Los Arreglos unidimensionales (arrays)

Los arreglos unidimensionales (arrays) en C++ son colecciones de elementos del mismo tipo que se almacenan en *ubicaciones de memoria contiguas*. Los arrays permiten acceder a los elementos mediante índices y son útiles para organizar y manipular datos de forma eficiente.

Declaración y Creación de Arrays
Para declarar un array en C++, se debe *especificar el tipo de dato que contendrá*, seguido del nombre del array y el número de elementos entre corchetes []. Aquí hay un ejemplo básico:

```
int numeros[5]; // Declara un array de 5 enteros
```

En este caso, se crea un array llamado numeros que puede almacenar 5 enteros. Los índices en un array comienzan desde **0**, por lo que el primer elemento es numeros[0] y el último es numeros[4].

Inicialización de Arrays
Se pueden inicializar los elementos del array en el momento de su declaración:

```
int numeros[5] = {1, 2, 3, 4, 5}; // Inicializa el array con valores
```

Si no se proporciona un valor para cada elemento, los elementos no inicializados se llenan con ceros (para tipos básicos como int):

```
int numeros[5] = {1, 2}; // Los elementos restantes serán 0: {1, 2, 0, 0, 0}
```

Operaciones Básicas con Arrays .
1. Acceder a Elementos:
Puedes acceder a los elementos del array usando el índice. Como mencionamos, el índice comienza en 0:

```
int primerElemento = numeros[0]; // Accede al primer elemento
int ultimoElemento = numeros[4]; // Accede al quinto elemento
```

2. Modificar Elementos:
También puedes modificar los valores de los elementos de un array usando su índice:

```
numeros[0] = 10; // Cambia el primer valor a 10
numeros[4] = 50; // Cambia el último valor a 50
```

3. Recorrer un Array con un Ciclo for:
Uno de los usos más comunes de los arrays es recorrerlos con un bucle, por ejemplo, para imprimir o procesar sus valores:

```
for (int i = 0; i < 5; i++) {
     std::cout << numeros[i] << " "; // Imprime cada elemento del array
}
```

4. Suma de Elementos de un Array:
Se puede sumar los valores de los elementos del array de la siguiente manera:

```
int suma = 0;
for (int i = 0; i < 5; i++) {
     suma += numeros[i]; // Suma cada elemento
}
std::cout << "Suma de elementos: " << suma << std::endl;
```

Usos Prácticos de los Arrays
Los arrays se utilizan en muchos escenarios donde es necesario manejar colecciones de datos homogéneos. Algunos ejemplos de uso práctico incluyen:

Almacenamiento de Datos de Sensores: Si estás trabajando en un proyecto de hardware, puedes usar arrays para almacenar valores obtenidos de sensores, *como temperaturas, presiones, etc.*

Almacenamiento de Calificaciones: En un sistema académico, podrías usar un array para almacenar las calificaciones de un estudiante:

```
float calificaciones[3] = {85.5, 90.0, 78.2}; // Almacena 3 calificaciones
```

Procesar Datos Estadísticos: Arrays se utilizan para realizar análisis estadísticos, como encontrar el promedio, el valor máximo o mínimo en una serie de datos.

Ejemplo Completo: Cálculo de Promedio y Valor Máximo

```
#include <iostream>

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

     // Calcular el promedio de los elementos del array
     int suma = 0;
     for (int i = 0; i < 5; i++) {
          suma += numeros[i];
     }
     float promedio = static_cast<float>(suma) / 5;
     std::cout << "Promedio: " << promedio << std::endl;

     // Encontrar el valor máximo en el array
     int max = numeros[0];
     for (int i = 1; i < 5; i++) {
          if (numeros[i] > max) {
               max = numeros[i];
          }
     }
     std::cout << "Valor máximo: " << max << std::endl;

     return 0;
}
```

Introducir datos

1. Asignación Directa

Especificas los valores al momento de declarar el arreglo.

```
#include <iostream>

int main() {

    int arreglo[5] = {10, 20, 30, 40, 50}; // Asignación directa

    for (int i = 0; i < 5; i++) {

        std::cout << arreglo[i] << " ";

    }

    return 0;

}
```

2. Inicialización Posterior

Primero declaras el arreglo y luego asignas valores por posición.

```
#include <iostream>


int main() {

    int arreglo[5];  // Declaración

    arreglo[0] = 10;

    arreglo[1] = 20;

    arreglo[2] = 30;

    arreglo[3] = 40;

    arreglo[4] = 50; // Asignación manual


    for (int i = 0; i < 5; i++) {

        std::cout << arreglo[i] << " ";

    }

    return 0;
```

```
}
```

3. Entrada Manual con std::cin

El usuario ingresa los valores del arreglo en tiempo de ejecución.

```
#include <iostream>

int main() {
    int arreglo[5];
    std::cout << "Ingresa 5 números:\n";
    for (int i = 0; i < 5; i++) {
        std::cin >> arreglo[i];
    }

    std::cout << "Los valores ingresados son:\n";
    for (int i = 0; i < 5; i++) {
        std::cout << arreglo[i] << " ";
    }
    return 0;
}
```

4. Generación Aleatoria

Puedes usar la biblioteca <random> para llenar el arreglo con números generados aleatoriamente.

```
#include <iostream>
#include <random>

int main() {
    int arreglo[5];
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(1, 100); // Valores entre 1 y 100

    for (int i = 0; i < 5; i++) {
        arreglo[i] = dist(gen); // Generación aleatoria
    }

    std::cout << "Valores aleatorios generados:\n";
    for (int i = 0; i < 5; i++) {
        std::cout << arreglo[i] << " ";
    }
    return 0;
}
```

5. Uso de Funciones

Llenas el arreglo con valores mediante una función personalizada.

```
#include <iostream>
void llenarArreglo(int arr[], int tamaño) {
    for (int i = 0; i < tamaño; i++) {
```

```
        std::cout << "Ingresa el valor para la posición " << i << ": ";
        std::cin >> arr[i];
    }
}

void mostrarArreglo(int arr[], int tamaño) {
    for (int i = 0; i < tamaño; i++) {
        std::cout << arr[i] << " ";
    }
}

int main() {
    int arreglo[5];
    llenarArreglo(arreglo, 5);
    std::cout << "Los valores ingresados son:\n";
    mostrarArreglo(arreglo, 5);
    return 0;
}
```

6. Llenar con un Valor Específico

Usas un bucle para asignar el mismo valor a todas las posiciones del arreglo.

```
#include <iostream>
int main() {
    int arreglo[5];
    for (int i = 0; i < 5; i++) {
        arreglo[i] = 42; // Todos los elementos tienen el valor 42
    }
    std::cout << "Valores del arreglo:\n";
    for (int i = 0; i < 5; i++) {
        std::cout << arreglo[i] << " ";
    }
    return 0;
}
```

7. Uso de Rango de Inicialización (C++11 en adelante)

Si trabajas con un compilador moderno y sabes los valores de antemano.

```
#include <iostream>
#include <array> // Biblioteca para std::array
int main() {
    std::array<int, 5> arreglo = {10, 20, 30, 40, 50};
    std::cout << "Valores del arreglo:\n";
    for (const auto& val : arreglo) {
        std::cout << val << " ";
    }
    return 0;
```

```
}
```

8. Uso de la Biblioteca std::fill

Para llenar un arreglo con el mismo valor.

```
#include <iostream>
#include <algorithm> // Para std::fill
int main() {
    int arreglo[5];
    std::fill(arreglo, arreglo + 5, 99); // Llena todo con 99

    std::cout << "Valores del arreglo:\n";
    for (int i = 0; i < 5; i++) {
        std::cout << arreglo[i] << " ";
    }
    return 0;
}
```

Evitar la sobre escritura en posiciones:

1. Usar un arreglo adicional como bandera
Puedes usar un segundo arreglo de tipo bool que indique si una posición está ocupada:

```
#include <iostream>

int main() {
    int numeros[5] = {0};         // Arreglo principal
    bool ocupado[5] = {false};    // Arreglo para marcar posiciones usadas

    // Llenar valores iniciales en las posiciones 0 y 3
    numeros[0] = 10;
    ocupado[0] = true;

    numeros[3] = 30;
    ocupado[3] = true;

    // Intentar llenar valores en el arreglo
    for (int i = 0; i < 5; ++i) {
        if (!ocupado[i]) { // Si la posición no está ocupada
            std::cout << "Ingrese un valor para la posición " << i << ": ";
            std::cin >> numeros[i];
            ocupado[i] = true; // Marcar como ocupada
        } else {
            std::cout << "La posición " << i << " ya está ocupada y no puede ser modificada.\n";
        }
    }

    // Mostrar los valores finales del arreglo
    std::cout << "Valores del arreglo:\n";
    for (int i = 0; i < 5; ++i) {
        std::cout << "numeros[" << i << "] = " << numeros[i] << '\n';
    }

    return 0;
}
```

2. Usar un valor especial como indicador
Otra técnica es asignar un valor "especial" a las posiciones vacías, como -1 (o cualquier otro valor que sepas que no será usado como dato válido):

```
#include <iostream>
#include <limits>

int main() {
    const int VACIO = std::numeric_limits<int>::min(); // Usar el valor más pequeño posible
    int numeros[5] = {VACIO, VACIO, VACIO, VACIO, VACIO}; // Inicializar el arreglo como "vacío"
```

```
    // Llenar valores iniciales en las posiciones 0 y 3
    numeros[0] = 10;
    numeros[3] = 30;

    // Intentar llenar valores en el arreglo
    for (int i = 0; i < 5; ++i) {
        if (numeros[i] == VACIO) { // Si la posición está "vacía"
            std::cout << "Ingrese un valor para la posición " << i << ": ";
            std::cin >> numeros[i];
        } else {
            std::cout << "La posición " << i << " ya tiene el valor " << numeros[i] << " y no
puede ser modificada.\n";
        }
    }

    // Mostrar los valores finales del arreglo
    std::cout << "Valores del arreglo:\n";
    for (int i = 0; i < 5; ++i) {
        std::cout << "numeros[" << i << "] = " << numeros[i] << '\n';
    }

    return 0;
}
```

3. Usar un contenedor dinámico (std::vector)

Si necesitas más flexibilidad, puedes usar un vector de la STL. Aunque no tiene un mecanismo integrado para "proteger" posiciones, puedes combinarlo con las estrategias anteriores:

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

int main() {
    std::vector<int> numeros(5, -1); // Inicializar vector con -1 como indicador de vacío

    // Llenar valores iniciales en las posiciones 0 y 3
    numeros[0] = 10;
    numeros[3] = 30;

    // Intentar llenar valores en el vector
    for (size_t i = 0; i < numeros.size(); ++i) {
        if (numeros[i] == -1) { // Si la posición está "vacía"
            std::cout << "Ingrese un valor para la posición " << i << ": ";
            std::cin >> numeros[i];
        } else {
            std::cout << "La posición " << i << " ya tiene el valor " << numeros[i] << " y no
puede ser modificada.\n";
        }
    }

    // Mostrar los valores finales del vector
    std::cout << "Valores del vector:\n";
    for (size_t i = 0; i < numeros.size(); ++i) {
        std::cout << "numeros[" << i << "] = " << numeros[i] << '\n';
    }

    return 0;
}
```

Consideraciones

Usar un arreglo adicional como bandera es útil si no puedes cambiar los valores en el arreglo principal (por ejemplo, si todos los valores posibles son válidos).
Usar un valor especial como indicador es más sencillo pero depende de que ese valor no pueda aparecer como dato real.
Usar **std::vector** te da más flexibilidad en programas avanzados.

Cosas Importantes a Tener en Cuenta .

Tamaño del Array: En C++, el tamaño del array debe ser conocido en tiempo de compilación, lo que significa que el tamaño del array no puede cambiar dinámicamente. Para arrays dinámicos, se utilizan otras estructuras de datos como std::vector.

Desbordamiento de Array: Debes tener cuidado de no acceder a elementos *fuera del rango* del array, ya que podría causar comportamiento indefinido.

Arrays Multidimensionales: Además de los arrays unidimensionales, C++ también permite arrays multidimensionales, como matrices. Pero esto es un tema aparte.

Conclusión

Los arrays son una estructura fundamental en C++ para *almacenar y trabajar con colecciones de datos*. Son útiles en una variedad de situaciones donde se requiere acceso rápido a múltiples valores del mismo tipo y se utilizan ampliamente en algoritmos y estructuras de datos más avanzadas.