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 5:

Punteros II

Contenido

1. Paso de punteros como parámetros de una función.
   - ✓Paso por valor, por punteros, por referencia
2. Arreglo de punteros
3. Punteros a punteros
4. Punteros y matrices
5. Prioridad de los operadores *, () y []
6. Puntero void, puntero constante
7. Punteros inteligentes (Smart pointers)

1. Paso de punteros como parámetros de una función.

- **Paso por valor:** Copia el valor. No modifica el original.
- **Paso por puntero:** Permite modificar el valor original.
- **Paso por referencia:** Similar al puntero, pero con sintaxis más clara

Ejemplos:

```
void fporValor(int n);

void fporPuntero(int* n);

void fporReferencia(int& n);
```

2. Arreglo de punteros

- Es una colección de direcciones de memoria (punteros); por ejemplo

```
int *a[5];
char *b[10];
const char *frutas[] = {"Manzana", "Fresa", "Cereza"};
```

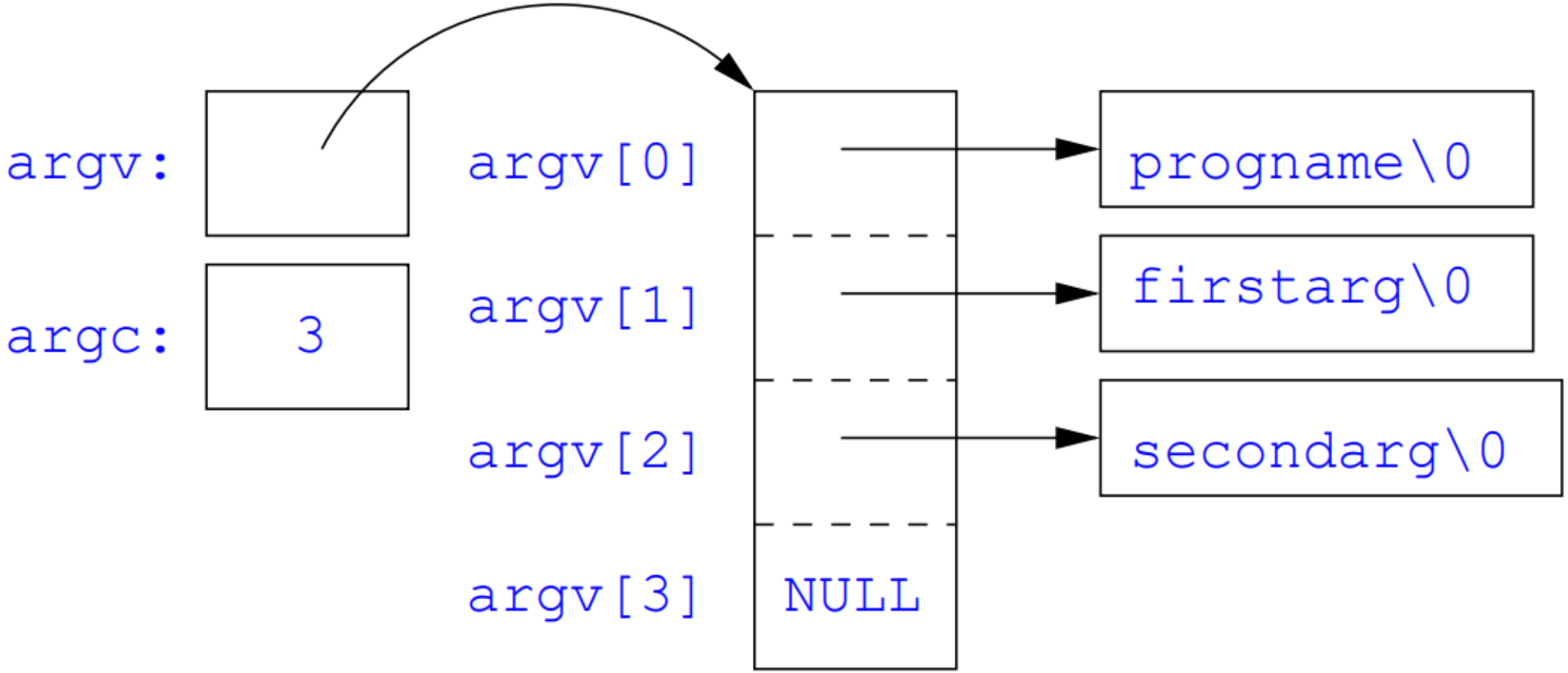
- Los arreglos de punteros son útiles con cadenas. Ejemplo: main con parámetros

```
int main(int argc, char *argv[]){//o char **argv
    //....
    return 0;
}
```

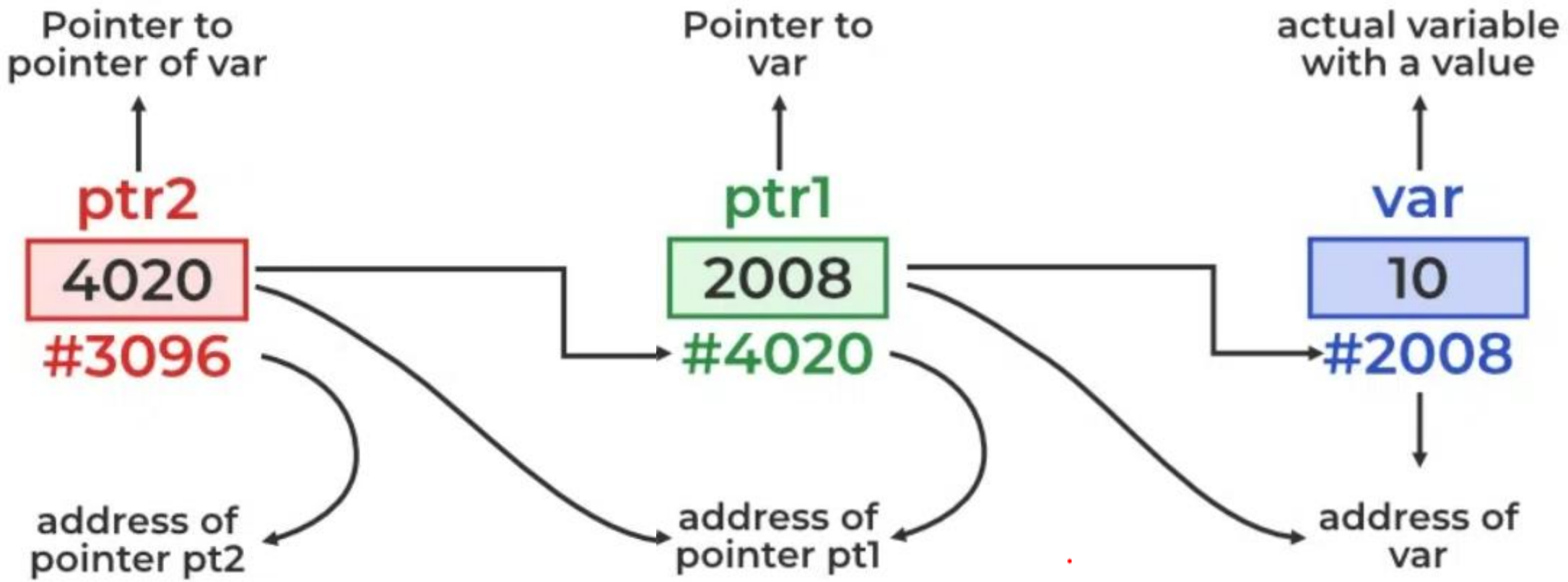
El primer argumento es el número de argumentos de línea de comandos y
El segundo es una lista de argumentos de línea de comandos

Ejemplo: arreglo de punteros

- argv es un arreglo de punteros a caracteres
- argc le indica al programador la longitud del arreglo



3. Punteros a punteros



Ejemplo: Punteros a punteros

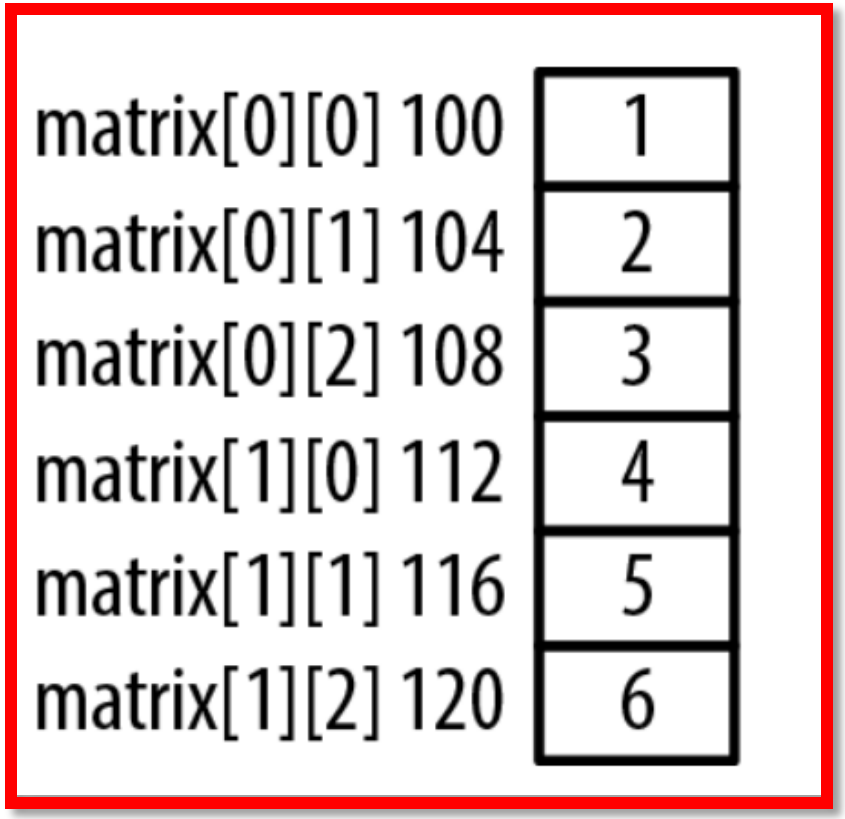
```
int arr[3];
int *ptr,**pptr; // declaración de punteros
arr[0] = 10;
ptr = arr; //
pptr = &ptr;//

cout << &arr[0] <<endl;//
cout << *&ptr <<endl; //
cout << *pptr <<endl; //    *pptr == *&ptr == ptr
cout << **pptr <<endl; // **pptr==*(*pptr)==*(ptr)==arr[0]
cout << arr <<endl; //
cout << &arr <<endl; //
```

4. Punteros y matrices

```
int matrix[2][3] = {{1,2,3},{4,5,6}};
```

Un arreglo bidimensional (matriz) **se trata como un arreglo de arreglo**,

Row	Column 0	1	2
0	1	2	3
1	4	5	6



Relación entre punteros y matrices

Dado la matriz: `int a[2][4] = {{1,2,3,4},{5,6,7,8}};`

- a **es la dirección del primer subarrego** // a == &a[0] tipo: int(*)[4].

- a[0] es la primera fila, // a[0] == &a[0][0] tipo int[4]

- a + 1 internamente se traduce en sumar el tamaño del subarreglo (4 * sizeof(int)) // a + 1 == &a[1] tipo: int(*)[4]

- *(a + 1) desreferencia el puntero a + 1, accediendo a a[1] (tipo int[4]) // *(a + 1) == a[1]

- `*(a+1) + 1;` apunta al segundo elemento del subarreglo a[1]
  // *(a + 1) + 1 ==&a[1][1] es un puntero a int

- * (*(a + 1) + 1); desreferenciamos la dirección anterior
  // * (*(a + 1) + 1) == a[1][1] == 6

Expresión	**Equivalente**	**Significado**
a	&a[0]	Dirección del primer subarreglo (int (*)[4])
a[0]	&a[0][0]	Dirección del primer entero (int*)
a + 1	&a[1]	Dirección del segundo subarreglo
*(a + 1)	a[1]	Segundo subarreglo (int[4])
*(a + 1) + 2	&a[1][2]	Dirección del 3er elemento de 2da fila
((a + 1) + 2)	a[1][2]	Valor: 7

En general dada una matriz int a[N1][N2], tenemos las siguientes formas equivalentes de desreferenciar uno de sus elementos a[i][j]:

1. `*(*(a+i) + j)`

2. `*(a[i] + j)`

3. `*(&a[0][0] + N2 * i + j)`

Expresiones como: a[i] == *(a + i) == *(i + a) == i[a]
son equivalentes y tienen sentido en C++.

Ejemplo

```
#include <iostream>
using namespace std;

int main() {
    int a[3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8},{9, 10, 11, 12} };

    // Mostrar dirección
    cout << "Direccion de a        : " << a << endl;
    cout << "Direccion de a[0]     : " << a[0] << endl;
    cout << "Direccion de a + 1    : " << (a + 1) << endl;
    cout << "Direccion de a[1]     : " << a[1] << endl;

    // Restar dos punteros
    cout << "\n(a + 1) - a = " << (a+1) - a << " (filas de distancia)" << endl;

    return 0;
}
```

Pasando matrices a funciones

- Al pasar una matriz bidimensional a una función, la primera dimensión (filas) no es obligatoria, ya que lo que realmente se pasa es un puntero al primer subarreglo (fila).Por eso, las siguientes formas son equivalentes y válidas:

```
void f(int a[5][10]) { ... } //Paso filas y columnas

void f(int a[][10]) { ... } // equivale a int(*a)[10]

void f(int (*a)[10]) { ... }//Es lo que sucede internamente


void f(int **a) { ... }//Válido sólo para matrices creadas como
                       arreglo de punteros
```

Punteros a funciones

- El **nombre de una función representa la dirección de inicio del código** de esa función en memoria; esto significa que **podemos asignar el nombre de una función a un puntero a función** y luego **invocar esa función a través del puntero**.
- Un puntero a función puede almacenar la dirección de una función con una firma específica (mismo tipo de retorno y los mismos parámetros).
- **Declaración**:
```
int (*f) (int, int);
//f es un puntero a una función que recibe dos int y retorna un int.
```
- Note que:
```
int *f(int, int);
```
Es una declaración de **una función que retorna un puntero a int**

Ejercicio: escriba un programa que calcule el máximo de 2 números utilizando puntero a función y una función que retorne puntero

5. Prioridad de los operadores *, () y []

Las reglas de precedencia para las declaraciones, de mayor a menor, son:

1. Paréntesis () que agrupan partes de una declaración. Se utilizan para forzar un orden específico y evitar ambigüedad

2. Operadores de sufijo (mayor precedencia que *)
   - ✓ [] Acceso a un elemento de un arreglo
   - ✓ () Llamada a función.

3. Operador de prefijo
   - ✓ * Operador de desreferenciación (puntero a)

Ejemplo

```
char* (*(*f[5])(char*))[];

//f es un

//arreglo de 5 punteros

//a funciones

//que aceptan un puntero a char

//y retornan un puntero

//a un arreglo de punteros

//a char
```

6. Punteros void

- Denominado también **puntero genérico**: **Declaración**: `void *ptr;`
- Puede almacenar la dirección de cualquier tipo de dato sin tener que hacer una conversión explícita (cast) y podemos convertirlo a cualquier tipo de dato en caso sea necesario.
- No podemos operar (desreferenciar) con el objeto señalado por el puntero void, ya que se desconoce el tipo.
- Tampoco podemos hacer aritmética de punteros.

Ejemplo: Punteros void

```
void *ptrVoid;
int a = 78, arr[20];
float r = 283.91, *ptrFloat;
char nombre[30] = "Javier";
ptrVoid = &a;// recibe la dirección de a
ptrVoid = arr;// apunta al primer elemento de arr
ptrVoid = &arr[5];// recibe la dirección de arr[5]
ptrVoid = &r;// recibe la dirección de r

(void*)nombre;
//para lograr imprimir la dirección

*ptrFloat = 456.78;
ptrVoid = ptrFloat;  //apunta a la misma dirección que ptrFloat

ptrVoid = nombre;//recibe la dirección de la cadena nombre
```

6. Tipos de Punteros con const

const se usa para indicar que algo no debe modificarse. Existen 3 combinaciones:

1. **Puntero a dato constante**: `const int *ptr;` o `int const *ptr`
   Los datos señalados NO SE PUEDEN cambiar.

```
int i1 = 8, i2 = 9;
const int * iptr = &i1; //int const *iptr
//*iptr = 9; error: assignment of read-only location
iptr = &i2;  // ok
```

2. **Puntero constante a dato**: `int *const ptr;`
   Los datos apuntados PUEDEN cambiarse; pero el puntero NO SE PUEDE cambiar para que apunte a otros datos.

```
int i1 = 8, i2 = 9;
int * const iptr = &i1; // puntero constante
//debe ser inicializado durante la declaración
*iptr = 9;   // ok
// iptr = &i2; error: assignment of read-only variable
```

3. **Puntero constante a dato constante**: `const int *const ptr;` los datos señalados NO SE PUEDEN cambiar; y el puntero NO SE PUEDE cambiar para que apunte a otros datos:

```
int i1 = 8, i2 = 9;
const int * const iptr = &i1;
// *iptr = 9;   // error: assignment of read-only variable
// iptr = &i2;  // error: assignment of read-only variable
```

Ejemplo: El nombre de un arreglo es un puntero constante

- Cuando declaramos un arreglo como: `int A[n];` el nombre del arreglo se comporta como un puntero constante al primer elemento

```
// A es equivalente a: int* const


int A[5] = {1, 2, 3, 4, 5};
int* ptr = A;  // Válido
ptr = ptr + 1; // Válido: podemos mover ptr
A = ptr;       //  Error
```

- Los arreglos tienen tamaño fijo. Se les asigna memoria en tiempo de compilación (en el stack). Al salir del bloque donde fueron definidos, la memoria se libera automáticamente.

7. Punteros inteligentes

- Es un mecanismo de C++ para resolver el problema de la gestión de memoria (desde la versión C++ 11 en adelante).

- El problema es que para alojar memoria en el heap debemos utilizar la palabra new y eliminar explícitamente las variables con el operador delete.

- Los punteros inteligentes permiten eliminar de la memoria automáticamente cuando el nombre de la variable sale de su ámbito.

- La biblioteca <memory> ofrece distintos tipos, lo más comunes son:
  - ✓ Punteros únicos std::unique_ptr
  - ✓ Punteros compartidos std::shared_ptr
  - ✓ Punteros débiles std::weak_ptr

Más sobre Smart pointers

- https://docs.hektorprofe.net/cpp/12-punteros-inteligentes/
- https://en.wikipedia.org/wiki/Smart_pointer
- https://www.geeksforgeeks.org/smart-pointers-cpp/
- https://youtu.be/8gUz60-GtPA

Ejercicio: primera parte

Escribir un programa que lea un texto, luego proceda a dividir el texto en palabras (secuencia de caracteres separados por un espacio en blanco) y almacene estas palabras en un arreglo. Realizado el almacenamiento proceda a ordenar las palabras en orden alfabético. Sugerencia:

I. Obtenga los caracteres uno por uno y verifique si se ha llegado al final de una cadena.
II. Cargue los caracteres en una matriz bidimensional cuyo primer índice (filas) cuenta las palabras y el segundo (columnas) los caracteres que componen una sola palabra.
III. Aplique un algoritmo de ordenamiento adecuado (burbuja, selección, inserción, merge sort, quick sort), para ordenar las palabras.

Ejemplo:

Input: bienvenidos al curso cc112
Debe almacenarse en una matriz y mostrarse en la forma:
bienvenidos
al
curso
cc112

Output:
al
bienvenidos
cc112
curso

Segunda parte

Escribir una segunda versión del ejercicio que involucre el uso de todas las herramientas vistas en el curso:

funciones, algoritmos de ordenamiento para arreglo de y punteros.

Detalles técnicos

- En int a[3]; a técnicamente no es un puntero. a es el nombre del arreglo, pero en la mayoría de las expresiones decae (se convierte implícitamente) en un puntero al primer elemento del arreglo. Por ejemplo, en asignaciones o al pasar el arreglo a a una función

- El nombre del arreglo es una dirección fija y constante (no puede ser cambiado para apuntar a otro lugar).
  Un puntero normal puede ser modificado para apuntar a cualquier otra dirección.

- Para que un puntero apunte a un arreglo completo de 3 elementos, debe declararse explícitamente como int (*ptr)[3]

Detalles técnicos

- Ejemplos donde el nombre de un arreglo no decae en un puntero a su primer elemento

  1. El operador sizeof(a) no decae en un puntero al primer elemento

```
int a[5] = {1, 2, 3, 4, 5};
cout << sizeof(a) <<endl; // Devuelve 20 si cada entero ocupa 4 bytes.
cout << sizeof(int*) <<endl; // tamaño de un puntero

//&a  NO decae, Puntero al arreglo completo ie dirección del arreglo completo
//a  se convierte en un puntero al primer elemento
//&a[0] dirección del primer elemento
```

  2. El operador & aplicado a un arreglo no decae a un puntero al primer elemento, sino que devuelve la dirección del arreglo como un todo.

```
int (*ptr1)[5] = &a;  // Puntero al arreglo de 5 elementos
int *ptr2 = a;  // Puntero al primer elemento
```

Detalles técnicos

3. Si tienes una referencia a un arreglo, el nombre del arreglo no decae a un puntero, ya que estás refiriéndote al arreglo completo.

```
  int arr[5];
  int (&ref)[5] = arr;  // ref es una referencia al arreglo completo, no un puntero
```

4. Cuando pasas un arreglo a una función como referencia, el nombre del arreglo no decae a un puntero al primer elemento, sino que la referencia se trata como un tipo completo de arreglo.

```
void funcion(int (&arr)[5]) {
    cout << sizeof(arr);  // Devuelve el tamaño del arreglo completo (20 bytes)
}

int main() {
    int a[5] = {1, 2, 3, 4, 5};
    funcion(a);  // Pasa el arreglo por referencia, no como puntero.
    return 0;
}
```

Detalles técnicos

5. En arreglos multidimensionales, el nombre del arreglo no decae inmediatamente a un puntero al primer elemento del primer subarreglo; en su lugar, decae progresivamente a punteros a subarreglos

```
int c[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
int (*ptr)[4] = c;  // ptr es un puntero a un arreglo de 4 enteros

cout << (*ptr)[1];  // Imprime 2, el segundo elemento del primer subarreglo
```

En este caso, c decae a un puntero a un arreglo de 4 enteros (int (*)[4]), no a un puntero a un puntero (int**).