UNIVERSIDAD NACIONAL DE INGENIERÍA

Facultad de Ciencias

Escuela Profesional de Ciencia de la Computación

- **Curso: Fundamentos de Programación**
- **Docente:** Américo Chulluncuy Reynoso

2025-3

Sesión 4:

Punteros I

Contenido

1. Introducción a la Memoria y Variables
2. ¿Qué es un Puntero?
3. Declaración de Punteros
4. Operadores de Referencia (&) y Desreferencia (*)
5. Puntero a Puntero (Doble Puntero)
6. Punteros y Cadenas de Caracteres
7. Punteros Nulos y buenas prácticas
8. Puntero y Arreglos unidimensionales
9. Aritmética de Punteros.
10. Eficiencia en el uso de Punteros
11. Referencias vs Punteros

1. Introducción a la Memoria y Variables

Cada variable declarada en C++ tiene :

- ✓ **Nombre**, identificador para referirnos a una variable
- ✓ **Tipo**, obligatorio en C++. int, float, char, etc.
- ✓ **Dirección de memoria,** ubicación física en la memoria RAM donde se almacena el valor de la variable.

¿Cómo accede la computadora a una variable?

Cada vez que usamos el nombre de una variable en nuestro código, la computadora realiza **internamente** los siguientes pasos:

1. **Busca la dirección de memoria** asociada al nombre de la variable.
2. **Accede a esa dirección** para recuperar o modificar el valor almacenado.

¿Podemos hacer esto nosotros?

¡Sí! En C++, los **punteros nos permiten acceder directamente a la dirección de memoria de una variable y manipular su contenido**.

C++ nos permite realizar cualquiera de estos pasos de forma independiente con los operadores & y *:

1. **&x** obtiene la **dirección de memoria de la variable x**.

2. ***&x** accede a esa dirección de memoria y **recupera el valor almacenado en ella**, es decir, desreferencia la dirección de x.

Ejercicio: Declara una variable float, muestre su valor y dirección de memoria

2. Punteros

- Un **puntero** es una variable que **almacena la dirección de memoria** donde se encuentra almacenado otra variable.
  Es decir, los punteros "**apuntan**" a la ubicación en la memoria de otra variable.
-

Podemos imaginar un puntero como una llave que abre un casillero.
La llave (el puntero) no tiene el contenido del casillero, sino la ubicación de ese casillero. **Al usar la llave, podemos acceder o modificar el contenido guardado en él.**

3. Declaración de un puntero

La declaración de un puntero sigue la siguiente sintaxis, en este orden:

- Tipo de dato del puntero (int, float, char, etc.).
- El operador asterisco *.
- Un nombre para el puntero.

Ejemplo:

```
int *ptr; //no inicializado
```

Buenas Prácticas:

- ✓ Inicializa puntero (al menos a nullptr).
- ✓ Usar nombres descriptivos.

Ejercicio: Declara un puntero a `char` y haz que apunte a una variable `char`. Imprima su dirección.

4. Operadores de Referencia (&) y Desreferencia (*)

- **&** obtiene la **dirección de memoria** de una variable

  Ejemplo
  ```
  int x = 3;
  int *p = &x;
  ```

- * accede al valor apuntado

  Ejemplo
  ```
  cout << *p; // imprime el valor apuntado por p;
  ```

- Además de acceder al valor, también podemos modificarlo a través del puntero desreferenciado.

  Ejemplo
  ```
  *p = 5; // modifica el valor de x a 5
          // usando el puntero
  ```

Ejercicio: Crea una función que reciba un puntero y modifique el valor original de la variable.

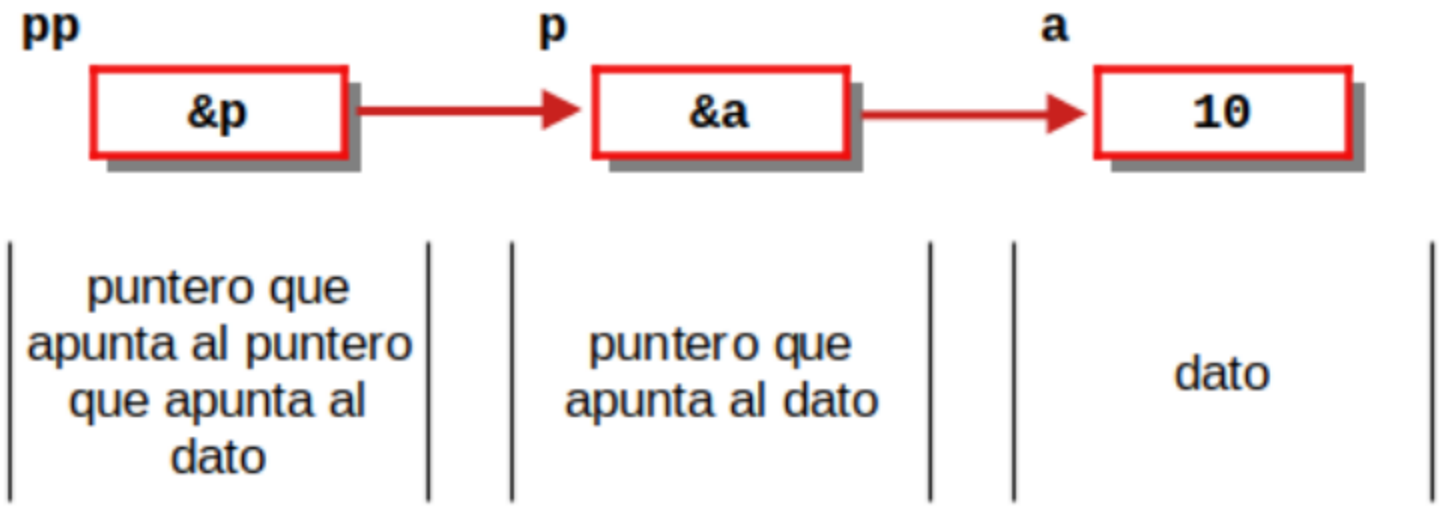
Ejercicio: Implemente una función en C++ que reciba como parámetro un **puntero a entero** y calcule el **cuadrado del valor al que apunta**.

5. Puntero a Puntero (Doble puntero)

- En C++, es posible declarar un puntero que **apunte a otro puntero**. A esto se le conoce como un **puntero a puntero**, y se declara usando **dos asteriscos (**)**.

```
int a = 5;
int *p = &a;
int **pp = &p;

cout << *p << endl;
cout << **pp << endl;
cout <<*p <<endl;
```



Ejercicio: Modifica el valor de a desde pp

6. Punteros y Cadenas de caracteres

Cuando trabajamos con cadenas de caracteres en C++, es común utilizar una **matriz de caracteres (arreglo bidimensional)** para almacenar varias cadenas.

```
char nombres[3][6] = { "Alan",
                       "Bob",
                      "Carol"};
```

0,0 char 'A'	0,1 char 'l'	0,2 char 'a'	0,3 char 'n'	0,4 char '\0'	0,5 char '\0'
1,0 char 'B'	1,1 char 'o'	1,2 char 'b'	1,3 char '\0'	1,4 char '\0'	1,5 char '\0'
2,0 char 'C'	2,1 char 'a'	2,2 char 'r'	2,3 char 'o'	2,4 char 'l'	2,5 char '\0'

```
for (int i = 0; i < 3 ; i++){
    cout << nombres[i] << endl;
}
```

¿ Qué sucede si no conocemos la longitud que los nombres tendrán? …

- Si en lugar de Carol tenemos ChristopherJones? Podemos asignar espacio adecuado (por ejemplo 20). Pero, esto provoca que se desperdicie más memoria:

0,0 char 'A'	0,1 char 'l'	0,2 char 'a'	0,3 char 'n'	0,4 char '\0'	0,5 char '\0'	0,6 char '\0'	0,7 char '\0'	0,8 char '\0'	0,9 char '\0'	0,10 char '\0'	0,11 char '\0'	0,12 char '\0'	0,13 char '\0'	0,14 char '\0'	0,15 char '\0'	0,16 char '\0'	0,17 char '\0'	0,18 char '\0'	0,19 char '\0'
1,0 char 'B'	1,1 char 'o'	1,2 char 'b'	1,3 char '\0'	1,4 char '\0'	1,5 char '\0'	1,6 char '\0'	1,7 char '\0'	1,8 char '\0'	1,9 char '\0'	1,10 char '\0'	1,11 char '\0'	1,12 char '\0'	1,13 char '\0'	1,14 char '\0'	1,15 char '\0'	1,16 char '\0'	1,17 char '\0'	1,18 char '\0'	1,19 char '\0'
2,0 char 'C'	2,1 char 'h'	2,2 char 'r'	2,3 char 'i'	2,4 char 's'	2,5 char 't'	2,6 char 'o'	2,7 char 'p'	2,8 char 'h'	2,9 char 'e'	2,10 char 'r'	2,11 char 'J'	2,12 char 'o'	2,13 char 'n'	2,14 char 'e'	2,15 char 's'	2,16 char '\0'	2,17 char '\0'	2,18 char '\0'	2,19 char '\0'

Aquí es donde los punteros permiten **gestionar memoria con mayor precisión y** trabajar con cadenas de forma más **segura y eficiente**.

Cuando utilizamos **punteros para manejar cadenas de caracteres**,
no es necesario especificar el tamaño de las columnas

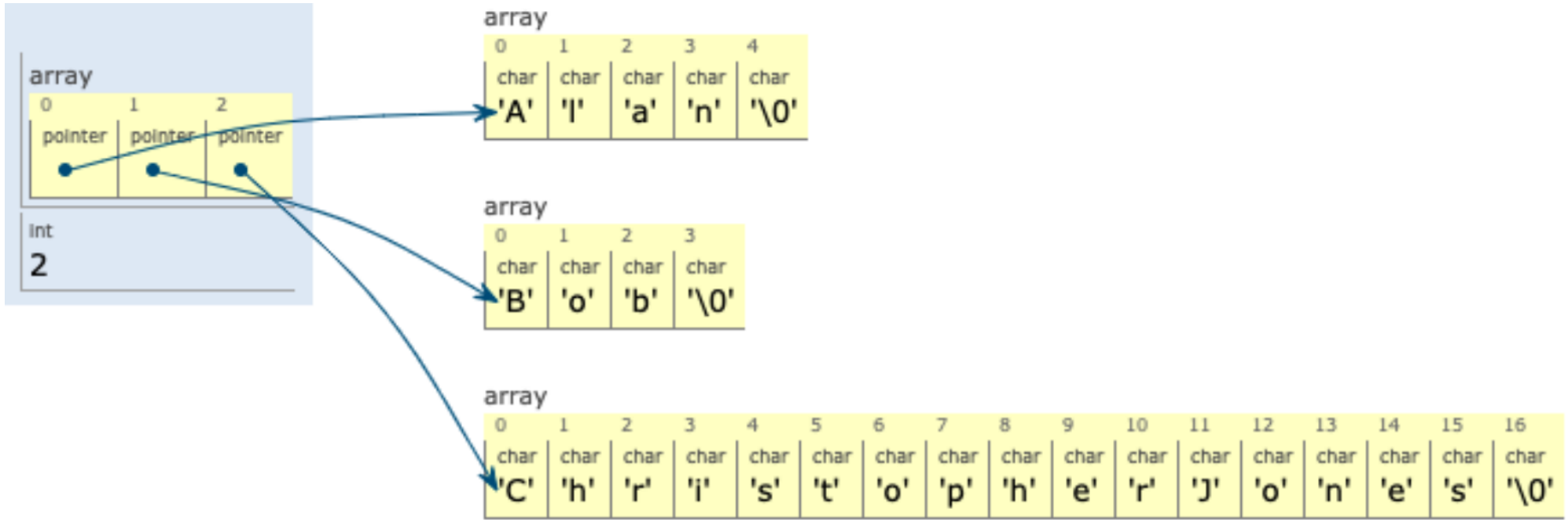
- Cada elemento del arreglo es un **puntero que apunta al primer carácter de una cadena**.

```
const char* nombres[] = { "Alan", "Bob", "ChristopherJones" };
```

En C++, cadenas literales (como `"Alan"`), estas se almacenan en una zona de **memoria de solo lectura:** `const char*` **protege esas cadenas** y le dice al compilador que **no se deben modificar**.

Ejercicio: Declare un **puntero a `char`** e **inicialícelo con una cadena literal**.
Luego, **muestre** el **valor** al que apunta el puntero.

- Note que no tuvimos que incluir ningún valor de índice (evitamos el desperdicio de memoria). Basta reservar suficiente memoria para la creación de 3 punteros.



7. Punteros nulos y buenas prácticas

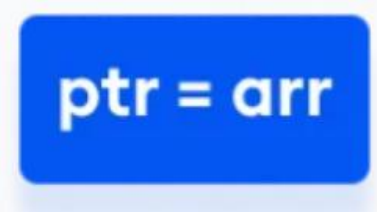
- Al igual que cualquier otra variable, los punteros deben **inicializarse correctamente**, ya sea al momento de declararlos o mediante una asignación posterior

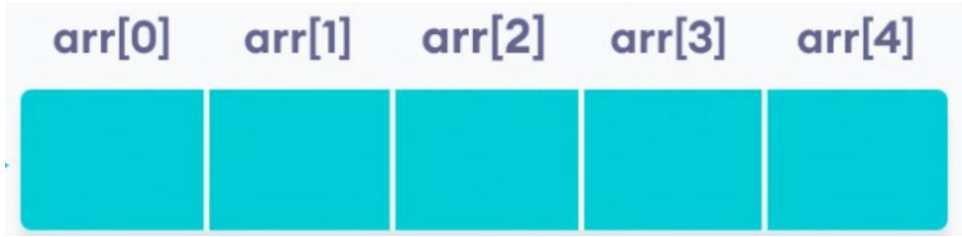
- Un puntero **nulo** es aquel que **no apunta a ninguna dirección de memoria válida**. En C++, esto se representa tradicionalmente con `0` o NULL, pero desde C++11 se recomienda utilizar **`nullptr`**, ya que es más seguro y específico para punteros.

- Intentar acceder al valor apuntado por un puntero nulo (desreferenciarlo) provoca **comportamiento indefinido.**

8. Punteros y Arreglos unidimensionales

- El **nombre de un arreglo** representa un **puntero constante al primer elemento** del arreglo.





- `arr[0]` se traduce **internamente** como: `*(arr + 0)`, es decir, **el elemento en la posición inicial del arreglo**. Por ello, en C++, los arreglos están indexados desde cero.

- Cuando se pasa un arreglo a una función, en realidad **se pasa un puntero al primer elemento**, no una copia del arreglo completo.

Ejercicio: Escribe una función que reciba un arreglo como puntero y devuelva la suma de sus elementos.

Ejemplo: relación entre punteros y arreglos

```
int vector[5] = {1, 2, 3, 4, 5};

int *pv = vector;//pv es un puntero al primer elemento

cout << vector <<endl; //devuelve la dirección del primer elemento

cout << &vector[0] << endl;// devuelve la dirección del primer elemento
```

Ejercicio: Implementar una función para inicializar y mostrar un arreglo de reales utilizando punteros. Mostrar además las direcciones de cada uno de sus elementos

9. Aritmética de Punteros

- Permite moverse entre ubicaciones en la memoria, generalmente entre elementos de un arreglo. Tenemos las siguientes **operaciones permitidas:**

- **Suma de un puntero y un entero**, agregar un número entero n a un puntero produce un nuevo puntero que avanza n posiciones en la memoria.

```
int arr [] = {6, 0, 9, 6, 5, 4};
int * ptr = arr;

ptr++; //  nos movemos al próximo elemento del arreglo (no al próximo byte)
          //en el compilador: arr + 1 x sizeof(int)

int *ptr2 = arr + 3; //mueve el prt2 para apuntar al cuarto elemento
                     //en el compilador: arr + 3 xsizeof(int)
```

- **Resta de un puntero y un entero**, restar un número entero n a un puntero, retrocede n posiciones en la memoria

- **Resta de dos punteros,** devuelve la cantidad de elementos del tipo dado que caben entre los dos punteros.
  `ptr2 – ptr1`

- **Comparación de punteros**, se pueden comparar mediante operadores relacionales como "==", "<=", ">=" y "!="

Ejercicio: Usando aritmética de punteros implemente una función para invertir un arreglo de enteros.

Intercambiabilidad entre Arreglos y Punteros

- En muchas situaciones se puede usar el nombre del arreglo como si fuera un puntero. Recíprocamente si int* ptr = arr; podemos usar p como si se tratase de un arreglo.

- `ptr[3]` puede expresarse como `*(ptr + 3)`

Ejercicio: Implemente una función que reciba un arreglo y trabaje directamente con el mismo.

10. Eficiencia de los Punteros

- Ayudan a evitar copias innecesarias, permiten la manipulación directa de datos, se usa con frecuencia en estructuras de datos complejas

- Acceder a los elementos de un arreglo utilizando punteros en lugar de los índices usuales es más eficiente.

```
double data [10] = {0}, a;
data[3] = 3.14;
a = data[3];
```

```
double data [10] = {0}, a;
*(data + 3) = 3.14;
a = *(data + 3);
```

  - Para calcular la posición de memoria de data[3] el compilador:
    1. Lee el valor de la dirección en la variable data.
    2. Calcula el incremento correcto (3 x 8 bytes)
    3. Agrega el incremento a data

En ambos casos tenemos el mismo número de operaciones ☹

```
double data [100];
int i;
for (i = 0; i < 100; i++) {
    if (data[i] < 0.) {
    cout <<"valor neg:\n " <<data[i];
    }
}
```

- Requiere calcular la dirección de data[i] para cada if:
  - 100 sumas: data + i
  - 100 multiplicaciones: i x 8 bytes.

```
double data [100], *pd , * lastpd;
lastpd = data + 100;
for (pd = data; pd < lastpd; pd++) {
    if (*pd < 0.) {
        cout << "Valor neg:\n" << *pd;
    }
}
```

- No necesitamos calcular en ninguno de los if para acceder al elemento actual *pd del arreglo
- La variable auxiliar lastpd sirve para evitar calcular 100 veces la suma data + 100 en la condición para salir del ciclo for.

Utilizando punteros obtenemos menor número de operaciones ☺

11 Referencias vs punteros

- En C y C++, se utiliza **&** para **obtener la dirección de memoria** de una variable. Sin embargo, en C++, el mismo símbolo & adquiere un **significado adicional cuando se utiliza en una declaración**: sirve para declarar una **referencia**, es decir, un **alias** para una variable existente.

- Ejemplos:

```
int y;
int &x = y; //x es una referencia (alias) de y

int jose = 5;
int &pepe = jose; //pepe es un alias de jose

void f(int &x){...}      f(y)
```

Aunque ambos permiten acceder indirectamente a otras variables, tienen **diferencias importantes en su uso y comportamiento**:

1. **Las referencias están implícitamente desreferenciadas**
   - ✓ No es necesario usar el operador * para acceder al valor de una referencia.
   - ✓ Se comportan como un alias directo de la variable original.

2. **Las referencias no se pueden reasignar**
   - ✓ Una vez inicializada, una referencia no puede cambiar para referenciar otra variable.
   - ✓ Un puntero sí puede apuntar a diferentes ubicaciones a lo largo del programa.

3. **Las referencias deben inicializarse al declararse**
   - ✓ No puedes declarar una referencia sin asignarla a una variable existente.
   - ✓ Los punteros pueden declararse sin inicialización (aunque no es recomendable).

Ejemplos: Punteros vs referencias C / C++

```
//Los punteros pueden ser
//inicializados posteriormente
int *ptr_i; // Declaración
ptr_i = &i; // Inicialización
```

```
//Las referencias deben inicializarse al
//momento en que se crean
int i = 4;
int &ref_i = i; // "ref_i" es el alias de la
                // variable "i"
int &ref_j; // ¡ERROR! A "ref_j" se le debe
            // asignar alguna variable
```

```
//Los punteros pueden cambiar de valor
//en cualquier momento
int *ptr_n = &x; // "ptr_n" apunta a la
variable "x"
ptr_n = &y;
// Ahora "ptr_n" apunta a la variable "y
```

```
//Las referencias no pueden
//reasignarse a otra variable
int x = 1;
int y = 2;
int &ref_n = x;
// "ref_n" es un alias de "x"
ref_n = y;
//¡ERROR! "ref_n" está unido de por vida a "x"
```

```
//Las referencias no pueden almacenar un puntero nulo
int *ptr = nullptr;
int &reference = nullptr // ¡ERROR!
```

Ejercicios

- Dado el arreglo `int numeros[5]{11, 22, 33, 44, 55};` imprimir sus elementos utilizando punteros

- Para `int numeros[8]{11, 22, 33, 44, 55, 66, 77, 88};` defina 2 punteros a algún elemento del arreglo y calcule su diferencia (número de elementos entre dos punteros)

- Para `int numeros[8]{11, 22, 33, 44, 55, 66, 77, 88};` defina 2 punteros a algún elemento del arreglo y compárelos utilizando los operadores relacionales

- Implemente dos funciones: una que use referencia y otra puntero, ambas deben modificar el valor original.

Resumen:

- El operador * se utiliza de dos maneras diferentes:
  1. **Declaración un puntero**, * se coloca después del tipo y antes del nombre de la variable. `int *ptr;`
  2. **Desreferencia**, * se coloca antes del nombre del puntero para desreferenciarlo, para acceder o modificar el valor al que apunta.
     `*ptr;` `*ptr = 4;`

- De forma similar el operador &, puede usarse como:
  1. **Referencia**, para indicar un tipo de dato por referencia
     `int &x = y;`
  2. **Dirección**, para tomar la dirección de una variable
     `int *ptr = &x;`

- Los punteros, mediante el uso de aritmética de punteros, permiten:
  - recorrer manualmente arreglos,
  - controlar directamente la memoria,
  - acceder de forma eficiente a estructuras de datos,
  - implementar estructuras dinámicas (listas, árboles, etc.),
  - manipular datos sin copiar grandes bloques de memoria y
  - optimizar el rendimiento en operaciones de bajo nivel.

- Sin embargo, su uso incorrecto puede provocar errores graves, como:
  - acceso fuera de los límites del arreglo,
  - punteros colgantes (dangling pointers),
  - desreferenciación de punteros nulos (nullptr),
  - modificaciones involuntarias de datos y
  - dificultad para depurar errores relacionados con memoria.