

# Computación de Altas Prestaciones Seminario 2



# Computación de Altas Prestaciones: Contenidos de la asignatura en seminarios

- ◆ Conceptos básicos de punteros en C
- ◆ Producto de matrices en C

# Programación en C

Punteros y arrays

A stylized, dark teal silhouette of a mountain range is located in the bottom right corner of the slide, adding a decorative element to the background.

# Programación en C

- ◆ El lenguaje C (y C++) son lenguajes de uso muy común, caracterizados por su eficiencia.
- ◆ Relativamente complejos, especialmente en lo que se refiere a los punteros.
- ◆ Han sido "ligeramente" desplazados por Java, .NET
- ◆ Las instrucciones básicas (for, if, while) tienen exactamente la misma sintaxis que en Java.
- ◆ La declaración de vectores en memoria estática también es idéntica a Java, así como la declaración de variables simples (int, double, float, char).

# Operador de dirección de memoria

- ◆ El operador & nos da la dirección de memoria de una variable:

```
#include <stdio.h>
void main()
{ int i;
  i=2;
  printf("La dirección de i es %d",&i);
}
```

Este operador solo se puede aplicar a variables, no a constantes

# Operador de indirección: \*

- ◆ Dada una dirección válida de memoria, El operador \* aplicado sobre esa dirección me da la variable contenida en esa dirección

```
#include <stdio.h>
void main()
{ int i;
  i=2;
  printf ("i vale %d \n", i);
  printf("Otra forma: i vale  %d \n",*&i);
}
```

El operador de indirección solo se debe aplicar sobre direcciones válidas de memoria

# Puntero

- ◆ Un puntero es una variable que contiene una dirección válida de memoria. En la declaración hay que indicar el tipo de datos de la variable que se va a "apuntar"

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

```
void main()
```

```
{
```

```
int i;
```

```
int *p;
```

```
i=2;
```

```
p=&i;
```

```
printf(" Lo que es apuntado por p, vale %d  
\n", *p);
```

```
}
```

# Aplicaciones de los Punteros



# Vectores y punteros

Vector estático

```
#include <stdio.h>
void main()
{
    int datos[3];
    ...
    datos[1]=56;
}
```

# Vectores y punteros

- Los vectores y los punteros están íntimamente relacionados, son "casi" lo mismo
- Mediante punteros y la función "malloc" (**m**emory **a**lloc**a**te), podemos obtener vectores cuyo tamaño se determina al ejecutar el programa.
- La memoria obtenida con **malloc** se debe liberar con **free** al final del programa.
- La función **sizeof** suele ser útil para obtener el tamaño correcto de nuestro vector dinámico.

# Vectores y punteros

Vector dinámico (pero estático)

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

```
main()
{
    int *datos ;
    datos= (int *) malloc (3*sizeof(int));
    ...
    datos[2]=56;
    free(datos);
}
```

# Vectores y punteros

Vector "realmente" dinámico

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

```
main()
{
    int *datos ,n;
    printf("Introduce el tamaño del vector:");
    scanf("%d",&n);
    datos= (int *) malloc (n*sizeof(int));
    ...
    datos[0]=56;
    free(datos);
}
```

# Vectores y punteros

NO se puede declarar un vector estático de esta forma

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

```
main()
{
    int n;
    printf("Introduce el tamaño del vector:");
    scanf("%d",&n);
    int datos[n];
    ...
    datos[0]=56;
    free(datos);
}
```

Daría un error...(aunque en algún compilador moderno funciona!)

# Vectores y punteros

El nombre de un vector es una dirección de memoria:

```
int datos[3];
```

**&datos[0]** o **datos** o **(datos+0)** es la dirección de memoria del primer dato del vector;

**&datos[1]** o **(datos+1)** es la dirección de memoria del segundo

dato del vector;

...

**\*datos** o **datos[0]** es el contenido o valor del primer dato del vector;

**\*(datos+1)** o **datos[1]** es el contenido o valor del primer dato del vector;

# Función para leer datos de teclado a un vector

CAP-MUlinf

```
#include <stdio.h>
#include <stdlib.h>
void leervector(int *vector, int longitud) // void leervector(int vector[], int longitud)
{ int i;
  for (i=0;i<longitud;i++)
  {   printf(" \n Introduce un numero: ");
      scanf("%d", &vector[i]);
  }
}
void main() {
int *vector;
int num=5, i;
vector=(int*) malloc(num*sizeof(int));
leervector(vector,num);
for (i=0;i<num;i++)
{
  printf(" \n posicion %d : %d ", i, vector[i]);
}
free(vector);
}
```

# Punteros para implementar paso por referencia;

El paso de argumentos en C es por valor (la variable enviada a la subrutina no ve modificado su valor en el programa principal)

```
#include <stdio.h>
void fun_invocada (int vble_recibir)
{vble_recibir=vble_recibir+1;
}

void main()
{
int vble_enviar=1;
fun_invocada(vble_enviar);
printf("El valor de la variable es %d\n",vble_enviar);
}
```



# Punteros para implementar paso por referencia;

Mediante punteros podemos implementar paso por referencia, fundamental para grandes estructuras de datos y para tener argumentos de salida.

```
#include <stdio.h>
void fun_invocada (int *vble_recibir)
{ *vble_recibir=*vble_recibir+1;
}

void main()
{
int vble_enviar=1;
fun_invocada(&vble_enviar);
printf("El valor de la variable es %d\n",vble_enviar);
}
```

# Matrices en Matlab y en C

# Matrices en Matlab y en C

CAP-MUInf

$$\text{Ejemplo: } A = \begin{pmatrix} 4 & -1 & 2 & -7 \\ 3 & 5 & 4 & -1 \\ -1 & 0 & 3 & -2 \end{pmatrix}$$

La primera posibilidad para declarar matrices en C es la definición "estática":

```
double A[3][4];
```

```
..
```

```
A[0][0]=2;
```

```
suma=0;
```

```
for (i=0;i<3;i++)
```

```
    for(j=0;j<4;j++)
```

```
        suma=suma+A[i][j];
```

# Almacenamiento de matrices en Matlab<sup>CAR-MUinf</sup> y en C

$$\text{Ejemplo: } A = \begin{pmatrix} 4 & -1 & 2 & -7 \\ 3 & 5 & 4 & -1 \\ -1 & 0 & 3 & -2 \end{pmatrix}$$

El problema de la declaración estática es que (según la definición del lenguaje C) en realidad estamos declarando 3 vectores de tamaño 4:

**double A[3][4];**

En memoria: [....., 4, -1, 2, 7, ....., 3, 5, 4, -1, ..., -1, 0, 3, -2, .....]

Puede haber “huecos” en memoria entre las diferentes filas de la matriz. Existen muchos programas y librerías de manejo de matrices que no admiten este formato (Matlab entre ellos)

# Almacenamiento de matrices en Matlab<sup>CAR-MUinf</sup> y en C

$$\text{Ejemplo: } A = \begin{pmatrix} 4 & -1 & 2 & -7 \\ 3 & 5 & 4 & -1 \\ -1 & 0 & 3 & -2 \end{pmatrix}$$

El almacenamiento que se usa en Matlab es de forma contigua,  
**por columnas:**

En memoria: [....., 4, 3, -1, -1, 5, 0, 2, 4, 3, -7, -1, -2 .....

Este almacenamiento se usa en muchas aplicaciones aparte de Matlab.

OJO!: Este es el almacenamiento \*\*\*interno\*\*\* de Matlab.  
Para meter la matriz del ejemplo por teclado en Matlab, lo haríamos así:

```
>> A=[4 -1 2 -7; 3 5 4 -1; -1 0 3 -2]
```

# Declaración de matrices en forma contigua en C

Ejemplo:  $A = \begin{pmatrix} 4 & -1 & 2 & -7 \\ 3 & 5 & 4 & -1 \\ -1 & 0 & 3 & -2 \end{pmatrix}$

Podemos declarar la matriz de forma estática:

```
double A[3*4];
```

o usando memoria dinámica:

```
double *A;  
int nfilas=3, ncolumnas=4;  
A= (double *) malloc (nfilas*ncolumnas*sizeof(double));
```

# Uso de matrices en forma contigua en C<sup>CAP-MUlinf</sup>

Ejemplo:  $A = \begin{pmatrix} 4 & -1 & 2 & -7 \\ 3 & 5 & 4 & -1 \\ -1 & 0 & 3 & -2 \end{pmatrix}$

Almacenada como vector contiguo: [4, 3, -1, -1, 5, 0, 2, 4, 3, -7, -1, -2]

De cualquiera de las dos formas, la matriz se almacenará en memoria como un solo bloque contiguo de memoria.

Una vez declarada esta matriz, el elemento en la fila  $i$ , columna  $j$  de la matriz, será el elemento:

$$A[i+j*nfilas]$$

Recordemos que la numeración de los vectores en C empieza por 0

También se suelen utilizar macros, por ejemplo, si la matriz se almacena por columnas:

```
#define A(i,j) A[(i)+(j)*n]
```

# Declaración de matrices en forma contigua<sup>CAP-MULinf</sup>

Ejemplo en C:

Leer de teclado los elementos de una matriz, y obtener la suma de todos los elementos de esa matriz:

## Versión estática no contigua

```
double A[3][4];
for (i=0;i<3;i++)
    for(j=0;j<4;j++)
        scanf("%f",&A[i][j]);
suma=0;
for (i=0;i<3;i++)
    for(j=0;j<4;j++)
        suma=suma+A[i][j];
```

## Versión estática contigua

```
double A[3*4];
for (i=0;i<3;i++)
    for(j=0;j<4;j++)
        scanf("%f",&A[i+j*3]);
suma=0;
for (i=0;i<3;i++)
    for(j=0;j<4;j++)
        suma=suma+A[i+j*3];
```

Necesitamos disponer del número de filas (3 en este caso)

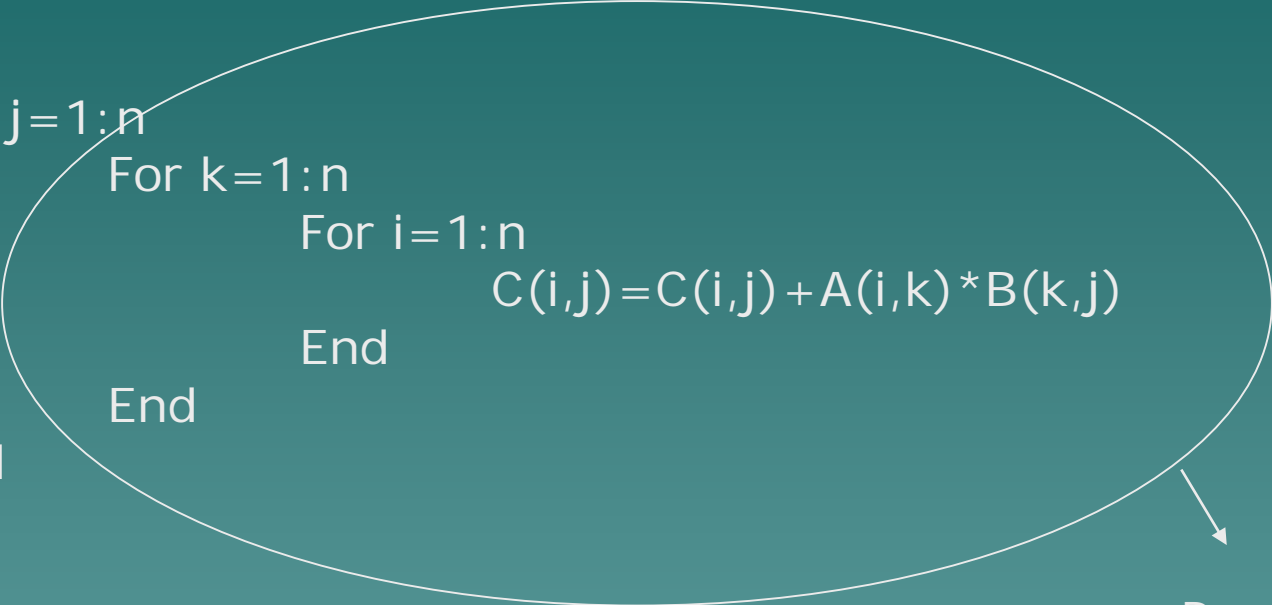


# Ejercicio: Producto de matrices en C

- ◆ Ejercicio: Se suministra un programa, prueba\_producto.c en poliformat, para probar el producto de matrices (almacenamiento contiguo por columnas, usando "leading dimension"). Escribe una subrutina en C para comprobar cuanto tarda en ejecutarse el producto de dos matrices cuadradas 1000 por 1000. Comprueba los diferentes ordenes de los bucles i,j,k.
- ◆ Una vez creada la ´subrutina para el producto de matrices, se compila con: gcc prueba\_producto.c ctimer.c
- ◆ Experimenta con opciones de compilaci3n: -O3, -g, para gcc (depende de la m1quina donde se ejecute);

# Paralelización del producto de matrices con OpenMP

```
For j=1:n
  For k=1:n
    For i=1:n
      C(i,j)=C(i,j)+A(i,k)*B(k,j)
    End
  End
End
```



Producto Matriz vector

```
C(:,1) = C(:,1) + A * B(:,1)
C(:,2) = C(:,1) + A * B(:,2)
```

...

La columna i-ésima de la matriz C se puede obtener como producto de la matriz A por la columna i-ésima de B

# Paralelización del producto de matrices con OpenMP

- ◆ El producto de matrices ofrece diferentes posibilidades de paralelización. La mas sencilla consiste en considerar el producto de matrices como una serie de productos de Matriz por vector, cada uno de ellos generando una columna de la matriz resultado C:
- ◆ A por la primera columna de B, da la primera columna de C;
- ◆ A por la segunda columna de B, da la segunda columna de C;
- ◆ ...

# Paralelización del producto de matrices con OpenMP

- ◆ Podemos paralelizar el bucle de la  $j$  usando OpenMP.
- ◆ Haz pruebas con la versión paralelizada, con matrices de tamaño  $2000*2000$