# Punteros Parte I

Sofía Beatriz Pérez
Daniel Agustín Rosso

sperez@iua.edu.ar
drosso@iua.edu.ar

Centro Regional Univesitario Córdoba
Instituto Univeristario Areonáutico

Informática II - Clase número 2 - Ciclo lectivo 2023

# Agenda

## Disclaimer

Los siguientes slides tienen el objetivo de dar soporte al dictado de la asignatura. De ninguna manera pueden sustituir los apuntes tomados en clases y/o la asistencia a las mismas.
Es importante mencionar que todos este material se encuentra en un proceso de mejora continua.
Si encuentra bugs, errores de ortografía o redacción, por favor repórtelo a sperez@iua.edu.ar y/o drosso@iua.edu.ar. También puede abrir issues en el repositorio de este link: ▸ infoI_IUA_GitLab

# Declaración y almacenamiento de variables I

## Declaración de variables

En general, un programa necesita guardar y leer datos desde la memoria de la computadora. De forma conceptual y simplificada, se pueden pensar a las posiciones de memoria donde esto ocurre se como habitaciones de un hotel con un número de identificación único e irrepetible.

Si se grafica cada declaración de variables:

# Declaración y almacenamiento de variables II

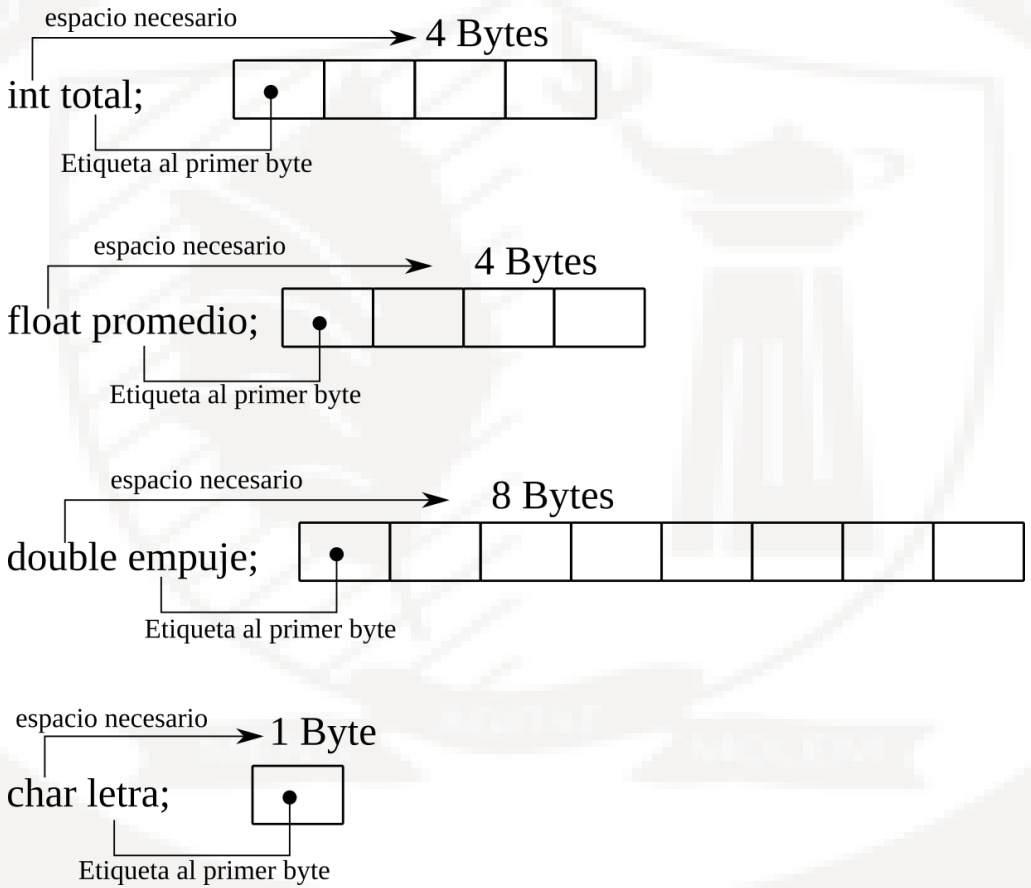


Figure: Diagrama en bloques declaración de variables.

# Aritmética de punteros I

Existen sólo dos operaciones aritméticas que se pueden usar con punteros **la suma y la resta.**

Supongamos que la variable var1 está almacenada en la posición de memoria 1938:

```
1  int var1=10; //almacenado en 1938
2  int *p=&var1;
3  p++;
```

Luego de realizar p++, la variable p contendrá el valor

$$1938 + (4bytes) = 1942$$

y no 1939 como es pensando.

# Aritmética de punteros II

Veamos otro ejemplo con double:

```
1  double var1=10.23; //almacenado en 1938
2  double *p=&var1;
3  p++;
```

Luego de realizar p++, la variable p contendrá el valor

$$1938 + (8bytes) = 1946$$

# Aritmética de punteros III

Veamos otro ejemplo con char:

```
1 char var1="a"; //almacenado en 1938
2 char *p=&var1;
3 p++;
```

Luego de realizar p++, la variable p contendrá el valor

$$1938 + (1byte) = 1939$$

Cada vez que se incrementa un puntero, apunta a la posición de memoria del siguiente elemento de su tipo base. Cuando se aplica a punteros de tipo "char", la aritmética es normal, ya que los caracteres ocupan un byte, **el resto de los punteros aumentan o decrecen en la longitud del tipo de dato a los que apuntan.**

# Aritmética de punteros IV

Diseñar un programa que declare y asigne tres variables de diferentes tipos. Luego declarar tres variables de tipo puntero y asignarles las direcciones de memoria de las tres primeras. Posteriormente, se debe imprimir las direcciones de memoria de cada una de ellas y su contenido. Incrementar en una unidad cada variable puntero y volver a imprimir. Luego de la escritura del programa se debe:

- Analizar los operadores involucrados
- ¿Por qué se debe indicar el tipo de dato en las variables puntero?
- ¿Cuántos bytes se incrementa o decrementan las variables puntero? ¿Depende esto el tipo de dato?

▸ Ver en git (variable_direction)

# Arreglos y punteros I

Se recuerda que los arreglos se definen de forma contigua en la memoria, supongamos las siguientes declaraciones:

```
1  int total[5]={0};
2  char letra[5]={0};
```

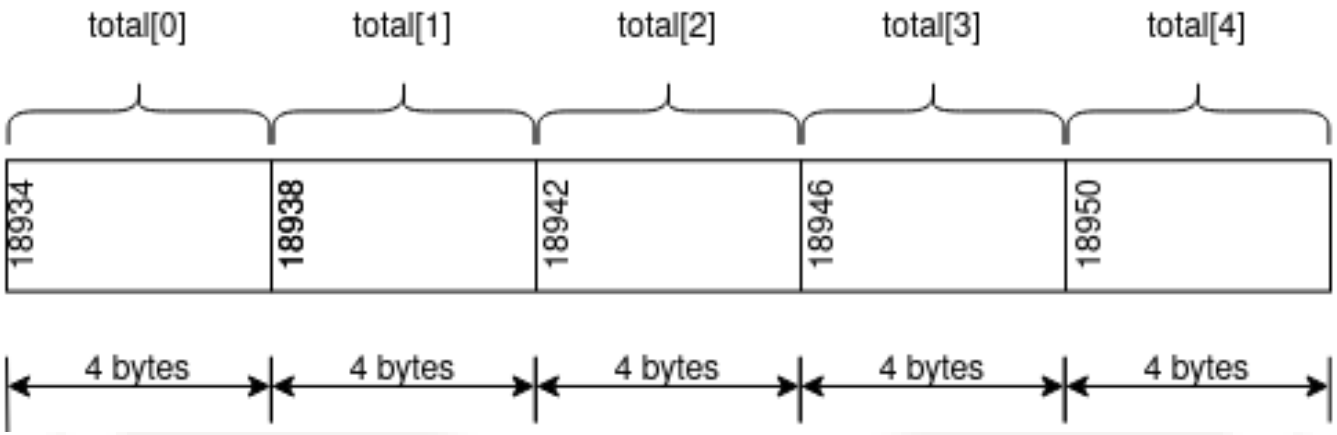


Figure: Arreglo de enteros.

# Arreglos y punteros II

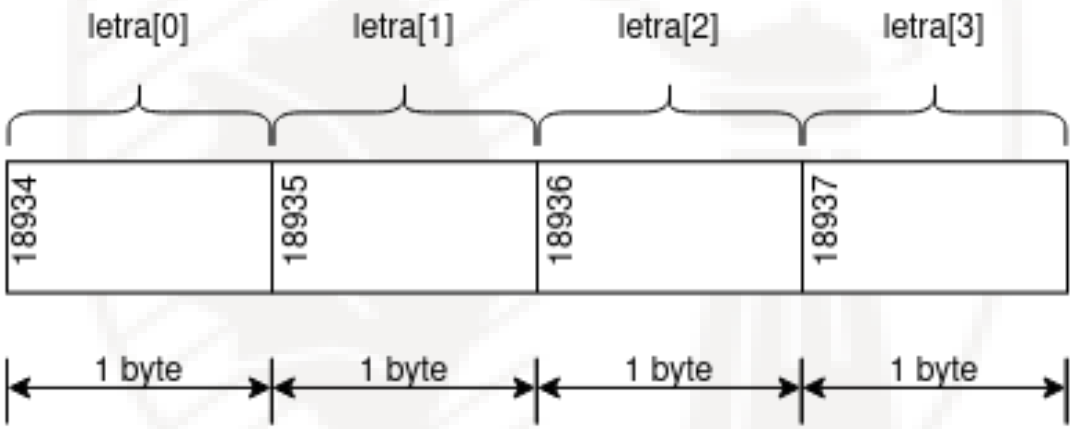


Figure: Arreglo de caracteres.

Conociendo el tamaño que ocupa en memoria cada elemento del arreglo:

# Arreglos y punteros III

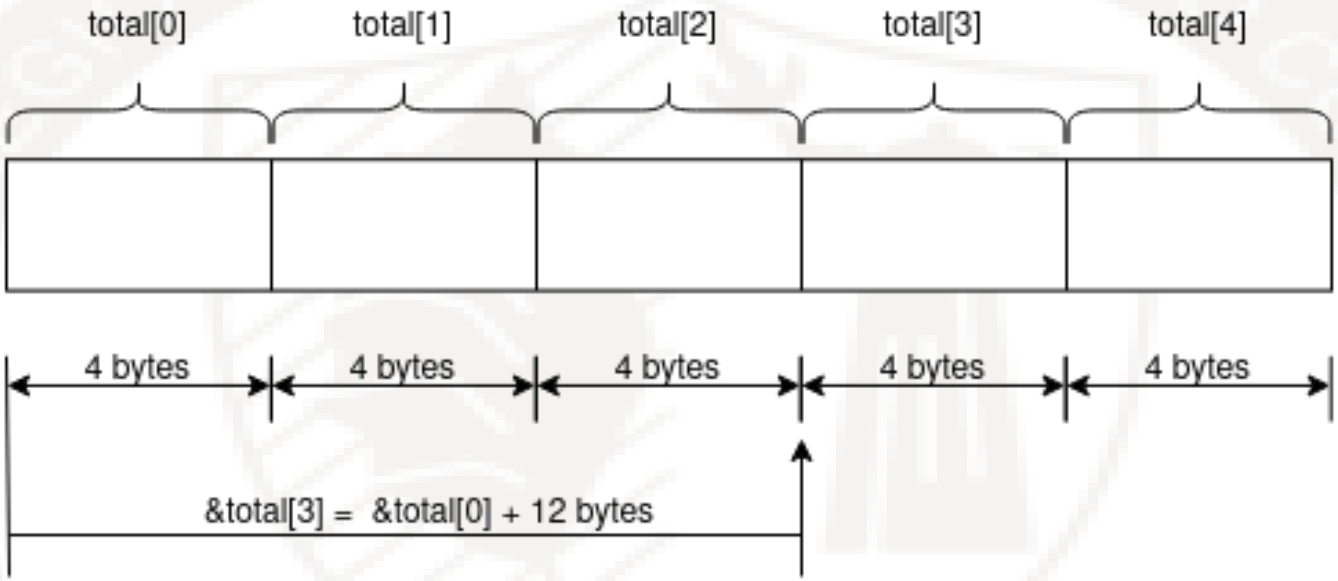


Figure: Arreglo de caracteres.

# Arreglos y punteros IV

Recordando que **en C/C++ el nombre de un arreglo es un puntero al primer elemento del mismo.**. Esto implica que las lineas 3 y 4 son equivalentes.

```
1 int total[5] = {1};
2 int *ptrtotal;
3 ptrtotal = total;
4 ptrtotal = &total[0];
```

Considerando las operaciones de suma y resta de punteros vista anteriormente (aritmética de punteros):

# Arreglos y punteros V

```
1  int total[5] = {1, 2, 3, 4, 5};
2  int *ptrtotal;
3  ptrtotal = total;
4
5  ptrtotal++;
6  total[1];
7
8  ptrtotal++;
9  total[2];
10
11 ptrtotal++;
12 total[2];
```

# Arreglos y punteros VI

Las lineas 5 y 6 apuntan a la misma posición de memoria y por ende al mismo elemento del arreglo.

De forma general:

| Elemento del arreglo | Notación de subíndice | Notación de puntero |
|---|---|---|
| Elemento 0 | total[0] | *ptrtotal |
| Elemento 1 | total[1] | *(ptrtotal+1) |
| Elemento 2 | total[2] | *(ptrtotal+2) |
| Elemento 3 | total[3] | *(ptrtotal+3) |
| Elemento 4 | total[4] | *(ptrtotal+4) |

# Arreglos y punteros VII

Diseñar un programa que declare un arreglo de 5 elementos de tipo entero. Luego, se debe cargar valores e imprimirlos utilizando aritmética de punteros. Finalmente, se deben imprimir las direcciones de memoria de todos los elementos del arreglo
Luego de la escritura del programa se debe:

- Analizar los operadores involucrados
- ¿Por qué no se pone el & en la invocación a scanf(); ?
- ¿En qué dirección de memoria está almacenado el elemento 0, 1 y 2? ¿Cómo accedo a cada uno de ellos utilizando aritmética de punteros?

▸ Ver en git (variable_direction)

# Arreglos, punteros y funciones I

## Desde Informática I...

A diferencia de las variables en las que podemos elegir pasarlas a una función por valor o referencia, los arreglos sólo se pasan a funciones **mediante referencia**.
Es decir que la función trabajará **SIEMPRE** con el arreglo original y no con una copia del mismo.

¿Qué reciben efectivamente estas funciones?

```
1
2  /*PROTOTIPO*/
3  void cargar_vector(int *);
4  void imprimir_vector(int *);
5  void cargar_vector(int []);
6  void imprimir_vector(int []);
```

# Arreglos, punteros y funciones II

```
1
2  /*PROTOTIPO*/
3  void cargar_vector(int *);
4  void imprimir_vector(int *);
5  void cargar_vector(int []);
6  void imprimir_vector(int []);
```

En C, estas funciones reciben **por valor** la dirección de memoria del primer elemento de un arreglo. Conociendo esta dirección de memoria y aplicando aritmética de punteros, se puede operar con las componentes del arreglo original; es por esto que se dice que los arreglos son recibidos en las funciones por referencia.

# Arreglos, punteros y funciones III

En C++ existen nativamente tres maneras de enviar datos a una función

- Por valor
- Por referencia
- por puntero

# Arreglos, punteros y funciones IV

Modificar el programa anterior para que la carga e impresión del arreglo sea realizado en funciones. Utilizar en todos los casos aritmética de punteros.

# Arreglos de punteros I

Los punteros pueden estructurarse en arrays como cualquier otro tipo de datos. Por lo cual, la forma de operar con ellos es igual a como si fuese un arreglo de los ya conocidos:

```
1  int *ptrarray[5];
2  int a=10;
3  ptrarray[3]=&a;
```

# Arreglos de punteros II

```
1 #include<stdio.h>
2 int main(void)
3 {
4     int *ptrarray[5];
5     int a=10;
6     ptrarray[3]=&a;
7     printf("%d ",*(ptrarray[3]));
8     return(0);
9 }
```

# Indirección múltiple I

Se puede hacer que un puntero apunte a otro puntero que apunte a un valor de destino. Esta situación es conocida como *indirección múltiple* o *punteros a punteros*.
La indirección múltiple puede llevarse a la extensión que uno desee, pero existen pocos casos en los que se necesite más de un puntero a puntero.

# Indirección múltiple II

```
1  #include<stdio.h>
2  int main(void)
3  {
4      int x;
5      int *p;
6      int **q;
7
8      x=10;
9      p=&x;
10     q=&p;
11
12     printf("x vale %d",**q);
13     return(0);
14 }
```

# Arreglos multidimencionales y punteros I

También puede tenerse acceso a arreglos multidimencionales usando notación de punteros. La notación se vuelve mas compleja conforme aumentan las dimensiones del arreglo.

```
1  int nums[2][3]={{16,18,20},{25,26,27}};
```

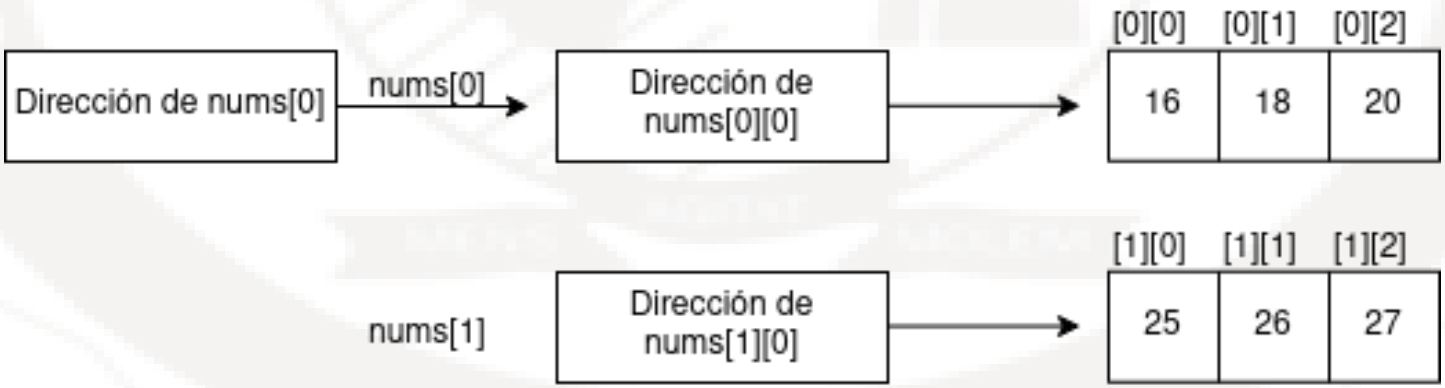


Figure: Arreglo de dos dimensiones y punteros.

# Arreglos multidimencionales y punteros II

| Notación de puntero | Notación de subíndice | Valor almacenado |
|---|---|---|
| *(*nums) | nums[0][0] | 16 |
| *(*nums +1 ) | nums[0][1] | 18 |
| *(*nums +2 ) | nums[0][2] | 20 |
| *(*(nums+1) ) | nums[1][0] | 25 |
| *(*(nums+1) +1 ) | nums[1][1] | 26 |
| *(*(nums+1) +2 ) | nums[1][2] | 27 |

La utilidad de esto se verá cuando se estudie el tema de creación de arreglos dinámicamente.

*¡Muchas gracias!*

Consultas:

sperez@iua.edu.ar
drosso@iua.edu.ar