1. Estructura de Datos: Array (Arreglo)

Un **array** (o **arreglo**) es una estructura de datos fundamental que almacena una colección de elementos del mismo tipo en ubicaciones contiguas de memoria. Cada elemento es accesible mediante un **índice**, comenzando desde 0.

—

0.1. Características de los Arrays (estáticos)

- **Acceso Directo:** El acceso a cualquier elemento se realiza en tiempo constante $\mathcal{O}(1)$ usando el índice.
- **Tamaño Fijo:** El tamaño del array se define al momento de su creación y no puede cambiarse (en arrays estáticos).
- **Tipo Homogéneo:** Todos los elementos deben ser del mismo tipo de dato.
- **Eficiente en Memoria:** Al estar almacenado de forma contigua, permite un acceso muy eficiente y predecible a nivel de hardware.

—

0.2. Operaciones Básicas sobre Arrays

1. **Acceso:** Obtener el valor de un elemento mediante su índice. Tiene complejidad $\mathcal{O}(1)$.
2. **Modificación:** Cambiar el valor de un elemento existente. Tiene complejidad $\mathcal{O}(1)$.
3. **Recorrido:** Procesar todos los elementos uno por uno (por ejemplo, para buscar o imprimir). Tiene complejidad $\mathcal{O}(n)$, siendo n la longitud del array.
4. **Inserción:** Añadir un elemento. Se puede hacer siempre que el número de elementos utilizados en el array es menor que su capacidad, en ese caso, tiene complejidad $\mathcal{O}(1)$. Si quisieramos insertar un elemento en una posición intermedia, tendríamos que mover todos los demás para dejar hueco. Por tanto, tiene complejidad $\mathcal{O}(n)$.
5. **Eliminación:** Quitar un elemento. Se puede, pero hay que desplazar todos los demás para evitar dejar un hueco. Por tanto, tiene complejidad $\mathcal{O}(n)$

—

0.3. Declaración y Uso de Arrays en C++

Array Estático en C++

Listing 1: Declaración y uso de un array estático

```cpp
#include <iostream>

using namespace std;

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

    // Acceso a elementos
    cout << "Primer elemento: " << numeros[0] << endl;
    cout << "Ultimo elemento: " << numeros[4] << endl;

    // Modificacion
    numeros[2] = 100;

    // Recorrido
    cout << "Array completo: ";
    for (int i = 0; i < 5; i++) {
        cout << numeros[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Array Dinámico usando `vector` en C++

Listing 2: Uso del contenedor vector de la STL

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> numeros = {1, 2, 3};

    // Anadir elementos al final
    numeros.push_back(4);
    numeros.push_back(5);

    // Recorrer el vector
    for (int num : numeros) { // forma abreviada de
        recorrer todos los elementos (in-range)
```

```
15          cout << num << " ";
16      }
17      cout << endl;
18
19      // Acceder a elementos
20      cout << "Elemento en indice 2: " << numeros[2] <<
            endl;
21
22      return 0;
23  }
```

—

0.4. Ventajas y Desventajas de los Arrays

Ventajas:

- Acceso rápido mediante índices ($\mathcal{O}(1)$).
- Fácil de implementar.
- Bajo coste de memoria por no tener punteros adicionales (a diferencia de listas).

Desventajas:

- Tamaño fijo (en arrays estáticos).
- Inserciones y eliminaciones costosas ($\mathcal{O}(n)$) debido al desplazamiento de elementos.
- Ineficiencia en uso de memoria si se reserva más de la necesaria.

—

0.5. Aplicaciones Comunes de los Arrays

- Almacenamiento de colecciones simples de datos (temperaturas, notas, conteos, etc.).
- Estructuras base para algoritmos como ordenamiento y búsqueda.
- Implementación de estructuras más complejas: pilas, colas, heaps.
- Representación de matrices bidimensionales (arrays de arrays).

Los arrays son el bloque fundamental sobre el cual se construyen muchas otras estructuras de datos. Su comprensión es esencial para trabajar con algoritmos eficientes y estructuras complejas.

1. Uso de la Biblioteca `vector` de la STL

La clase `vector` de la STL (*Standard Template Library*) es una versión dinámica y segura de los arrays tradicionales. Es altamente utilizada por su flexibilidad y eficiencia.

Características de `vector`:

- Crece automáticamente cuando se agregan elementos.
- Permite acceso mediante índices como un array.
- Internamente maneja la gestión de memoria.
- Soporta funciones integradas para insertar, eliminar y recorrer elementos.

Funciones Útiles de `vector`:

- `push_back(x)` — Inserta el elemento x al final.
- `pop_back()` — Elimina el último elemento.
- `size()` — Devuelve el número de elementos actuales.
- `empty()` — Retorna `true` si el vector está vacío.
- `clear()` — Elimina todos los elementos.
- `resize(n)` — Cambia el tamaño del vector a n elementos.
- `insert(it, x)` — Inserta el valor x en la posición indicada por el iterador `it`.
- `erase(it)` — Elimina el elemento en la posición indicada por el iterador.
- `front()` y `back()` — Devuelven el primer y último elemento.
- `begin()` y `end()` — Similares a los anteriores, pero devuelven un iterador que apunta al primer y último elemento. El iterador es simplemente una generalización de un puntero que permite acceder a todos los elementos de una estructura de datos de forma ordenada y tiene sobrecargados los operadores $++$ y $--$.
- `assign(n, x)` — Rellena el vector con n copias del valor x (reemplaza su contenido).

Ejemplo Práctico:

Listing 3: Uso de funciones útiles del vector

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> v = {1, 2, 3};

    v.push_back(4);       // Anadir 4 al final
    v.pop_back();         // Eliminar el ultimo (4)
    v.insert(v.begin(), 0); // Insertar 0 al principio

    cout << "Elementos del vector: ";
    for (int x : v) cout << x << " ";
    cout << endl;

    cout << "Tamano actual: " << v.size() << endl;
    cout << "Primer elemento: " << v.front() << ", Ultimo
        : " << v.back() << endl;

    return 0;
}
```

1.1. Matriz Dinámica usando vector

Una **matriz dinámica** en C++ se puede representar usando un vector de vectores, lo que permite trabajar con estructuras 2D de tamaño variable en tiempo de ejecución.

Declaración y Uso:

Listing 4: Crear y recorrer una matriz dinámica

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int filas = 3, columnas = 4;

    // Crear matriz 3x4 inicializada en 0
    vector<vector<int>> matriz(filas, vector<int>(
        columnas, 0));

```

```
    // Asignar valores
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            matriz[i][j] = i + j;
        }
    }

    // Imprimir la matriz
    cout << "Matriz:" << endl;
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            cout << matriz[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Ventajas de vector:

- No es necesario conocer el tamaño en tiempo de compilación.
- Puedes redimensionar cualquier fila de forma independiente.
- Acceso indexado igual que con arrays tradicionales.

Desventajas:

- Acceso ligeramente más lento debido al nivel adicional de indirección.
- No es tan eficiente en memoria como una matriz contigua (tipo C).

Una matriz dinámica es especialmente útil cuando se leen datos desde entrada estándar o se trabaja con estructuras tipo grafo, tableros, o mallas.