
        encontrado ← verdadero
    Buscar2 ← encontrado
Fin_Funcion
```

# Arrays: Vectores - operaciones (9/27)

- Podemos evitar la comparación de ‘i < TAM’ si estamos seguros de que el elemento ‘x’ está en el vector. Y podemos estar seguros si lo ponemos al final del vector. A esta forma de buscar se le llama *búsqueda secuencial con centinela*.

```
Funcion Buscar3 ( vect[TAM+1]: entero, x : entero) : booleano
Variables
    i : entero
    encontrado : booleano
Inicio
    vect[TAM] ← x
    i ← 0
    Mientras vect[i] ≠ x hacer
        i ← i + 1
    Fin_Mientras
    Si i = TAM entonces
        encontrado ← falso
    Sino
        encontrado ← verdadero
    Buscar3 ← encontrado
Fin_Funcion
```

# Arrays: Vectores - operaciones (10/27)

- Si el vector estuviese ordenado se puede aplicar una técnica especial de búsqueda llamada *búsqueda binaria o dicotómica:*

```
Funcion Buscar4 ( vect[TAM]: entero, x : entero) : booleano
Variables
    primero, ultimo, central : enteros
    encontrado : booleano
Inicio
    primero ← 0
    ultimo ← TAM-1
    central ← (primero+ultimo)/2
    Mientras primero < ultimo AND vect[central] ≠ x hacer
        Si x < vect[central] entonces
            ultimo ← central -1
        Sino
            primero ← central +1
        Fin_Si
        central ← (primero+ultimo)/2
    Fin_Mientras
    Si x = vec[central] entonces
        encontrado ← verdadero
    Sino
        encontrado ← falso
    Buscar4 ← encontrado
Fin_Funcion
```

*Búsqueda binaria o dicotómica:*

*1. Se mira el elemento central*

*2. Si es el elemento buscado terminar*

*3. Si no lo es:*

*3.a. Determinar en que 'zona' del vector está (por arriba o por debajo del central)*

*3.b. Repetir el proceso en la nueva zona.*

# Arrays: Vectores - operaciones (11/27)

- *El programa principal que llamará a las distintas funciones buscar (que podrían ser la 1 o la 2) sería:*

```
Variables
    v[TAM] , x : enteros
    si_esta : booleano
Inicio
    Escribir("Dame los datos del vector")
    RellenaVector( v )
    Escribir("Dame el numero a buscar")
    Leer( x )

    si_esta ← Buscar2( v, x )

    Si si_esta = verdadero entonces
        Escribir("Se ha encontrado el elemento")
    Sino
        Escribir("El elemento no está")
    Fin_Si
Fin
```

# Arrays: Vectores - operaciones (12/27)

- *Modificación a la función Buscar2 para que me devuelva en qué posición ha encontrado el elemento:*

```
Funcion BuscarConPos ( vect[TAM]: entero, x : entero) : entero
Variables
    i , encontrado: entero
Inicio
    i ← 0
    encontrado ← -1

    Mientras i < TAM AND vect[i] ≠ x hacer
        i ← i + 1
    Fin_Mientras

    Si i ≠ TAM entonces
        encontrado ← i
    Fin_Si
    BuscarConPos ← encontrado
Fin_Funcion
```

```
PROGRAMA PRINCIPAL
Variables
    v[TAM] , x , si_esta : enteros
Inicio
    Escribir("Dame los datos del vector")
    RellenaVector( v )
    Escribir("Dame el numero a buscar")
    Leer( x )

    si_esta ← BuscarConPos( v, x )

    Si si_esta >=0 entonces
       Escribir("Esta en la posicion", si_esta)
    Sino
       Escribir("El elemento no está, lo siento")
    Fin_Si
Fin
```

# Arrays: Matrices (13/27)

- Las *matrices* o *arrays bidimensionales* son agrupaciones similares a los vectores pero en las que cada elemento tiene asociados dos índices enteros, de manera que se puede acceder a cada uno de los elementos mediante la utilización de esos índices.

- La declaración de una matriz la realizaremos de la siguiente manera: Nombre[Tamaño1][Tamaño2] : Tipo

  Donde Tipo es el tipo de los datos guardados en la matriz, Nombre el nombre que le vamos a dar a la matriz y Tamaño1 y Tamaño2 son el número de filas y columnas que tiene.

# Arrays: Matrices - operaciones (14/27)

**Operaciones**:

- **Asignación**
- Se da un valor determinado a algún elemento de la matriz. La manera de hacerlo es mediante el nombre de la matriz y los índices (ahora son dos) que queremos asignar:

  Nombre[indice1][indice2] ← Valor

- **Ejemplo**: Para asignar el valor 4 a los índices (o posición de la matriz) ocho, cinco:

  vect [8][5] ← 4

- Al igual que para los vectores NO se pueden realizar asignaciones entre matrices.
- Por lo tanto, NUNCA una función podrá devolver un vector o matriz,.. en general, un array.

# Arrays: Matrices - operaciones (15/27)

- **Recorrido**
- Se pasa por los elementos de la matriz para realizar una tarea concreta en cada elemento.

- Recorrido completo

```
Desde1 i ← 0 hasta TAMX-1 hacer
    Desde2 j ← 0 hasta TAMY-1 hacer
        Procesar ( Nombre[i][j] )
        j ← j+1
    Fin_desde2
    i ← i+1
Fin_desde1
```

- Recorrido parcial

```
i ← 0
Mientras1 (Nombre[i][j] cumpla una cierta condición) hacer
    j ← 0
    Mientras2 (Nombre[i][j] cumpla una cierta condición) hacer
        Procesar (Nombre [i][j])
        j ← j+1
    Fin_mientras2
    i ← i+1
Fin_mientras1
```

# Arrays: Matrices - operaciones (16/27)

- **Ejemplo**: Realizar una función que recorra toda una matriz y calcule la media de todos los elementos contenidos en ella.

```
Funcion MediaMatriz ( mat[TAM1][TAM2]: entero) : real
Variables
    i , j : entero
    med : real
Inicio
    med ← 0
    Desde1 i ← 0 hasta TAM1-1 hacer
        Desde2 j ← 0 hasta TAM2-1 hacer
            med ← med + mat[i][j]
            j ← j + 1
        Fin_desde2
        i ← i + 1
    Fin_desde1

    med ← med / (TAM1*TAM2)
    MediaMatriz ← med

Fin_Funcion
```

# Arrays: Vectores - operaciones (17/27)

- **Iniciación**
- Para hacer la iniciación de todos y cada uno de los elementos de la matriz, habría que recorrerla y asignar o pedir al usuario que introdujese cada uno de los elementos. El resto de operaciones también son similares a la de los vectores pero accediendo con 2 índices.

- **Ejemplo**: Supongamos que queremos pedir al usuario que nos dé todos los valores de la matriz:

```
Desde1 i ← 0 hasta TAM1-1 hacer
    Desde2 j ← 0 hasta TAM2-1 hacer
        Leer( mat[i][j] )
        j ← j+1
    Fin_desde2
    i ← i+1
Fin_desde1
```

- **Ejemplo**: Supongamos que queremos poner todos los elementos de la matriz a cero:

```
Desde1 i ← 0 hasta TAM1-1 hacer
    Desde2 j ← 0 hasta TAM2-1 hacer
        mat[i][j] ← 0
        j ← j+1
    Fin_desde2
    i ← i+1
Fin_desde1
```

# Arrays Multidimensionales (18/27)

- Los *arrays multidimensionales* son agrupaciones similares a los vectores pero en las que cada elemento tiene asociados más de dos índices enteros, de manera que se puede acceder a cada uno de los elementos mediante la utilización de esos índices.

- La declaración de un array multidimensional la realizaremos de la siguiente manera:

  Nombre[Tamaño1][Tamaño2]...[TamañoN] : Tipo

  Donde Tipo es el tipo de los datos guardados en la matriz, Nombre el nombre que le vamos a dar al array y Tamaño1, Tamaño2, ..., TamañoN son el el tamaño de las diferentes dimensiones del array.

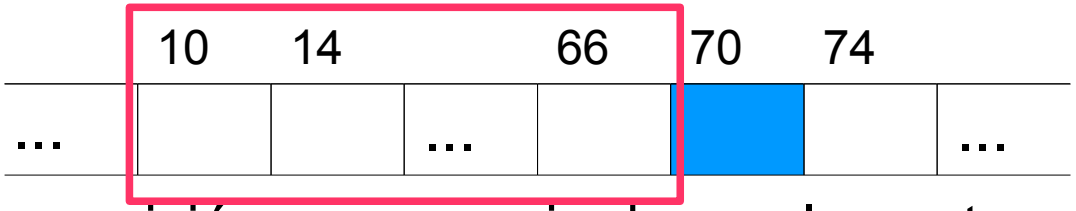
- Las operaciones que se puedan definir serán similares a las vistas en vectores o matrices pero ampliando el número de índices de forma adecuada, y recordando que para acceder a un elemento concreto hay que determinar el valor de todos y cada uno de los índices.

# Arrays: Consideraciones (19/27)

- Paso de arrays como parámetros
- Los arrays, como cualquier otro tipo de datos, se pueden pasar como parámetros a una función por valor o por referencia.
- Como una función no puede devolver un array (al no poderse asignar), la única forma de modificarlo será pasando el parámetro por referencia.

- Representación en memoria de un array
- Los elementos de un vector se almacenan en memoria de forma contigua, es decir, uno al lado del otro.

Ejemplo: Vect[15] : enteros
Si empieza en la dirección de memoria 10 y suponemos que un entero ocupa 4 bytes... entonces:




Por lo tanto, para calcular la posición en memoria de un elemento se utiliza la fórmula: *d = do + Ind * tamañode (Elemento)*
Donde: `do` es la dirección de memoria donde empieza el vector `Ind` es el índice del elemento del vector del que queremos saber su dirección, y *`Elemento`* es el tipo de elementos que contiene el vector.

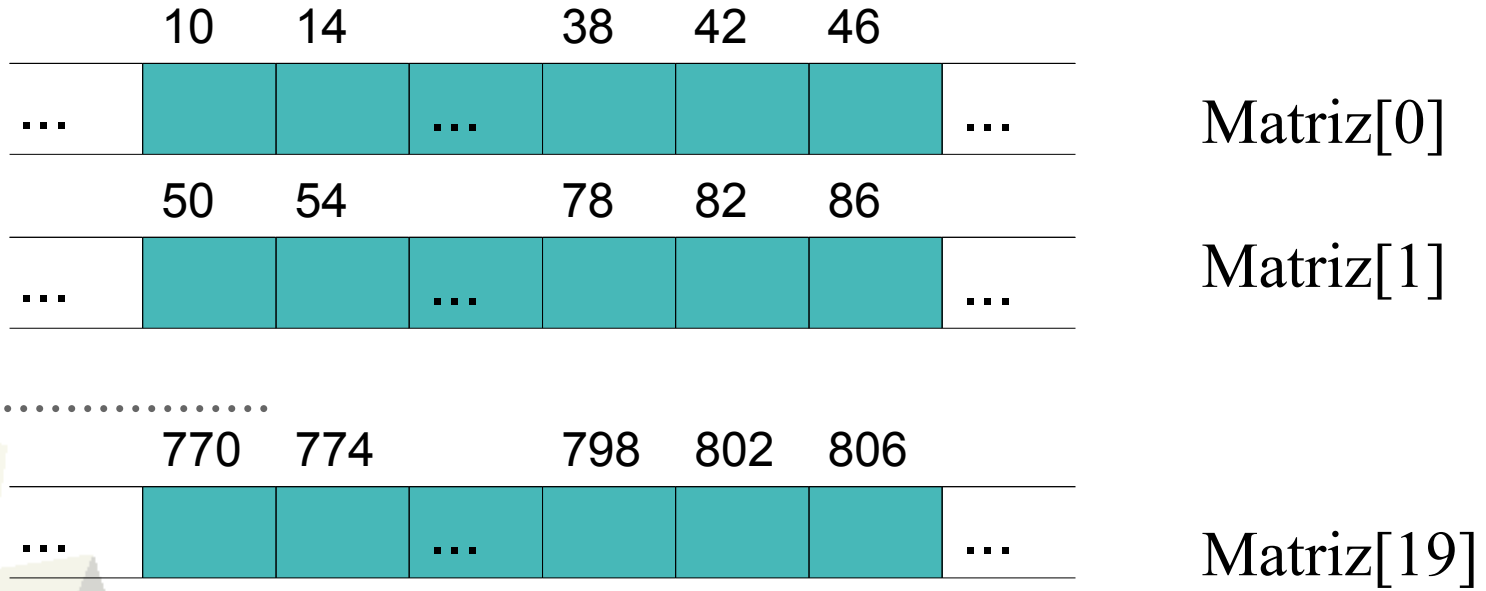
Nota: En C/C++ el *tamañode* es la función sizeof(x). Ej.: sizeof(int) es 4.

# Arrays: Consideraciones (20/27)

- Puesto que una matriz no es más que un vector de vectores, el almacenamiento en memoria es también contiguo:

- Ejemplo: Matiz[20][10] : enteros

Si empieza en la dirección de memoria 10 y *tamañode(entero)* es 4 bytes:




Para calcular la posición del elemento Matriz[19][1], lo que hay que hacer es primero resolver el primer índice (Matriz[19]), que tiene como elementos vectores enteros de 10 elementos, y después el segundo tomando como dirección inicial la que resulta del calculo anterior. Así, si el vector empieza por ejemplo en la posición 10, la fórmula será:

$d$ = 10 + 19 * tamañode(vect [10] de enteros) + 1 * tamañode(entero) = 10 + 760 + 4 = 774

# Arrays: Consideraciones (21/27)

- Uso de arrays con número de elementos variable
- El tamaño de un array ha de ser una constante, por lo tanto no es posible modificar su tamaño después de haber compilado el programa.
- Para solucionar esto existen dos posibilidades:
  - la primera es utilizar memoria dinámica (lo veremos al final del tema)
  - la segunda es declarar un array lo suficientemente grande y utilizar sólo una parte de este array.
- **Por ejemplo:** si vamos a realizar un programa para sumar vectores matemáticos y sabemos que en ningún caso vamos a tener vectores de un tamaño mayor de 10, podemos declarar todos los vectores de tamaño 10 y después utilizar sólo una parte del vector.
- Para ello necesitaremos utilizar también una variable entera extra para saber el tamaño actualmente utilizado por el vector, puesto que todas las operaciones se realizarán considerando el tamaño actual del vector y no el total.
- Los inconvenientes de este método son que estamos desperdiciando memoria y además que no siempre hay un límite conocido al tamaño del array.
- Ventajas: es muy fácil trabajar con el array.

# Arrays: Consideraciones (22/27)

*Ej.: Suma de dos vectores de un espacio vectorial real de dimensión n*

```
Proc. Suma (v1[TAM]: real, v2[TAM]: real, ref res[TAM]: real, n :entero)
Variables
      i : entero
Inicio
      Desde i ← 0 hasta n-1 hacer
            res[i] ← v1[i] + v2[i]
            i ← i + 1
      Fin_Desde
Fin_Procedimiento
```

```
Proc. Rellena( ref vect[TAM]: real, n: entero)
Variables
      i : entero
Inicio
      Desde i ← 0 hasta n-1 hacer
            Leer (vect[i])
            i ← i + 1
      Fin_Desde
Fin_Procedimiento
```

```
Proc. Mostrar ( vect[TAM]: real, n: entero)
Variables   i : entero
Inicio
      Desde i ← 0 hasta n-1 hacer
            Escribir (vect[i])
            i ← i + 1
      Fin_Desde
Fin_Procedimiento
```

```
PROGRAMA PRINCIPAL

Variables
     x[TAM] , y[TAM], z[TAM] : real
     dim : entero
Inicio
     Hacer
        Escribir("Dame la dimensión")
        Leer(dim)
     Mientras( dim < 1 OR dim > TAM)

     Escribir("Coord del vector 1er vector")
     Rellena ( x , dim )
     Escribir("Coord del vector 1er vector")
     Rellena ( y , dim )

     Suma ( x , y , z , dim )

     Mostrar ( z , dim )
Fin
```

# Arrays: Algoritmos (23/27)

- **Borrar un elemento en una posición dada**
- Para borrar un elemento lo único que tenemos que hacer es mover todos los elementos posteriores a él a una posición anterior y decir que tenemos un elemento menos.

```
Proc. Borra (ref v[TAM]: real, ref  n : entero, pos : entero)
Variables
      i : entero
Inicio
      Desde i ← pos hasta n-2 hacer
            v[i] ← v[i+1]
            i ← i + 1
      Fin_Desde
      n ←  n -1
Fin_Procedimiento
```

```
PROGRAMA PRINCIPAL

Variables
      vect[TAM] , dato : real
      dim,  i: entero
Inicio
      Rellena (vect , dim )
      Escribir("Dime el dato a borrar")
      Leer (dato)

      i ← Buscar ( vect , dim , dato )
      Si i >= 0 entonces
         Borrar ( vect , dim , i )
      Sino
         Escribir("Ese dato no esta")
      Fin_si
      Mostrar ( vect , dim )
Fin
```

# Arrays: Algoritmos (24/27)

- **Insertar un elemento en una posición dada**
- Para insertar un elemento en una posición, lo único que hay que hacer es mover una posición todos los elementos del vector desde el final hasta el sitio donde queremos insertar el dato y decir que hay uno más.

```
Proc. Insertar (ref v[TAM]: real, ref  n : entero, pos : entero, d : real)
Variables
      i : entero
Inicio
      Si n = 0 hacer
            v[n] ← d
            n ← n + 1
      Sino
            Desde i ← n-1 hasta pos hacer
                  v[i+1] ← v[i]
                  i ← i - 1
            Fin_Desde
            v[ pos ] ← d
            n ←  n +1
      Fin_si
Fin_Procedimiento
```

```
PROGRAMA PRINCIPAL

Variables
      vect[TAM] , dato : real
      dim,  i: entero
Inicio
      Rellena (vect , dim )
      Si dim < TAM entonces
         Hacer
            Escribir("Dame el dato y su posicion")
            Leer (dato , i )
         Mientras ( i >= dim OR i < 0)
         Insertar ( vect , dim , i , dato)
      Sino
         Escribir("Vector lleno, no cabe mas...")
      Fin_si
      Mostrar ( vect , dim )
Fin
```

# Arrays: Algoritmos (25/27)

- **Rellena un vector con elementos insertando de forma ordenada**
- Supongamos que queremos rellenar un vector de números reales de manera que estén ordenados, por ejemplo, de menor a mayor.

```
Proc. InsertarOrdenado (ref v[TAM]: real, ref  n : entero, d : real)
Variables
      pos : entero
Inicio
      pos ← BuscarPosicion (v , n, d)

      Si pos >= 0  entonces
            Insertar ( v , n , pos , d)
      Fin_si

Fin_Procedimiento
```

```
Func. BuscarPosicion (v[TAM]: real, n: entero, d: real) : entero
Variables
     i  : entero
Inicio
     i ←  0
     Mientras i < n AND d > v[i] hacer
        i ←  i + 1
     Fin_Mientras

     BuscarPosicion ←  i
Fin_Funcion
```

```
PROGRAMA PRINCIPAL

Variables
      vect[TAM] , dato : real
      i  : entero
Inicio
      i ← 0

      Mientras i < TAM  hacer
         Escribir("Dame el dato")
         Leer (dato )
         InsertarOrdenado ( vect , i , dato)
      Fin_mientras

      Mostrar ( vect , i )
Fin
```

# Arrays: Cadenas y strings (26/27)

- **Cadenas**
- Hablaremos de cadenas como un caso especial de vectores en el que guardamos caracteres.
- Una cadena no es más que un vector que contiene caracteres, pero que tiene como característica especial que sólo se utiliza una parte del vector, de manera que se coloca un delimitador al final de los caracteres utilizados (por ejemplo, en C/C++ es el carácter con valor 0 habitualmente indicado como '\0').
- Por ejemplo: Supongamos una cadena de 10 caracteres, en la que deseamos guardar la palabra 'hola'. De los diez posibles caracteres, sólo vamos a utilizar cuatro. El contenido del vector será pues:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 'h' | 'o' | 'l' | 'a' | '\0' | ? | ? | ? | ? | ? |

- La declaración de una cadena se realizará como la de un vector en el que vayamos a guardar caracteres:

  Nombre [Tamaño] : caracteres
- Teniendo en cuenta que una cadena no es más que un caso especial de vector, las operaciones sobre las cadenas se realizarían elemento a elemento al igual que en el caso general de vectores, además de considerar el elemento de final de cadena.

# Arrays: Cadenas y strings (27/27)

- **Strings**

- Aunque los lenguajes de programación tienen funciones específicas que permiten realizar operaciones sobre las cadenas, estas resultan algo engorrosas.
- Por ello, casi todos los lenguajes definen un tipo nuevo para manejar cadenas de caracteres. Este tipo es básicamente un vector de caracteres que lleva asociado un entero y un conjunto de funciones que permiten realizar operaciones de cadenas. El entero representa la longitud actual de la cadena (es decir, el número de caracteres que almacena). A este tipo de dato le llamaremos *string*.
- El concepto de string es equivalente al de cadena pero las funciones que permiten realizar operaciones sobre ellos son más sencillas de manejar y además la longitud de la cadena no se determina por ningún carácter especial, por lo que siempre que manejemos variables de tipo string lo haremos a través de las operaciones definidas por el lenguaje de programación para trabajar sobre los string.

- Este tema lo veremos en prácticas.

# Arrays: Cadenas y strings (27/27)

- Funciones para el manejo de strings
- // nº de caracteres de palabra
  i = palabra.length();
- // inserta palabra en la posición 3 de frase
  frase.insert(3, palabra);
- // concatena (une) palabra y "hola" y almacena el resultado en frase
  frase = palabra + "hola";
- // concatena (añade al final) palabra a frase
  frase += palabra;
- // borra 7 caracteres de frase desde la posición 3
  frase.erase (3,7);
- // sustituye (reemplaza) 6 caracteres de frase, empezando en la posición 1, por la cadena palabra
  frase.replace (1, 6, palabra);
- //busca palabra como una subcadena dentro de frase desde la posición comienzo, devuelve la posición donde la encuentra (o -1 si no la encuentra)
  i = frase.find(palabra, comienzo);
  i = frase.find(palabra); //empieza desde el principio
- //devuelve la subcadena formado por 3 caracteres desde la posición 5 de frase
  palabra = frase.substr(5,3);
- //conversión de string a cadena de caracteres de C
  **`cadena = frase.c_str();`**

# Estructuras o registros (1/14)

- Las estructuras o registros son agrupaciones heterogéneas, contiguas y estáticas.

- Los elementos contenidos en la agrupación pueden ser de distintos tipos y se suelen llamar 'campos de información', y suelen estar relacionados con la información referente a un objeto concreto.

- Ejemplos:
  - Una persona (Nombre, edad, D.N.I.,...)
  - Un libro (Título, autor, precio, disponible/agotado,...)

# Estructuras o registros (2/14)

- La declaración de una estructura o registro se realizará de la siguiente manera:

```
Registro Nombre
   Campo1 : Tipo1
   Campo2 : Tipo2
    …
FIN_Registro
```

- Nombre es el nombre que va a recibir nuestra estructura.
- Tipo1 será el tipo de datos del primer campo de información y Campo1 será el nombre del primer campo de información
- Tipo2 será el tipo de información guardado en el segundo campo de información y Campo2 el nombre del segundo campo de información.
- Y así para todos los campos de información que queramos tener.

# Estructuras o registros (3/14)

- La manera de declarar una estructura es en C/C++ es:

```
struct Nombre
{
    Tipo1 Campo1;
    Tipo2 Campo2;
    ...
};
```

- Los registros o estructuras definen nuevos tipos de datos que podemos utilizar a lo largo de nuestro programa.

- Es decir, una vez declarada la estructura podemos utilizar y declarar variables de ese nuevo tipo (igual como si declarásemos variables de tipo entero o real.):
  - NombreVar : Nombre
  - En C/C++: struct Nombre NombreVar o Nombre NombreVar

# Estructuras o registros (4/14)

- **Asignación**

- La manera de acceder a cada uno de los campos es mediante el operador de acceso '.'
  - NomVar.Campo1

- El tratamiento de cada uno de estos campos depende de su tipo (si es un entero lo asignaremos y trataremos como entero, si es un vector como un vector, si es un string como un string,...).
  - NomVar.Campo1 ← Valor

- Ejemplos: Si p es una variable que define a una persona
  - p.Nombre ← "Pepe"
  - p.Edad ← 45
  - p.DNI ← 45678987
  - p.Sexo ← 'H'

# Estructuras o registros (5/14)

- Ejemplo: Creación y asignación de valores a personas:

```
Registro persona
    Nombre: string
    Edad: entero
    DNI: entero
    Sexo: carácter
FIN_Registro
```

```
per : persona
…
Escribir (“Dame el nombre: \n“)
Leer (per.Nombre)
Escribir (“Dame la edad: \n“)
Leer (per.Edad)
Escribir (“Dame el D.N.I.: \n“)
Leer (per.DNI)
Escribir (“Dame el sexo: \n“8)
Leer (per.Sexo)
…
```

# Estructuras o registros (6/14)

- En C/C++ se pueden inicializar los campos de un registro en el momento de la creación de una variable:

```
struct Fecha
{
    int dia;
    string mes;
    int anyo;
};

Fecha f = {1,”Noviembre”,2010};
```

# Estructuras o registros (7/14)

- **Anidamiento de estructuras:**
- Las estructuras se pueden anidar, es decir, uno de los campos de una estructura puede ser a su vez otra estructura:

```
Registro alumno
    InfoPersonal : persona
    FechaNacimiento : fecha
    Curso : entero
FIN_Registro
```

- También podemos definir arrays de registros:
  - NombreVector[Tamaño] : NombreEstructura

# Estructuras o registros (8/14)

- Las estructuras sí se pueden asignar, al igual que los strings o cualquier tipo simple.
- Una asignación de estructuras es equivalente a una asignación de cada uno de los componentes.
  - hoy, ayer : fecha;
  - ayer ← hoy;
- **Nota**: la asignación es válida siempre y cuando la estructura no contenga un vector como uno de sus campos.
- Las estructuras se pasan como parámetros exactamente igual que cualquier otro tipo simple, tanto por valor como por referencia.
- Dado que podemos hacer asignaciones entre estructuras: Las funciones también pueden devolver estructuras.

# Estructuras o registros (9/14)

- Ejemplo: Decir quién es la persona de mayor edad entre dos

```
FUNCION QuienEsMayor(p1 : persona, p2 : persona) : persona
 Variables:
       mayor : persona
 Inicio
    SI (p1.edad >= p2.edad) ENTONCES
       mayor ← p1
    SINO
       mayor ← p2
    FIN_SI
    QuienEsMayor ← mayor
FIN_Funcion
```

```
FUNCION LeerPer() : persona
 Variables:
       p : persona
 Inicio
    Escribir("Dame el nombre,edad, sexo")
    Leer ( p.Nombre, p.edad, p.sexo)
    LeerPer ← p
FIN_Funcion

PROCEDIMIENTO MostrarPer(p : persona)
 Inicio
    Escribir("Los datos son:")
    Escribir ( p.nombre, p.edad, p.sexo)
FIN_Procedimiento
```

```
Registro persona
   nombre: string
   edad: entero
   sexo: carácter
FIN_Registro


PROGRAMA PRINCIPAL

Variables
     per1, per2, m : persona
Inicio
     per1 ← LeerPer()
     per2 ← LeerPer()

     m ← QuienEsMayor(per1, per2)

     MostrarPer (m )
Fin
```

# Estructuras o registros (10/14)

- Ejemplo(v1): Decir quién es “el alumno” de mayor edad entre dos

```
FUNCION AlumnoMayor(p1 : alumno, p2 : alumno) : alumno
 Variables:
 mayor : alumno
 Inicio
    SI (p1.datos.edad >= p2.datos.edad) entonces
       mayor ← p1
    SINO
       mayor ← p2
    FIN_SI
    AumnoMayor ← mayor
FIN_Funcion
PROCEDIMIENTO LeerAlum( ref  a : alumno)
 Inicio
    Escribir(“Dame el nombre,edad, sexo y curso”)
    Leer ( a.datos.nombre, a.datos.edad, a.datos.sexo)
    Leer( a.curso)
 FIN_Procedimiento

PROCEDIMIENTO MostrarAlum (a : alumno)
 Inicio
    Escribir(“Los datos son:”,  a.datos.nombre)
    Escribir ( a.datos.edad, a.datos.sexo, a.curso)
 FIN_Procedimiento
```

```
 Registro persona
    nombre: string
    edad: entero
    sexo: carácter
FIN_Registro

 Registro alumno
    datos: persona
    curso: entero
FIN_Registro

 PROGRAMA PRINCIPAL

 Variables
      a1, a2, m : alumno
 Inicio
      LeerAlum( a1 )
      LeerAlum( a2 )

      m ← AlumnoMayor(a1, a2)

      MostrarAlum (m )
 Fin
```

# Estructuras o registros (11/14)

- Ejemplo(v2): Usando las funciones “de persona”

```
FUNCION AlumnoMayor(p1 : alumno, p2 : alumno) : alumno
 Variables:
 mayor : alumno
 Inicio
    mayor.datos ← QuienEsMayor(p1.datos,p2.datos)
    AumnoMayor ← mayor
 FIN_Funcion

PROCEDIMIENTO LeerAlum( ref  a : alumno)
 Inicio
    a.datos ← LeerPer()
    Escribir (“Dame el curso”)
    Leer( a.curso)
 FIN_Procedimiento

PROCEDIMIENTO MostrarAlum (a : alumno)
 Inicio
    MostrarPer( a.datos )
    Escribir ( “Y el curso es “, a.curso)
 FIN_Procedimiento
```

```
 Registro persona
    nombre: string
    edad: entero
    sexo: carácter
FIN_Registro

 Registro alumno
    datos: persona
    curso: entero
FIN_Registro

 PROGRAMA PRINCIPAL

 Variables
      a1, a2, m : alumno
 Inicio
      LeerAlum( a1 )
      LeerAlum( a2 )

      m ← AlumnoMayor(a1, a2)

      MostrarAlum (m )
 Fin
```

# Estructuras o registros (12/14)

- Ejemplo: Programa para ver quién es el mayor de 10 alumnos (usando las funciones/procedimientos del ejemplo anterior)

```
PROGRAMA PRINCIPAL

Variables
      a[10], m : alumno
      i : entero
Inicio
      Desde1 i ← 0 hasta 9 hacer
            LeerAlum( a[i] )
            i ← i + 1
      Fin_desde1

      m ← a[0]

      Desde2  i ← 1 hasta 9 hacer
            m ← AlumnoMayor(m, a[i] )
            i ← i + 1
      Fin_desde2

      MostrarAlum (m )
Fin
```

```
 Registro persona
    nombre: string
    edad: entero
    sexo: carácter
FIN_Registro

 Registro alumno
    datos: persona
    curso: entero
FIN_Registro
```

# Estructuras o registros (13/14)

- Ejemplo: Programa para ver quién es el mayor de 10 alumnos (haciendo una función que devuelva la posición en que se encuentra en el vector)

```
FUNCION AlumnoMayor( v[10] : alumno) : entero
 Variables:
 mayor, pos, i : entero
 Inicio
      pos ← 0
      mayor ← v[0].datos.edad

      Desde  i ← 1 hasta 9 hacer
            SI (v[i].datos.edad >= mayor) entonces
                  mayor ← v[i].datos.edad
                  pos ← i
            FIN_SI
           i ← i + 1
      Fin_desde

      AlumnoMayor ← pos
FIN_Funcion
```

```
Registro persona
   nombre: string
   edad: entero
   sexo: carácter
FIN_Registro

Registro alumno
   datos: persona
   curso: entero
FIN_Registro
PROGRAMA PRINCIPAL

Variables
     a[10] : alumno
     i , pos: entero
Inicio
     Desde i ← 0 hasta 9 hacer
        LeerAlum( a[i] )
        i ← i + 1
     Fin_desde

     pos ← AlumnoMayor( a )
     MostrarAlum ( a[pos] )
Fin
```

# Estructuras o registros (14/14)

- **Ejercicio**: Arregla el programa anterior para que ahora nos devuelva quién es el mayor de N alumnos, dónde N es un valor introducido al principio por teclado y que nunca será mayor a 100.

- **Versión 1: usando sólo las estructuras anteriores de persona y alumno**

FUNCION AlumMayor1( v[100] : alumno , t : entero) : entero

```
Registro persona
   nombre: string
   edad: entero
   sexo: carácter
FIN_Registro

Registro alumno
   datos: persona
   curso: entero
FIN_Registro
```

- **Versión 2: usando la estructura lista_alum**

FUNCION AlumMayor2( lis : lista_alum) : entero

```
Registro lista_alum
      v[TAM]: alumno
      num : entero
FIN_Registro
```

# Introducción a las estructuras dinámicas (1/11)

- **Punteros**
- Existe en casi todos los lenguajes de programación un tipo de datos simple que es el 'puntero'.
- Un puntero es un tipo de dato que es capaz de guardar una dirección de memoria.
- Las direcciones de memoria sirven al ordenador para saber en qué lugar de la memoria se sitúa exactamente una determinada información, y en general cualquier variable lleva asociada una dirección de memoria.
- Los operadores básicos con punteros son:

**`&`** Obtiene la dirección de memoria en donde se encuentra una variable, a partir del nombre de la variable.

**`*`** Obtiene el contenido (valor) que se guarda en una determinada posición de memoria.

**`malloc`** Reserva espacio en memoria para una determinada información y devuelve la dirección de memoria en donde se encuentra ese espacio.

**`free`** Libera el espacio reservado para una información (le dice al ordenador que ese espacio puede ser utilizado por otros usuarios.)

# Introducción a las estructuras dinámicas (2/11)

- Con el tipo de datos puntero y los registros podemos construir agrupaciones de elementos que pueden ir creciendo en memoria a medida que va aumentando al información que deseamos guardar en el ordenador, de manera que no tenemos que fijar el número máximo que queremos guardar en la agrupación en el momento de escribir el programa.

- En general podremos construir agrupaciones dinámicas incluyendo un puntero como campo de un registro, de manera que a través de este puntero puedo acceder a otros elementos de la agrupación.

- Una definición de un registro de este tipo, podría ser en C:

```
struct Nodo
{
   Valor Info;
   struct Nodo * Sig;
};
```

# Introducción a las estructuras dinámicas (3/11)

- A estos registros, que contienen un puntero se les suele llamar NODOS, y son la base de las agrupaciones dinámicas, ya que gracias a estos punteros podemos enlazar todos los elementos de información que queramos (limitados tan solo por la capacidad física de la memoria del ordenador) y además gracias a la instrucción 'malloc' podemos reservar memoria para nuevos elementos a medida que nos vaya haciendo falta.

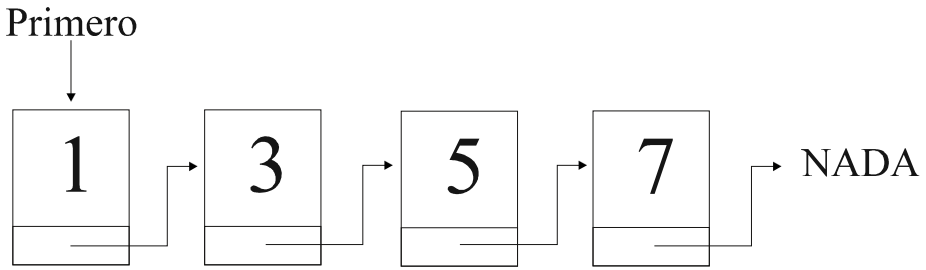
- La manera de acceder a los campos de un registro a partir de un puntero es mediante el operador 'contenido' (el *) y el operador de acceso '.'
  struct Nodo * p_aux;

  (*p_aux).Info / (*p_aux).Sig;

- Esta combinación es equivalente al operador de acceso en punteros '->' (un menos seguido de un mayor) de manera que quedaría:
  p_aux->Info / p_aux->Sig

# Intro. estr. dinámicas: Listas enlazadas (4/11)

- Una lista es una agrupación de elementos entre los que existe uno que podemos llamar 'primero' y a partir del cual se puede acceder al resto de los elementos uno a uno.
- Las listas enlazadas están definidas por:
  1. Una estructura de tipo nodo.
  2. Un puntero que nos marca el primer elemento, a partir del cual puede accederse a cualquier otro elemento de la agrupación.
- La idea a la que se quiere llegar sería la siguiente: La lista de números 1, 3, 5, 7 quedaría en forma enlazada como sigue:




- Primero sería el puntero que señalaría al primer elemento de la lista. Mediante el puntero situado en cada uno de los nodos es posible acceder al siguiente elemento desde uno cualquiera.
- A 'NADA' en C/C++ se le llama 'NULL'

# Intro. estr. dinámicas: Listas enlazadas (5/11)

- Operaciones
- *Creación*
- Cuando creamos una lista, en principio estará sin elementos. La creación de una lista llevará asociado, la declaración del tipo Nodo y una variable que sea capaz de guardar dónde está el primero de los elementos, así como la asignación de que no hay ningún elemento en la lista.

```
struct Nodo
{
      Valor Info;
      struct Nodo * Sig;
};
...
struct Nodo * Primero;
...
Primero = NULL;
```

- La declaración del tipo Nodo se debe realizar al comenzar el programa, después de los “includes” y los “defines”.
- La variable de tipo puntero se declara en el lugar adecuado de la declaración de variables. Y la asignación en el punto del programa en el que deseemos iniciar la variable.

# Intro. estr. dinámicas: Listas enlazadas (6/11)

- *Búsqueda de un elemento*
- En el caso de listas dinámicas, sólo podremos realizar búsqueda secuencial, es decir, partiendo del primer elemento ir mirando si el que estamos buscando es o no el indicado.

```
bool Buscar (struct Nodo * primero, int x)
{
    struct Nodo * p_aux;
    bool encontrado;

    p_aux = primero;
    while ( (p_aux != NULL) && (p_aux->Info != x) )
        p_aux = p_aux->Sig

    if (p_aux == NULL)
        encontrado = false;
    else
        encontrado = true;

    return encontrado;
}
```

# Intro. estr. dinámicas: Listas enlazadas (7/11)

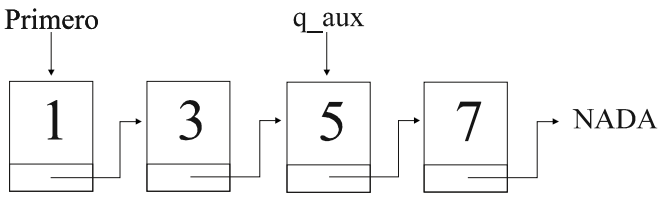
- *Inserción de un elemento delante del primero*
- Supongamos que queremos insertar el elemento ‘x’ delante del primero:

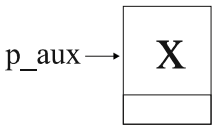
```
struct Nodo * p_aux;
...
/* Reservamos memoria para el nuevo elemento */
p_aux = malloc (sizeof (struct Nodo) );

/* Actualizamos su informacion */
p_aux->Info = x;

/* Enlazamos el elemento en la lista */
p_aux->Sig = primero;

/* El primero ahora es el que hemos insertado */
primero = p_aux;
```

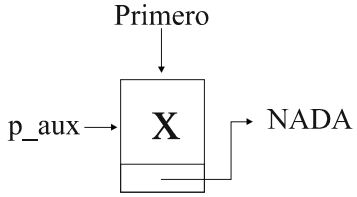
# Intro. estr. dinámicas: Listas enlazadas (8/11)

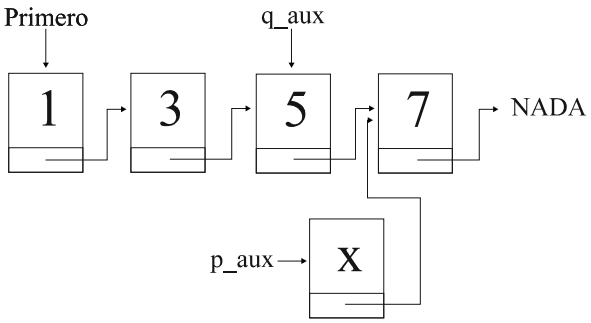
- *Inserción de un elemento detrás de uno dado*
- Supongamos que queremos insertar el elemento ‘x’ detrás de uno marcado con ‘q_aux’:

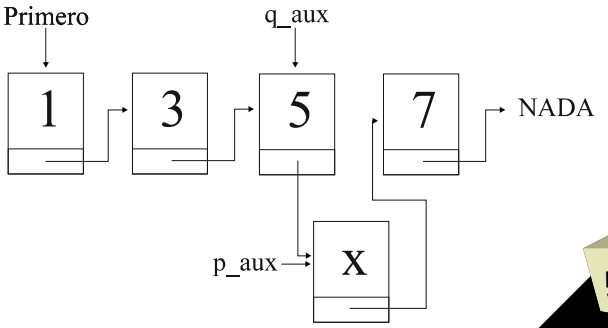
```
void InsertarDetras (struct Nodo * primero, struct Nodo * q_aux, Valor x)
{
   struct Nodo * p_aux;


   p_aux = malloc (sizeof (struct Nodo) );
   p_aux->Info = x;

   /* Si no habia elementos solo tendremos el que acabamos de crear */
   if (primero == NULL)
   {
      p_aux->Sig = NULL;
      primero = p_aux;
   }
   else
   {
      p_aux->Sig = q_aux->Sig;

      q_aux->Sig = p_aux;

   }
}
```



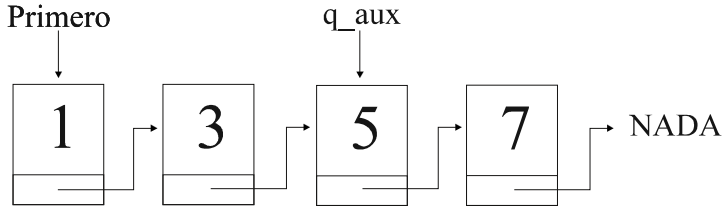



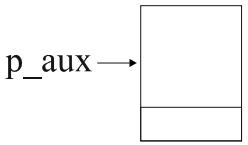



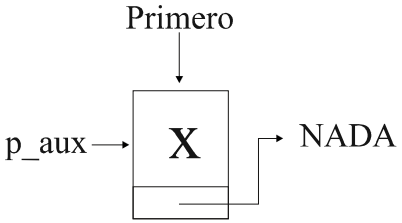
# Intro. estr. dinámicas: Listas enlazadas (9/11)

- *Inserción de un elemento delante de uno dado*
- Supongamos que queremos insertar el elemento ‘x’ delante de uno marcado con ‘q_aux’:

```
void InsertarDelante (struct Nodo * primero, struct Nodo * q_aux, Valor x)
{
   struct Nodo * p_aux;


   p_aux = malloc (sizeof (struct Nodo) );


   /* Si no habia elementos solo tendremos el que acabamos de crear */
   if (primero == NULL)
   {
      p_aux->Info = x
      p_aux->Sig = NULL;
      primero = p_aux;
   }
   else
   {

         (ver diapositiva siguiente...)

   }
}
```

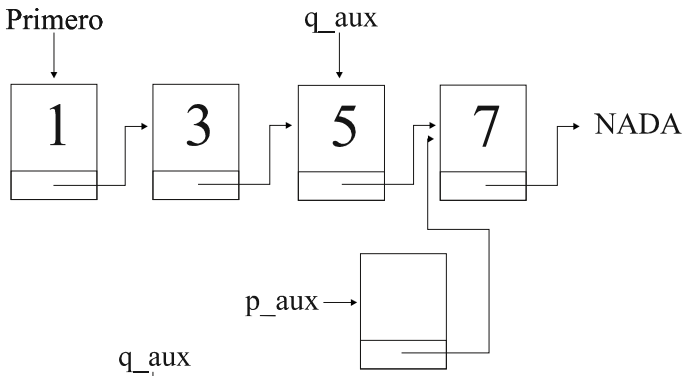


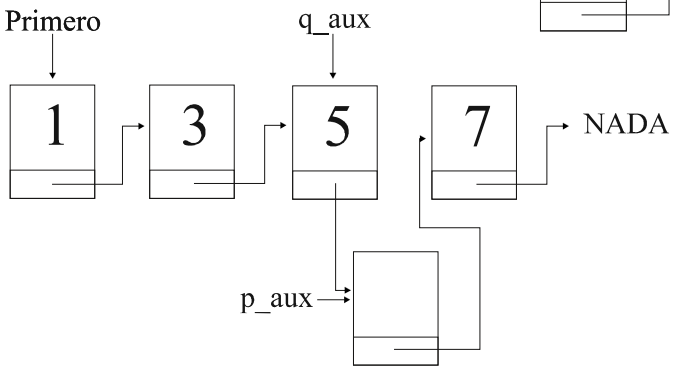



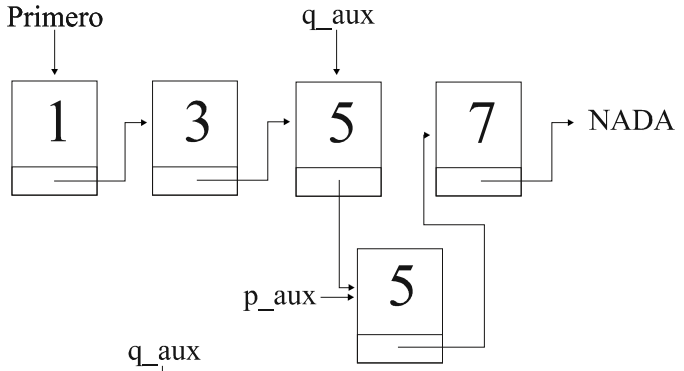



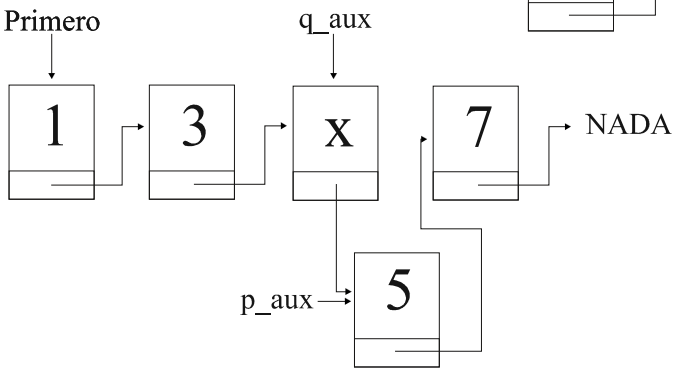
# Intro. estr. dinámicas: Listas enlazadas (10/11)

```
else
{
        p_aux->Sig = q_aux->Sig;

        q_aux->Sig = p_aux;

        p_aux->Info = q_aux->Info;

        q_aux->Info = x;

}
```



# Intro. estr. dinámicas: Listas enlazadas (11/11)

- *Ventajas e inconvenientes de las agrupaciones estáticas frente a las agrupaciones dinámicas:*

  - Mayor facilidad para insertar y eliminar elementos en las agrupaciones dinámicas.
  - Más fácil y rápido realizar búsquedas en vectores.
  - Si el número de elementos es conocido o no va a variar mucho → agrupaciones estáticas
  - Si el número de elementos va a ser cualquiera → agrupaciones dinámicas.
  - En general, para un mismo número de elementos las agrupaciones dinámicas ocuparán más memoria que las estáticas (por cada elemento se tiene también un puntero en las agrupaciones dinámicas.)