



# UNIDAD 1

PROGRAMACIÓN IMPERATIVA



## ÍNDICE

|  |    |
|--|----|
| 1. Estructuras de Control .....                                    | 2  |
| 1.1. If.....   | 2  |
| 1.2. Switch .....  | 3  |
| 1.3. For.....  | 6  |
| 1.4. While.....  | 9  |
| 1.5. Do- while.....  | 12 |
| 2. Estructuras de Control Anidadas en C++ .....                    | 16 |
| 2.1. Beneficios y Consideraciones de las Estructuras Anidadas..... | 19 |
| 3. Arreglos Estáticos .....  | 20 |
| 3.1. Arreglos Unidimensionales.....                                | 20 |
| 3.2. Matrices (Arreglos Multidimensionales).....                   | 21 |
| 3.3. Declaración de arreglos.....                                  | 22 |
| 3.4. Acceder a un elemento .....                                   | 23 |
| 3.5. Iteración sobre un arreglo .....                              | 24 |
| 3.6. Insertar elementos en un arreglo .....                        | 25 |
| 3.7. Buscar elementos en un arreglo .....                          | 26 |
| 3.8. La importancia de la dimensión lógica.....                    | 27 |
| 3.9. Dimensión Física.....   | 28 |
| 3.10. Arreglos estáticos: pasaje por parámetro .....               | 29 |
| 3.11. Eliminar un elemento de un arreglo .....                     | 30 |
| 3.12. Eliminar varias ocurrencias de un elemento .....             | 32 |
| 4. Resumen de la unidad .....                                      | 32 |

## Programación Imperativa

### Unidad 1

#### 1. Estructuras de Control

Las estructuras de control son fundamentales en la programación ya que permiten dirigir el flujo de ejecución de un programa.

En C++, las estructuras de control principales son las condicionales (como if, else if, y switch) y los bucles (como for, while, y do-while).

##### 1.1. If

Supongamos que queremos escribir un programa en C++ que solicite al usuario un número entero y determine si el número es positivo, negativo o cero.

Una solución podría ser la siguiente:

```
#include <iostream>
int main() {
    int numero;
    std::cout << "Ingrese un número entero: ";
    std::cin >> numero;
    if (numero > 0) {
        std::cout << "El número es positivo" << std::endl;
    } else if (numero < 0) {
        std::cout << "El número es negativo" << std::endl;
    } else {
        std::cout << "El número es cero" << std::endl;
    }
    return 0;
}
```

[Link al simulador](#)

Ahora, supongamos que el usuario ingresa el número "-5". El programa debería imprimir "El número es negativo".

Entonces, en primer lugar, se declara una variable llamada "*numero*", de tipo "int", para almacenar el número entero ingresado por el usuario.

Luego, se solicita la entrada de Usuario:

```
std::cout << "Ingrese un número entero: ";
```

```
std::cin >> numero;
```

Se solicita al usuario que ingrese un número entero y se almacena en la variable “numero”.

A continuación, se debe evaluar si el número es mayor, menor o igual a cero. Para ello se utiliza la estructura de Control “if-else”

```
if (numero > 0) {
```

```
    std::cout << "El número es positivo" << std::endl;
```

```
} else if (numero < 0) {
```

```
    std::cout << "El número es negativo" << std::endl;
```

```
} else {
```

```
    std::cout << "El número es cero" << std::endl;
```

```
}
```

- Si el número es mayor que 0 ( $\text{numero} > 0$ ), se imprime "El número es positivo".

- Si el número es menor que 0 ( $\text{numero} < 0$ ), se imprime "El número es negativo".

- Si el número no es mayor ni menor que 0, entonces es 0 y se imprime "El número es cero".

## 1.2. Switch

El uso de la estructura de control “switch” en C++ es útil para ejecutar diferentes bloques de código basados en el valor de una variable.

### Ejemplo 1: Menú de Opciones

```
#include <iostream>
using namespace std;
int main() {
    int opcion;
    cout << "Elige una opción (1-3): ";
    cin >> opcion;
    switch (opcion) {
        case 1:
            cout << "Has elegido la opción 1." << endl;
            break;
        case 2:
            cout << "Has elegido la opción 2." << endl;
            break;
        case 3:
            cout << "Has elegido la opción 3." << endl;
            break;
        default:
            cout << "Opción no válida." << endl;
            break;
    }
    return 0;
}
```

[Link al simulador](#)

En el ejercicio anterior sucede que:

- ✓ El usuario ingresa una opción (1, 2 o 3).
- ✓ El “switch” evalúa la variable “opcion” y ejecuta el bloque correspondiente.
- ✓ Si la opción no es válida, se ejecuta el bloque “default”.

## Ejemplo 2: Días de la Semana

```
#include <iostream>
using namespace std;
int main() {
    int dia;
    cout << "Ingresa un número de día de la semana (1-7): ";
    cin >> dia;
    switch (dia) {
        case 1:
            cout << "Lunes" << endl;
            break;
        case 2:
            cout << "Martes" << endl;
            break;
        case 3:
            cout << "Miércoles" << endl;
            break;
        case 4:
            cout << "Jueves" << endl;
            break;
        case 5:
            cout << "Viernes" << endl;
            break;
        case 6:
            cout << "Sábado" << endl;
            break;
        case 7:
            cout << "Domingo" << endl;
            break;
        default:
            cout << "Número de día no válido." << endl;
            break;
    }
    return 0;
}
```

[Link al simulador](#)

En el código anterior sucede que:

- ✓ El usuario ingresa un número del 1 al 7.
- ✓ El “switch” evalúa el número y muestra el día correspondiente.

- ✓ Si el número no está entre 1 y 7, se ejecuta el bloque “default”.

### Ejemplo 3: Fall Through (sin “break”)

```
#include <iostream>
using namespace std;
int main() {
    char letra;
    cout << "Ingresa una letra (A-C): ";
    cin >> letra;
    switch (letra) {
        case 'A':
            cout << "Letra A" << endl;
        case 'B':
            cout << "Letra B" << endl;
        case 'C':
            cout << "Letra C" << endl;
        default:
            cout << "Letra no válida." << endl;
    }
    return 0;
}
```

[Link al simulador](#)

**Explicación**, asumiendo que se trabajará con mayúsculas:

- ✓ Si el usuario ingresa 'A', se imprimirán "Letra A", "Letra B" y "Letra C" debido al "fall through".
- ✓ Si ingresa 'B', se imprimirán "Letra B" y "Letra C".
- ✓ Si ingresa 'C', solo se imprimirá "Letra C".
- ✓ Si la letra no es 'A', 'B' o 'C', se ejecuta el bloque “default”.

El uso correcto del “break” es fundamental para evitar el "fall through" (caer a través) no deseado, salvo que se quiera intencionadamente que el flujo de ejecución caiga al siguiente caso.

### 1.3. For

Se trata de una estructura de control que permite repetir un bloque de código un número determinado de veces. Es útil cuando conoces de antemano cuántas veces quieres que se ejecute el bloque de código.

### Ejemplo 1: Imprimir Números del 1 al 10

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 10; i++) {
        cout << i << " ";
    }
    return 0;
}
```

[Link al simulador](#)

### ¿Cómo funciona el programa anterior?

Se inicializa la variable del for: **int i = 1** (se establece la variable de control “i” en 1). Luego, se establece la condición correspondiente: **i <= 10** (el bucle se ejecuta mientras “i” sea menor o igual a 10). Además, se actualiza (en cada iteración), la variable “i”.

Entonces, el bucle imprime los números del 1 al 10.

### Ejemplo 2: Sumar Números del 1 al 100

```
#include <iostream>
using namespace std;
int main() {
    int suma = 0;
    for (int i = 1; i <= 100; i++) {
        suma += i;
    }
    cout << "La suma de los números del 1 al 100 es: " << suma << endl;
    return 0;
}
```

[Link al simulador](#)

### Explicación del programa anterior:

Se inicializa la variable del for: **int i = 1** (se establece la variable de control “i” en 1). Luego, se establece la condición correspondiente: **i <= 100** (el bucle se ejecuta mientras “i” sea menor o igual a 100). Además, se actualiza (en cada iteración), la variable “i”. También, dentro de cada iteración, se va incrementando el valor de la variable “suma” a partir del valor de **i**. Por último, una vez finalizado el bucle (cuando i toma el valor 100) se imprime el valor de la variable “suma”.



### Ejemplo 3: Iterar a través de un Array

```
#include <iostream>
using namespace std;
int main() {
    int numeros[] = {2, 4, 6, 8, 10};
    for (int i = 0; i < 5; i++) {
        cout << "Elemento en el índice " << i << ": " << numeros[i] << endl;
    }
    return 0;
}
```

[Link al simulador](#)

#### Explicación:

- ✓ Inicialización: "int i = 0" (se establece la variable de control "i" en 0).
- ✓ Condición: "i < tamaño" (el bucle se ejecuta mientras "i" sea menor que el tamaño del array).
- ✓ Actualización: "i++" (incrementa "i" en 1 después de cada iteración).
- ✓ El bucle itera a través de todos los elementos del array "numeros" y los imprime junto con su índice.

### Ejemplo 4: Bucle Infinito

```
#include <iostream>
using namespace std;
int main() {
    for (;;) {
        cout << "Este es un bucle infinito." << endl;
    }
    return 0;
}
```

[Link al simulador](#)

Este bucle "for" no tiene inicialización, condición ni actualización explícitas, lo que provoca que se ejecute indefinidamente. Se utiliza generalmente para situaciones en las que el bucle debe continuar hasta que se cumpla una condición interna y se rompa el bucle con una instrucción "break". ¿Cómo sería esa opción?

### Ejemplo 5: Bucle “for” con múltiples variables

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 0, j = 10; i <= 10; i++, j--) {
        cout << "i: " << i << ", j: " << j << endl;
    }
    return 0;
}
```

[Link al simulador](#)

En este caso se inicializan dos variables: “int i = 0, j = 10”. Luego, se establece una condición “i <= 10” (el bucle se ejecuta mientras “i” sea menor o igual a 10), y se actualizan del siguiente modo: “i++” y “j--” (incrementa “i” en 1 y decrementa “j” en 1 después de cada iteración).

Finalmente, el bucle imprime los valores de “i” y “j” en cada iteración.

## 1.4. While

El bucle “while” en C++ es una estructura de control que permite repetir un bloque de código mientras una condición específica sea verdadera. A diferencia del bucle “for”, el bucle “while” es más adecuado cuando no se conoce de antemano el número de iteraciones que se necesita.

### Ejemplo 1: Imprimir Números del 1 al 10

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    while (i <= 10) {
        cout << i << " ";
        i++;
    }
    return 0;
}
```

[Link al simulador](#)

En el código se inicializa “i” a 1, se establece como condición que se itere mientras “i <= 10” (mientras sea verdadera, el bloque de código se ejecuta). Dentro del bloque, se imprime “i” y luego se incrementa “i”. Por último, el bucle termina cuando “i” es mayor que 10.

Tanto en este ejemplo como en el que sigue a continuación es necesario aclarar que se conoce de antemano el número de repeticiones, por lo que, sería más apropiado utilizar un bucle for. Puede suceder que en lugar de que el valor de la variable “i” (que es la que permite la iteración del while) este dada por el sistema (línea: int i = 1), se solicite al usuario. De ese modo el programa dependería del valor capturado y debería evaluarlo en cada iteración de la estructura de control.

### Ejemplo 2: Sumar Números del 1 al 100

```
#include <iostream>
using namespace std;
int main() {
    int suma = 0;
    int i = 1;
    while (i <= 100) {
        suma += i;
        i++;
    }
    cout << "La suma de los números del 1 al 100 es: " << suma << endl;
    return 0;
}
```

[Link al simulador](#)

En el programa anterior se inicializan “i” a 1 y “suma” a 0. Se establece como condición que se itere mientras “i <= 100”.- Dentro del bloque, se añade “i” a “suma” y luego se incrementa “i”. Por último, el bucle termina cuando “i” es mayor que 100.

Finalmente, el programa imprime la suma total.

### Ejemplo 3: Pedir al Usuario un Número Positivo

```
#include <iostream>
using namespace std;
int main() {
    int numero;
    cout << "Ingresa un número positivo: ";
    cin >> numero;
    while (numero <= 0) {
        cout << "El número no es positivo. Intenta de nuevo: ";
        cin >> numero;
    }
    cout << "Has ingresado un número positivo: " << numero << endl;
    return 0;
}
```

[Link al simulador](#)

#### Explicación:

- ✓ Se pide al usuario que ingrese un número positivo.
- ✓ La condición “numero <= 0” se evalúa. Mientras sea verdadera, el bloque de código se ejecuta.
- ✓ Si el número ingresado no es positivo, se pide al usuario que intente de nuevo.
- ✓ El bucle termina cuando el usuario ingresa un número positivo.

### Ejemplo 4: Bucle Infinito

```
#include <iostream>
using namespace std;
int main() {
    while (true) {
        cout << "Este es un bucle infinito." << endl;
    }
    return 0;
}
```

[Link al simulador](#)

En el programa anterior la condición “true” siempre es verdadera, por lo que el bucle se ejecuta indefinidamente. Se utiliza generalmente en situaciones donde el bucle debe continuar hasta que se cumpla una condición interna y se rompa el bucle con una instrucción “break”. ¿Cómo sería esa opción?

### Ejemplo 5: Leer Enteros Hasta que el Usuario Ingrese un Cero

```
#include <iostream>
using namespace std;
int main() {
    int numero;
    cout << "Ingresa un número (0 para salir): ";
    cin >> numero;
    while (numero != 0) {
        cout << "Has ingresado: " << numero << endl;
        cout << "Ingresa otro número (0 para salir): ";
        cin >> numero;
    }
    cout << "Has salido del bucle." << endl;
    return 0;
}
```

[Link al simulador](#)

#### Explicación:

- ✓ Se pide al usuario que ingrese un número.
- ✓ La condición “numero != 0” se evalúa. Mientras sea verdadera, el bloque de código se ejecuta.
- ✓ Dentro del bloque, se imprime el número ingresado y se pide otro número.
- ✓ El bucle termina cuando el usuario ingresa 0.
- ✓ El programa imprime un mensaje indicando que se ha salido del bucle.

#### 1.5. Do- while

El bucle “do-while” en C++ es una estructura de control que permite ejecutar un bloque de código al menos una vez y luego repetir la ejecución del bloque mientras una condición específica sea verdadera. La diferencia principal entre “while” y “do-while” es que el bloque de código en “do-while” se ejecuta al menos una vez, ya que la condición se evalúa después de la ejecución del bloque.

### Ejemplo 1: Imprimir Números del 1 al 10

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    do {
        cout << i << " ";
        i++;
    } while (i <= 10);
    return 0;
}
```

[Link al simulador](#)

#### Explicación:

- ✓ Se inicializa "i" a 1.
- ✓ El bloque de código se ejecuta, imprimiendo "i" y luego incrementando "i".
- ✓ La condición, del while, "i <= 10" se evalúa después de la ejecución del bloque.
- ✓ El bucle se repite mientras "i" sea menor o igual a 10.

### Ejemplo 2: Pedir al Usuario un Número Positivo

```
#include <iostream>
using namespace std;
int main() {
    int numero;
    do {
        cout << "Ingresa un número positivo: ";
        cin >> numero;
        if (numero <= 0) {
            cout << "El número no es positivo. Intenta de nuevo." << endl;
        }
    } while (numero <= 0);
    cout << "Has ingresado un número positivo: " << numero << endl;
    return 0;
}
```

[Link al simulador](#)

### Explicación:

- ✓ Se pide al usuario que ingrese un número positivo.
- ✓ Si el número ingresado no es positivo, se muestra un mensaje y el bucle se repite.
- ✓ La condición "numero <= 0" se evalúa después de la ejecución del bloque.
- ✓ El bucle termina cuando el usuario ingresa un número positivo.

### Ejemplo 3: Menú Simple

```
#include <iostream>
using namespace std;
int main() {
    int opcion;
    do {
        cout << "Menú de opciones:" << endl;
        cout << "1. Opción 1" << endl;
        cout << "2. Opción 2" << endl;
        cout << "3. Salir" << endl;
        cout << "Selecciona una opción: ";
        cin >> opcion;
        switch (opcion) {
            case 1:
                cout << "Has seleccionado la Opción 1" << endl;
                break;
            case 2:
                cout << "Has seleccionado la Opción 2" << endl;
                break;
            case 3:
                cout << "Saliendo..." << endl;
                break;
            default:
                cout << "Opción no válida. Intenta de nuevo." << endl;
        }
    } while (opcion != 3);
    return 0;
}
```

[Link al simulador](#)

En el programa anterior se muestra un menú de opciones al usuario. Entonces, el usuario selecciona una opción y se procesa la entrada utilizando un "switch". La condición "opcion != 3" se evalúa después de la ejecución del bloque.

El bucle se repite mientras el usuario no seleccione la opción para salir (opción 3).

#### Ejemplo 4: Leer Enteros Hasta que el Usuario Ingrese un Cero

```
#include <iostream>
using namespace std;
int main() {
    int numero;
    do {
        cout << "Ingresa un número (0 para salir): ";
        cin >> numero;
        if (numero != 0) {
            cout << "Has ingresado: " << numero << endl;
        }
    } while (numero != 0);
    cout << "Has salido del bucle." << endl;
    return 0;
}
```

[Link al simulador](#)

#### Explicación:

- ✓ Se pide al usuario que ingrese un número.
- ✓ Si el número ingresado no es cero, se imprime el número.
- ✓ La condición, del while, "numero != 0" se evalúa después de la ejecución del bloque.
- ✓ El bucle se repite mientras el usuario no ingrese cero.
- ✓ El programa imprime un mensaje indicando que se ha salido del bucle.

#### Ejemplo 5: Sumar Números Hasta que el Usuario Ingrese un Cero

```
#include <iostream>
using namespace std;
int main() {
    int numero, suma = 0;
    do {
        cout << "Ingresa un número (0 para salir): ";
        cin >> numero;
        suma += numero;
    } while (numero != 0);
    cout << "La suma de los números ingresados es: " << suma << endl;
    return 0;
}
```



[Link al simulador](#)

En el programa se solicita ingresar un número y se le suma a la variable “suma”. La condición “numero != 0” se evalúa después de la ejecución del bloque.

El bucle se repite mientras el usuario no ingrese cero, cuando finalmente sale del bucle, el programa imprime la suma total de los números ingresados.

## 2. Estructuras de Control Anidadas en C++

Cuando decimos que las estructuras de control están "anidadas", nos referimos a que una estructura de control está contenida dentro de otra. Esto significa que puedes tener un bucle dentro de otro bucle, una instrucción condicional dentro de un bucle, un bucle dentro de una instrucción condicional, etc.

La anidación permite crear comportamientos más complejos y específicos en los programas.

### Ejemplos de Estructuras de Control Anidadas en C++

#### Ejemplo 1: Bucles Anidados (“for” dentro de “for”)

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 3; i++) { // Bucle externo
        for (int j = 1; j <= 2; j++) { // Bucle interno
            cout << "i = " << i << ", j = " << j << endl;
        }
    }
    return 0;
}
```

[Link al simulador](#)

Se va a imprimir lo siguiente:

```
i = 1, j = 1
i = 1, j = 2
i = 2, j = 1
i = 2, j = 2
i = 3, j = 1
i = 3, j = 2
```

Dado que el bucle externo controla la variable “i” y se ejecuta 3 veces.

- Dentro del bucle externo, hay un bucle interno que controla la variable “j” y se ejecuta 2 veces por cada iteración del bucle externo.
- El resultado es que se ejecutan un total de 6 iteraciones ( $3 * 2$ ), con combinaciones de valores de “i” y “j”.

### Ejemplo 2: Instrucciones “if” Anidadas

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

int main() {
    int a = 5, b = 10;

    if (a < b) { // Primera condición
        if (a == 5) { // Segunda condición anidada dentro de la primera
            cout << "a es menor que b y a es igual a 5" << endl;
        }
    }

    return 0;
}
```

[Link al simulador](#)

### Explicación:

- ✓ La primera condición “if (a < b)” se evalúa.
- ✓ Si es verdadera, se evalúa la segunda condición “if (a == 5)”.
- ✓ Si ambas condiciones son verdaderas, se ejecuta el bloque de código anidado.

### Ejemplo 3: Bucles y Condiciones Anidadas

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 5; i++) { // Bucle externo
        if (i % 2 == 0) { // Condición anidada dentro del bucle
            cout << i << " es un número par." << endl;
        } else {
            cout << i << " es un número impar." << endl;
        }
    }
    return 0;
}
```

```
}
```

[Link al simulador](#)

#### Explicación:

- ✓ El bucle externo se ejecuta 5 veces, iterando “i” de 1 a 5.
- ✓ Dentro del bucle, hay una instrucción “if-else” que verifica si “i” es par o impar y luego imprime un mensaje apropiado.

#### Ejemplo 4: “while” Anidado dentro de “for”

```
#include <iostream>
using namespace std;
int main() {
    for (int i = 1; i <= 3; i++) { // Bucle externo
        int j = 1;
        while (j <= 2) { // Bucle interno
            cout << "i = " << i << ", j = " << j << endl;
            j++;
        }
    }
    return 0;
}
```

[Link al simulador](#)

#### Explicación:

- ✓ El bucle externo “for” se ejecuta 3 veces.
- ✓ Dentro de cada iteración del bucle externo, hay un bucle “while” que se ejecuta 2 veces.
- ✓ Se imprimen las combinaciones de valores de “i” y “j”.

#### Ejemplo 5: “do-while” Anidado dentro de “while”

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
    while (i <= 3) { // Bucle externo
        int j = 1;
        do { // Bucle interno
            cout << "i = " << i << ", j = " << j << endl;
            j++;
        } while (j <= 2);
        i++;
    }
}
```

```
        i++;  
    }  
    return 0;  
}
```

[Link el simulador](#)

En el programa anterior, el bucle externo “while” se ejecuta mientras “i” sea menor o igual a 3. Dentro de ese bucle, hay un bucle “do-while” que se ejecuta mientras “j” sea menor o igual a 2.

Finalmente, se imprimen las combinaciones de valores de “i” y “j”.

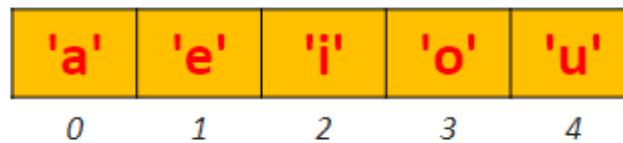
### 2.1. Beneficios y Consideraciones de las Estructuras Anidadas

Las estructuras de control anidadas son una herramienta poderosa en C++ para manejar situaciones complejas que requieren múltiples niveles de condiciones y repeticiones, permitiendo construir programas más complejos y específicos, y facilitando la implementación de lógica que depende de varias condiciones y repeticiones.

Sin embargo, es importante considerar que la legibilidad del código puede disminuir con demasiados niveles de anidamiento. Mantener el código limpio y bien comentado es esencial para facilitar su mantenimiento y comprensión. Además, el exceso de anidamiento puede llevar a errores lógicos difíciles de detectar.

### 3. Arreglos Estáticos

Un arreglo estático en C++ es una colección de elementos del mismo tipo que se almacena de manera contigua en la memoria. La característica principal de los arreglos estáticos es que su tamaño debe conocerse en tiempo de compilación y no puede cambiar durante la ejecución del programa.



*Ilustración 1: Representación simbólica de un arreglo de char.*

#### Declaración y Ejemplo:

```
#include <iostream>
int main() {
    int arr[5]; // Declaración de un arreglo estático de 5 enteros
    // Inicialización del arreglo
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    arr[3] = 40;
    arr[4] = 50;
    // Acceso e impresión de los elementos del arreglo
    for (int i = 0; i < 5; ++i) {
        std::cout << "arr[" << i << "] = " << arr[i] << std::endl;
    }
    return 0;
}
```

[Link al simulador](#)

En este ejemplo, se declara un arreglo de 5 enteros y se inicializan sus elementos. Luego, se imprime cada elemento utilizando un bucle “for”.

#### 3.1. Arreglos Unidimensionales

Un arreglo unidimensional es el tipo más básico de arreglo, donde los elementos se almacenan en una sola fila. El arreglo anterior es un ejemplo de un arreglo unidimensional.

### 3.2. Matrices (Arreglos Multidimensionales)

Una matriz es un arreglo de arreglos, lo que permite almacenar datos en más de una dimensión. *Las matrices más comunes son las de dos dimensiones, que se pueden imaginar como tablas con filas y columnas.*

#### Declaración y Ejemplo:

```
#include <iostream>
int main() {
    int matrix[3][3] = { // Declaración e inicialización de una matriz 3x3
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    // Acceso e impresión de los elementos de la matriz
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            std::cout << "matrix[" << i << "][" << j << "] = " << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

[Link al simulador](#)

En este ejemplo, se declara e inicializa una matriz de 3x3 enteros. Los elementos se acceden mediante dos bucles anidados.

### 3.3. Declaración de arreglos

Cuando se declara un arreglo, se debe indicar el tamaño de dicho arreglo, y éste permanecerá constante durante el ciclo de vida del programa o de la función, es decir, no podrá cambiar su tamaño de manera dinámica (por esto es que son arreglos de tamaño estático): si el espacio es insuficiente o sobra durante la ejecución del programa, no hay nada que hacer (más que modificar el código y volver a correr el programa), ya que el tamaño es fijo.

```
int numeros[5] = {1, 2, 3, 4, 5};
int numeros2[] = {1, 2, 3, 4, 5, 6};
int valores[3] = {10, 20};
```

|          |       |     |     |     |     |     |     |
|----------|-------|-----|-----|-----|-----|-----|-----|
| numeros  | array | 0   | 1   | 2   | 3   | 4   |     |
|          | int   | int | int | int | int |     |     |
|          | 1     | 2   | 3   | 4   | 5   |     |     |
| numeros2 | array | 0   | 1   | 2   | 3   | 4   | 5   |
|          | int   | int | int | int | int | int | int |
|          | 1     | 2   | 3   | 4   | 5   | 6   |     |
| valores  | array | 0   | 1   | 2   |     |     |     |
|          | int   | int | int |     |     |     |     |
|          | 10    | 20  | 0   |     |     |     |     |

Ilustración 2: Ejemplos de declaración de arreglos

Un arreglo se declara indicando: el tipo de sus elementos, el identificador de la variable y, entre corchetes, la cantidad máxima de elementos que podrá contener cada dimensión.

`char vocales[5] ;`

Al declarar un arreglo, podemos inicializarlo con valores:

`char vocales[5] ={'a', 'e'};`

Al declarar un arreglo no es obligatorio especificar el tamaño. *Notar que no se definieron todos los valores, sólo algunos.*

En este caso, la dimensión se calcula a partir del número de elementos en la lista de valores iniciales.

`float S[] = {2, 32, 4.6, 2, 1, 0.5, 3, 8, 0, 12};`

Si el arreglo se declara en `main{ }`, durante toda la vida o ejecución del programa estará ocupando la porción de memoria necesaria y ese recurso no podrá ser empleado en otra cosa. Si se lo declara dentro de una función o un bloque, ocupará memoria mientras el control de ejecución opere dentro de dicha función o bloque.

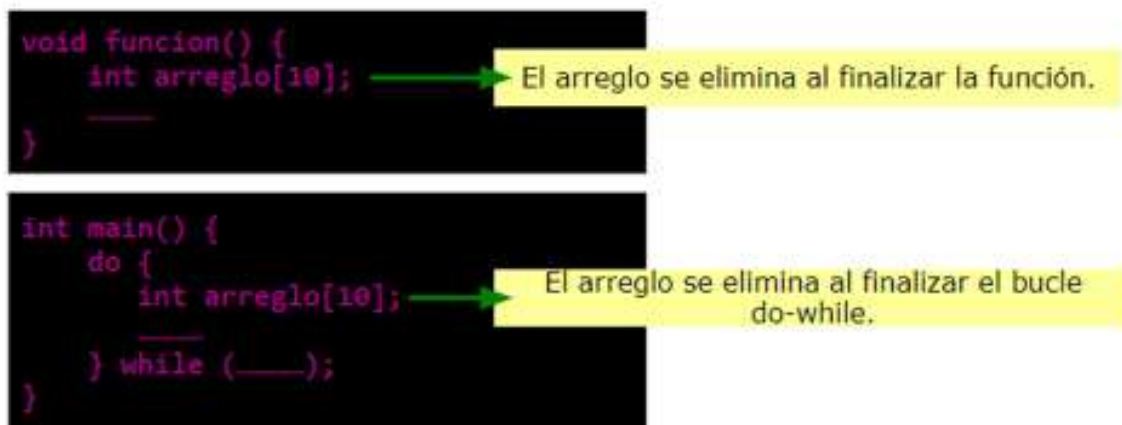


Ilustración 3: ejemplo de "ámbitos" de arreglos

### 3.4. Acceder a un elemento

Para acceder a un elemento en particular se deben indicar el nombre del arreglo (identificador de la variable) y la posición del índice donde se encuentra el elemento (ver la Ilustración 4).

```
char V[5] = { 'a', 'e', 'i', 'o', 'u' };
cout << V[3]<< endl;

int i=5;

cout << V[i-1]<< endl;
int cuenta= 5 - 4;
cout << V[cuenta]<< endl;
```

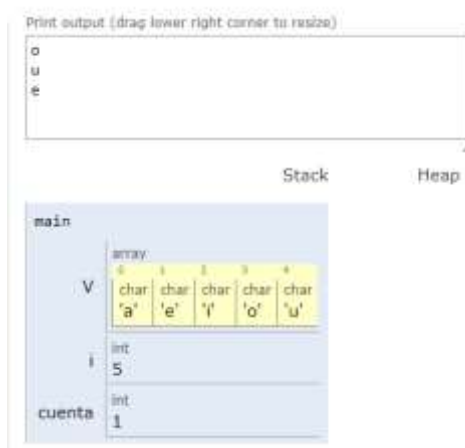


Ilustración 4

[Link al simulador](#)



### 3.5. Iteración sobre un arreglo

Para acceder a todos los elementos de un arreglo (por ejemplo: para imprimirlos o para buscar la posición de un elemento en particular), iteraremos incrementando una variable numérica, que nos servirá de índice.

Pero ¿hasta dónde vamos a iterar?

```
char V[5] = { 'a', 'e', 'i'};
cout << V[0]<< endl;
cout << V[1]<< endl;
cout << V[2]<< endl;

cout << V[3]<< endl;
cout << V[4]<< endl;
```

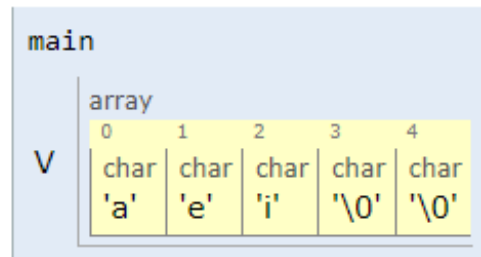


Ilustración 5: ejemplo de acceso a cada uno de los elementos de un arreglo

Recordemos en este punto los conceptos de **Dimensión Lógica** y **Dimensión Física**.

Entonces, para recorrer el arreglo me voy a guiar por la dimensión lógica.

```
char V[5] = { 'a', 'e', 'i'};
int dimension_logica=3;
for (int i = 0; i < dimension_logica; i++)
    cout << V[i] << endl;
```

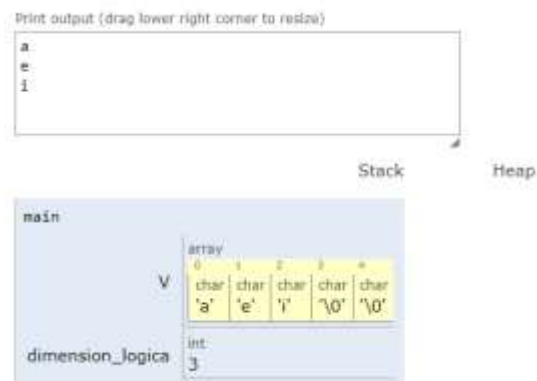


Ilustración 6: ejemplo de recorrido de un arreglo a través de su dimensión lógica.

### 3.6. Insertar elementos en un arreglo

Al insertar un nuevo elemento en un arreglo, la dimensión lógica nos indicará si es posible o no agregar un nuevo elemento.

```
char V[5] = { 'a', 'e', 'i' };
int dimension_logica=3;
const int dimension_fisica=5;
for (int i = 0; i < dimension_logica; i++)
    cout << V[i] << endl;
if (dimension_logica != dimension_fisica){
    V[dimension_logica]='o';
    dimension_logica++;
}
```

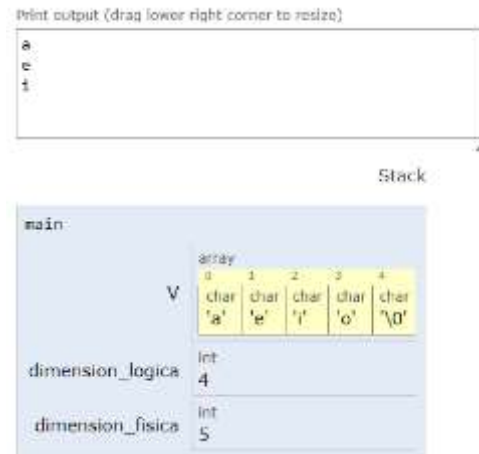


Ilustración 7

En el ejemplo de la *Ilustración 7*, antes de insertar el nuevo elemento se corrobora que haya espacio en el arreglo. Esto se hace preguntando lo siguiente:  
*if ( dimension\_logica != dimension\_fisica)*

### 3.7. Buscar elementos en un arreglo

Para buscar un elemento en un arreglo se debe considerar que se va a recorrer siempre que no se encuentre el elemento buscado y mientras que no se termine la estructura.

En ese sentido, habrá que preguntar por ambas condiciones. Por ejemplo, como se muestra en la siguiente ilustración.

```
char V[5] = { 'a', 'e', 'i' };
int dimension_logica=3;
const int dimension_fisica=5;
int indiceActual=0;
char elementoBuscado='i';
while (indiceActual < dimension_logica && V[indiceActual] != elementoBuscado)
    indiceActual++;
if (indiceActual < dimension_logica){
    cout<<"Se encontró el elemento "<< endl;
}
else
    cout<<"No se encontró el elemento "<< endl;
```

Print output (drag lower right corner to resize)

Se encontró el elemento

Stack

|                  |   |      |      |      |   |   |      |      |      |      |      |     |     |     |      |
|------------------|---|------|------|------|---|---|------|------|------|------|------|-----|-----|-----|------|
| main             |   |      |      |      |   |   |      |      |      |      |      |     |     |     |      |
| V                | array   |      |      |      |   |   |      |      |      |      |      |     |     |     |      |
|                  | <table><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>char</td><td>char</td><td>char</td><td>char</td><td>char</td></tr><tr><td>'a'</td><td>'e'</td><td>'i'</td><td>'\0'</td><td>'\0'</td></tr></table> | 0    | 1    | 2    | 3 | 4 | char | char | char | char | char | 'a' | 'e' | 'i' | '\0' |
| 0                | 1   | 2    | 3    | 4    |   |   |      |      |      |      |      |     |     |     |      |
| char             | char  | char | char | char |   |   |      |      |      |      |      |     |     |     |      |
| 'a'              | 'e'   | 'i'  | '\0' | '\0' |   |   |      |      |      |      |      |     |     |     |      |
| dimension_logica | int<br>3  |      |      |      |   |   |      |      |      |      |      |     |     |     |      |
| dimension_fisica | int<br>5  |      |      |      |   |   |      |      |      |      |      |     |     |     |      |
| indiceActual     | int<br>2  |      |      |      |   |   |      |      |      |      |      |     |     |     |      |
| elementoBuscado  | char<br>'i'   |      |      |      |   |   |      |      |      |      |      |     |     |     |      |

[Link al simulador](#)

### 3.8. La importancia de la dimensión lógica

La dimensión lógica no es más que la cantidad de elementos útiles contenidos en un arreglo (siempre comenzando desde el 0 y considerando que los elementos se almacenan en posiciones consecutivas). Usualmente la representamos mediante una variable de tipo entero, que acompaña al arreglo.

Si los elementos del arreglo contienen datos útiles o no, es una cuestión que sólo interesa al programador: para la máquina todo el espacio de memoria destinado al arreglo (dimensión física) está ocupado. Para determinar qué parte contiene datos válidos usamos el concepto de dimensión lógica.

La dimensión lógica nunca podrá ser mayor que la física, porque la física es el máximo de elementos que podemos guardar en un arreglo determinado.

Al declarar un arreglo debemos dar una dimensión física (a menos que lo hayamos inicializado, como ya se vio anteriormente), pero es muy importante que también declaremos una variable más, para indicar la dimensión lógica. Esta dimensión lógica se inicializará siempre en 0.

```
int numeros[100];  
int dimension_logica = 0;
```

Cada vez que agreguemos un elemento, la dimensión lógica se incrementará. De igual forma, al eliminar elementos, la dimensión lógica se decrementará.

La máquina desconoce cuántos elementos del arreglo "le interesan" al usuario: sólo sabe cuánto espacio ocupa el arreglo. Asimismo, el usuario desconoce cuánto espacio ocupa el arreglo: sólo sabe cuántos elementos ha indicado al programa que debe guardar.

Entonces, el usuario tiene una visión de "alto nivel" donde sólo le interesa que sus datos se almacenen. En el código usamos la dimensión lógica para indicar esto, cuando sabemos que el usuario podría no utilizar todo el espacio que tiene disponible el arreglo.

Sólo en casos puntuales un arreglo no necesitará una dimensión lógica, y estos son cuando el arreglo siempre estará lleno en su totalidad (es decir: dimensión lógica y dimensión física coinciden). Por ejemplo: un arreglo que guarde los nombres de los 7 días de la semana.

A menos que sepamos con seguridad que ese es el caso, siempre deberemos declarar una dimensión lógica, porque nos permitirá llenar el arreglo hasta la cantidad de

elementos deseados y no necesariamente hasta el final. Es decir: nos permite abstraer al usuario de qué tipo de estructura estamos usando para almacenar sus datos.

### 3.9. Dimensión Física

Sabemos que la dimensión física de un arreglo no cambia en toda la ejecución del programa, y que nos indica el "tope" o la cantidad máxima de elementos que podemos almacenar, por lo que es posible que necesitemos usarla en algunas funciones.

Por ello es útil declarar a la dimensión física como constante, al inicio del programa:

```
#include <iostream>
using namespace std;
const int DIMENSION_FISICA = 100;
//otras funciones del programa
int main() {
    int arreglo[DIMENSION_FISICA];
    int dimension_logica = 0;
    return 0;
}
```

Si se hace necesario alterar el código para cambiar la dimensión física, basta con cambiar sólo una línea: la declaración de constante.

| Importante para recordar |  |   |
|--------------------------|--|---|
| Característica           | Dimensión Lógica                                       | Dimensión Física  |
| Definición               | Cantidad de elementos <b>útiles</b> actualmente en uso | Cantidad <b>máxima</b> de elementos que puede almacenar     |
| Tipo de dato             | Variable entera (ej. int dimension_logica)             | Constante entera (ej. const int DIMENSION_FISICA)           |
| Quién la usa             | El <b>programador</b> o usuario del arreglo            | El <b>compilador</b> y el programa                          |
| Valor inicial            | Generalmente <b>0</b>                                  | Se define al <b>declarar el arreglo</b>                     |
| ¿Puede cambiarse?        | ✓ Sí, al insertar o eliminar elementos                 | ✗ No, es fija durante la ejecución del programa             |
| Controla iteraciones     | ✓ Sí, para recorrer sólo los datos útiles              | A veces, pero solo si se requiere iterar toda la estructura |

### 3.10. Arreglos estáticos: pasaje por parámetro

Debido a que el pasaje de parámetros por valor implica una copia completa de la variable en la memoria, C++ no permite el pasaje de arreglos por valor y fuerza el pasaje por referencia. Esto implica que no es necesario incluir el operador &, porque los arreglos se pasan automáticamente por referencia al usar corchetes ([]):

```
void funcion (int arreglo[], int dl) {  
    //cuerpo de la función  
}  
int main() {  
    int arreglo[50];  
    int dimension_logica = 0;  
    funcion(arreglo, dimension_logica);  
    return 0;  
}
```

En el ejemplo anterior, aunque no esté el operador &, el pasaje del arreglo se está haciendo por referencia. Por otra parte, la función main() realiza la llamada a la función, con el arreglo y la dimensión lógica como argumentos.

Entonces, lo que en verdad sucede es que, al pasar un arreglo como parámetro, el compilador lo reemplaza por una referencia al primer elemento. Los elementos siguientes se obtienen mediante un desplazamiento por la memoria, ya que se sabe de antemano que un arreglo se almacena en posiciones contiguas.

Debido a que los arreglos se pasan siempre por referencia, C++ no permite que una función retorne un arreglo estático.

Lo que muestra el siguiente código es un error:

```
arreglo[] funcion (int arreglo[], int dl) {  
    //cuerpo de la función  
}
```


### 3.11. Eliminar un elemento de un arreglo

Los elementos de un arreglo no se “eliminan”, sino que se sobrescriben con otros datos.

Por ejemplo, si en un arreglo de dimensión lógica 8 queremos eliminar el elemento en la posición 4, haremos un corrimiento de los datos ubicados del 5 en adelante.

```
for (int i = 4; i < dimension_logica-1; i++) {  
    arreglo[i] = arreglo[i+1];  
}  
dimension_logica--;
```

Iteramos hasta el elemento ubicado en la posición dada por la dimensión lógica menos dos: el elemento en la posición 7 es el último, por lo que no se le debería colocar lo que hay en la posición siguiente (que no es válida).

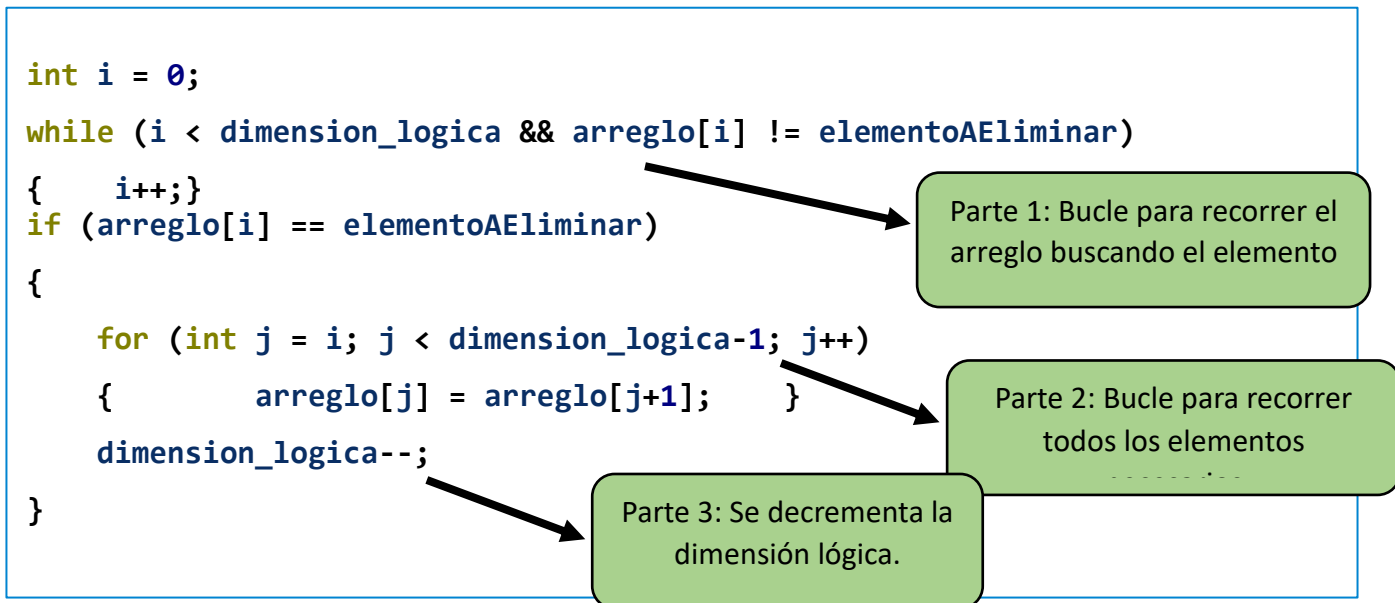


|   |   |   |    |    |    |    |    |   |   |
|---|---|---|----|----|----|----|----|---|---|
| 2 | 5 | 8 | 13 | 19 | 25 | 28 | 32 |   |   |
| 0 | 1 | 2 | 3  | 4  | 5  | 6  | 7  | 8 | 9 |

Recordemos que no debemos dejar "huecos" de datos no útiles en el arreglo. Todos los datos que nos interesan deben estar contiguos, empezando desde la posición 0.

Normalmente, el algoritmo de eliminación tiene tres partes:

- 1- Buscar el elemento a eliminar (que también podría no existir dentro del arreglo), y obtener su posición en el índice.
- 2- Si se encontró, correr los elementos desde el siguiente a eliminar hasta el final (lógico) del arreglo, para sobrescribir al eliminado.
- 3- Decrementar la dimensión lógica.



Pero ¡Atención! En el ejemplo anterior sólo se elimina la primera ocurrencia del elemento.



### 3.12. Eliminar varias ocurrencias de un elemento

También podemos usar un algoritmo similar si queremos eliminar todas las ocurrencias de un elemento (y no sólo la primera).

```

int i=0;
while (i < dl) {
    if (arreglo[i] == elementoAEliminar) {
        for (int j = i; j < dl-1; j++) {
            arreglo[j] = arreglo[j+1];
        }
        dl--;
    }
    else
        i++;
}

```

Utilizamos while porque sólo queremos incrementar el índice si el elemento no se encontró en la posición actual

Si se encuentra una ocurrencia, realizamos el corrimiento visto anteriormente y decrementamos la dimensión lógica

Si la posición actual no contiene una ocurrencia del elemento a eliminar, avanzamos el índice para seguir buscando

## 4. Resumen de la unidad

Esta primera unidad aborda como tema fundamental las estructuras de control en C++. Se explican y ejemplifican aquellas estructuras que son fundamentales, y permiten dirigir el flujo de ejecución de un programa.

En la sección de estructuras de control condicionales, se detallan ejemplos prácticos del uso de "if" y "switch". Se muestra cómo utilizar "if" para evaluar condiciones y ejecutar diferentes bloques de código en función del resultado de estas evaluaciones. También se presentan varios ejemplos de "switch", ilustrando su uso para seleccionar entre múltiples opciones en función del valor de una variable. Por otro lado, en la parte dedicada a los bucles se explica cómo repetir bloques de código utilizando "for", "while", y "do-while". Se presentan varios ejemplos prácticos para cada tipo de bucle, mostrando cómo utilizarlos para tareas comunes como iterar sobre arrays, realizar sumas acumulativas, y manejar entradas del usuario.

Luego, se abordan las estructuras de control anidadas, destacando los beneficios y consideraciones al utilizarlas en C++.

Finalmente, se introducen los arreglos estáticos, incluyendo arreglos unidimensionales y multidimensionales (matrices). Se explica cómo declarar, acceder, iterar e insertar elementos en arreglos; así como también cómo eliminar una o más ocurrencias de un elemento.