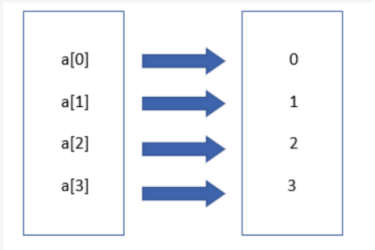
# Arreglos y punteros usando C++

## 1. Arreglos Unidimensionales




**Definición:**

Un arreglo unidimensional (también llamado vector) es una estructura de datos que almacena una secuencia de elementos del mismo tipo, accesibles mediante un índice.

**Declaración:**

```
tipo nombreArreglo[tamaño];
```

```
int n;
cout << "Ingrese la cantidad: ";
cin>>n;
int numeros[n]{};
char caracteres[5]{};
```

**Inicialización:**

```
int numeros[5] {1, 2, 3, 4, 5};
/*otra forma de inicializar*/
int meses[5]{1, 2, 3};
char caracteres[5] {'a', 'b', 'c', 'd', 'e'};
En el ejemplo anterior meses[3] y meses[4] se llenan con ceros.
```

**Gestión de Memoria:**

- Al declarar un arreglo, se reserva un bloque contiguo de memoria para almacenar los elementos.
- Cada elemento del arreglo ocupa un espacio en la memoria dependiendo del tipo de dato (por ejemplo, un `int` generalmente ocupa 4 bytes).
- Ocupa un espacio finito de memoria definido por el tamaño que se le coloca al inicializar

**Ejemplo de Vida Real:**

- Fila para comprar boletos en cine

**Ejemplo de Código:**

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

int main() {
    int numeros[5] {10, 20, 30, 40, 50};

    // Acceso a los elementos
    for(int i = 0; i < 5; i++) {
        cout << "Elemento en el índice " << i << ": "
             << numeros[i] << endl;
    }

    return 0;
}
```

**Paso por Parámetros:**

```
void imprimirArreglo(int * arr, int tamano) {
    for(int i = 0; i < tamano; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}
```

## 2. Arreglos Bidimensionales


**Definición:**

Un arreglo bidimensional (también conocido como matriz) es una estructura de datos que organiza elementos en filas y columnas.

**Declaración:**

```
tipo nombreArreglo[filas][columnas];
```

Ejemplo:

```
int matriz[3][3];
```

**Inicialización:**

```
int matriz[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

**Gestión de Memoria:**

- En un arreglo bidimensional, la memoria también se asigna de manera contigua, pero en este caso, se organiza en función de las filas y columnas.
- En memoria, se almacena primero una fila completa, luego la siguiente, y así sucesivamente.
- En la memoria se ponen todos los elementos uno a continuación que del otro y el sistema debe hacer cálculos complicados para llegar al elemento [i][j].

**Ejemplo de Vida Real:**

- Un tablero de ajedrez
- Un salón de clases
- Cartón de bingo

**Ejemplo de Código:**

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

int main() {
    int matriz[2][3] = {{1, 2, 3}, {4, 5, 6}};

    // Acceso a los elementos
    for(int i = 0; i < 2; i++) {
           cout << "[" << i <<
                    "][" << j << "]:" << matriz[i][j] << " ";
        }
        cout << endl;
    }
```

```
    return 0;
}
```

**Paso por Parámetros:**

```
void imprimirMatriz(int matriz[][3], int filas) {
    for(int i = 0; i < filas; i++) {
        for(int j = 0; j < 3; j++) {
            cout << matriz[i][j] << " ";
        }
        cout << endl;
    }
}
```

En C++, cuando trabajas con matrices (arrays bidimensionales) y deseas pasarlas como parámetros a una función, generalmente necesitas especificar el número de columnas. Esto se debe a cómo se gestionan los arrays multidimensionales en memoria y cómo el compilador accede a los elementos.

## Concepto de Matriz en C++

Una matriz en C++ es esencialmente un array de arrays. Por ejemplo, una matriz `int matriz[3][4]` tiene 3 filas y 4 columnas, lo que significa que es un array de 3 elementos, cada uno de los cuales es a su vez un array de 4 enteros.

## Razón para Especificar Columnas

Cuando pasas una matriz como parámetro a una función, lo que realmente estás pasando es un puntero al primer elemento del array (el cual es un array unidimensional). Sin embargo, para que el compilador sepa cómo navegar por la matriz, necesita conocer la cantidad de columnas, ya que esta información es necesaria para calcular la ubicación de cualquier elemento en memoria.

**Acceso a Elementos en una Matriz**

Para entenderlo mejor, considera cómo se almacenan los elementos de una matriz en memoria:

1. **Matriz 2D:** `int matriz[3][4];`
2. **Memoria:** La matriz se almacena en memoria de manera continua (fila por fila).

La memoria se verá algo así (considerando que los elementos son enteros):

css
Copiar código
`matriz[0][0] matriz[0][1] matriz[0][2] matriz[0][3]`

```
matriz[1][0] matriz[1][1] matriz[1][2] matriz[1][3]
matriz[2][0] matriz[2][1] matriz[2][2] matriz[2][3]
```

Para acceder a `matriz[i][j]`, el compilador necesita saber cuántos elementos hay en cada fila (número de columnas) para calcular la dirección en memoria:

**Dirección de `matriz[i][j]`**: Dirección base+ *i×número de columnas + j*

Cómo lo entendemos los programadores

| | 0 | 1 | 2 |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 1 | 4 | 5 | 6 |
| 2 | 7 | 8 | 9 |

Cómo lo entiende la computadora

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0x3+0 | 0x3+1 | 0x3+2 | 1x3+0 | 1x3+1 | 1x3+2 | 2x3+0 | 2x3+1 | 2x3+2 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

## Ejemplo de Pasar Matriz como Parámetro

Cuando defines una función que recibe una matriz, debes especificar al menos el número de columnas:

cpp
Copiar código

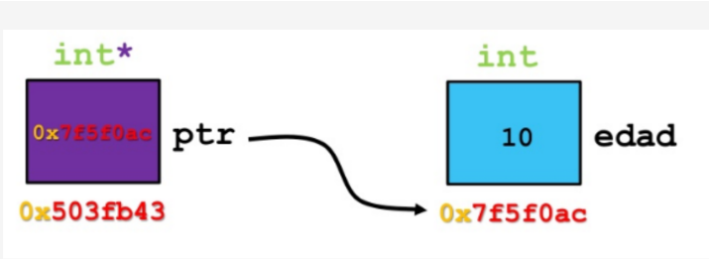
```cpp
void imprimirMatriz(int matriz[3][4], int filas, int columnas) {
    for (int i = 0; i < filas; ++i) {
        for (int j = 0; j < columnas; ++j) {
            std::cout << matriz[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
```

Aquí, el número de columnas (4) es necesario para que el compilador pueda calcular correctamente la dirección de cada elemento en la matriz.

### ¿Por Qué No Necesitas Especificar las Filas?

- Cuando pasas la matriz, la dirección base es suficiente para navegar por las filas porque las filas están dispuestas secuencialmente en memoria.
- Pero para moverse dentro de una fila (acceder a columnas), necesitas conocer el número de columnas para saber cuántos elementos saltar.

## 3. Punteros




**Definición:**

Un puntero es una variable que almacena la dirección de memoria de otra variable.

**Declaración:**

```
tipo * nombrePuntero;
```

Ejemplo:
```
int* p;
```

**Inicialización y uso:**

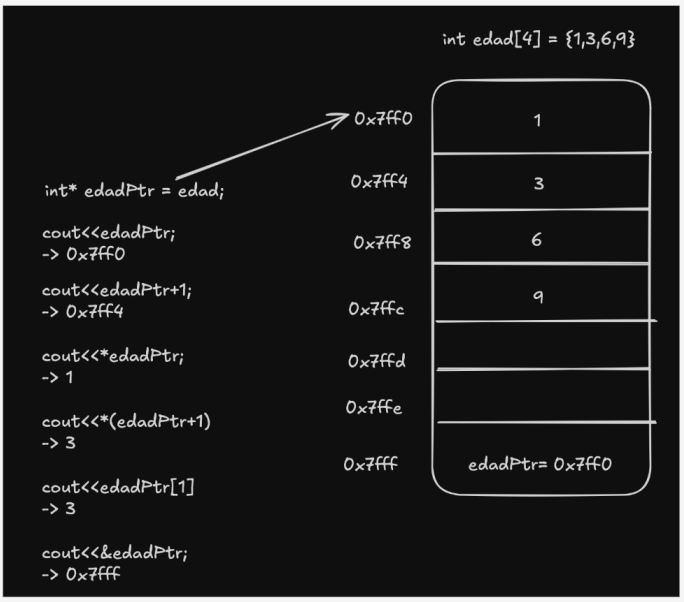
```
int x = 10;
int* p = &x;  // p ahora almacena la dirección de memoria de x
cout << "Valor de x: " << x << endl;
cout << "Dirección de x: " << p << endl;
cout << "Valor en la dirección a la que apunta p: " << *p << endl;
```

**Gestión de Memoria:**

- Un puntero ocupa un espacio en la memoria, generalmente 4 u 8 bytes (dependiendo del sistema) para almacenar una dirección.

- Los punteros permiten acceder y manipular datos almacenados en diferentes ubicaciones de memoria, lo cual es crucial para la gestión eficiente de la memoria en programas complejos.

¿Cómo funciona las variables y la memoria?




**Ejemplo de Código con Arreglos:**

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

int main() {
    int numeros[5] {1, 2, 3, 4, 5};
    int* p = numeros;  // Puntero al primer elemento del arreglo

    for(int i = 0; i < 5; i++) {
        cout << "Elemento " << i << ": " << *(p + i) << endl;
    }

    return 0;
}
```

## 4. Conclusión y Ejercicios Prácticos

**Ejercicio 1:**

Crear un programa que lea un arreglo unidimensional de 10 enteros, los ordene y luego imprima los valores ordenados.

**Ejercicio 2:**

Escribir un programa que llene una matriz de 3x3 con números enteros ingresados por el usuario y luego calcule la suma de los elementos de la diagonal principal.

**Ejercicio 3:**

Implementar un programa que use punteros para intercambiar los valores de dos variables.

**Acceso a los Elementos:** En ambos casos, puedes acceder a los elementos del arreglo usando la sintaxis de índice (`p[i]`) o la notación de puntero (`*(p + i)`).

Ejemplo:

```
void imprimirArreglo(int* p, int tamaño) {
    for(int i = 0; i < tamaño; i++) {
        cout << p[i] << " ";  // Usando índice
        cout << *(p + i) << " ";  // Usando notación de puntero
    }
    cout << endl;
}
```

## Diferencias:

1. **Especificidad:**
    - `int* p`: Indica explícitamente que `p` es un puntero a un entero.
2. **Información sobre el Tamaño:**
    - En `int p[]`, la información de que `p` es un arreglo puede sugerir al programador que está trabajando con una secuencia de elementos, aunque el compilador lo trate como un puntero.
    - En `int* p`, es más general y no proporciona ninguna información sobre el tamaño del arreglo al que apunta, lo cual puede hacer que el código sea menos intuitivo si no se maneja correctamente.

## Ejemplo Práctico:

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

void procesarPuntero(int* p) {
    cout << "Procesando como puntero: ";
    for(int i = 0; i < 3; i++) {
        cout << p[i] << " ";
    }
    cout << endl;
}
```

```
void procesarArreglo(int p[]) {
    cout << "Procesando como arreglo: ";
    for(int i = 0; i < 3; i++) {
        cout << p[i] << " ";
    }
    cout << endl;
}

int main() {
    int numeros[3] = {1, 2, 3};

    procesarPuntero(numeros);  // Llamada con puntero
    procesarArreglo(numeros);  // Llamada con arreglo

    return 0;
}
```

En este ejemplo, ambas funciones `procesarPuntero` y `procesarArreglo` se comportan de la misma manera, aunque el nombre del parámetro sugiere un propósito ligeramente diferente.

### Conclusión:

- **Funcionalmente**: En el contexto de parámetros de función, `int p[]` y `int* p` son casi equivalentes
- **Semánticamente**: Usar `int p[]` puede ser más informativo para indicar que esperas un arreglo, mientras que `int* p` es más general y podría referirse a un solo entero o a un arreglo.

Por tanto, la elección entre ambos depende más de la claridad que deseas transmitir en tu código.

## Diferencia entre memoria estática y memoria dinámica

La **memoria estática** y la **memoria dinámica** son dos formas de gestionar la memoria en un programa. Entender la diferencia entre ambas es crucial para escribir código eficiente y seguro en lenguajes como C y C++. A continuación, se explican las principales diferencias entre memoria estática y memoria dinámica.

### Memoria Estática:

1. **Asignación y Liberación:**

- La memoria estática se asigna en tiempo de compilación. Esto significa que el tamaño y la cantidad de memoria que el programa va a usar se determinan antes de que el programa comience a ejecutarse.
- La liberación de memoria estática ocurre automáticamente al final del programa o al salir del bloque en el que fue definida la variable.

2. **Almacenamiento:**
   - Se almacena en la **pila de ejecución** (stack) para variables locales o en la **segmento de datos** para variables globales y estáticas.
   - Las variables tienen un tiempo de vida predeterminado: las variables locales existen mientras el programa está en el bloque donde fueron declaradas, y las globales o estáticas existen durante toda la ejecución del programa.

**Ejemplo:**

```
int main() {
    int numeros[10];  // Memoria estática asignada en la pila
    // ...
    return 0;
}
```

3. **Ventajas:**
   - Rápida y sencilla de manejar.
4. **Desventajas:**
   - El tamaño de las estructuras de datos debe ser conocido y fijado en tiempo de compilación, lo que puede llevar a desperdiciar memoria si se asigna más de la necesaria o a limitaciones si se asigna menos.

## Memoria Dinámica:

1. **Asignación y Liberación:**
   - La memoria dinámica se asigna en tiempo de ejecución, es decir, mientras el programa está corriendo. El tamaño de la memoria y su cantidad pueden cambiar según las necesidades del programa.
   - La liberación de memoria dinámica debe realizarse manualmente por el programador, utilizando funciones como `delete` en C++ o `free` en C.
2. **Almacenamiento:**
   - Se almacena en el **montón** (heap), una región de memoria más grande y flexible que la pila.

**Ejemplo:**

```
int main() {
    int* numeros = new int[10];  // Memoria dinámica asignada en el
montón
    // Uso del arreglo
```

```
    delete[] numeros;  // Liberación de la memoria
    return 0;
}
```

3. **Ventajas:**
    - Flexibilidad en el uso de la memoria, ya que puedes asignar y liberar memoria según sea necesario durante la ejecución del programa.
    - Permite crear estructuras de datos cuyo tamaño no necesita conocerse hasta que el programa esté en ejecución.
4. **Desventajas:**
    - La gestión manual de la memoria puede llevar a mayor fugas de memoria por el uso incorrecto.

## Comparación Resumida:

| Característica | Memoria Estática | Memoria Dinámica |
|---|---|---|
| **Momento de asignación** | Tiempo de compilación | Tiempo de ejecución |
| **Almacenamiento** | Pila de ejecución o segmento de datos | Montón (heap) |
| **Liberación de memoria** | Automática | Manual, requiere uso de `delete` o `free` |
| **Flexibilidad** | Tamaño fijo y determinado | Tamaño flexible y determinado en ejecución |
| **Velocidad** | Más rápida | |
| **Riesgo de errores** | Menor | Mayor (fugas de memoria, uso incorrecto) |

## Conclusión:

- **Memoria estática** es adecuada cuando conoces de antemano el tamaño y la cantidad de datos que necesitas y quieres aprovechar la simplicidad y la velocidad de la pila.
- **Memoria dinámica** es esencial cuando necesitas flexibilidad para manejar *estructuras de datos* cuyo tamaño puede cambiar durante la ejecución del programa, pero requiere una gestión cuidadosa para evitar errores comunes.