



UNIVERSIDAD NACIONAL DEL ALTIPLANO



Facultad de Ingeniería Estadística e Informática

Alumno: Paul Edward Mamani Vilca

Codigo: 241199

Curso: Estructura de Datos

Docente: Fred Torres Cruz

Índice

- Índice
- Introducción
- I Operadores
- II Estructuras De Control
- III Funciones
- IV Arrays
- V Punteros y Referencias
- VI Operadores tipo & y *
- VII Listas Enlazadas
- VIII Listas Doblemente Enlazadas
- IX Listas Circulares
- X Colas (Queues)
- XI Pilas (Stacks)
- XII Recursión
- Conclusión

Introducción

Las **estructuras de datos** permiten almacenar, organizar y manipular información eficientemente. Son la base de la resolución de problemas computacionales complejos.

Ejemplo 1

Crear una lista enlazada que almacene nombres y los imprima en **C++**

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

struct Nodo {
    string nombre;
```

```

    Nodo* siguiente;
};

int main() {
    Nodo* primero = new Nodo{"Ana", nullptr};
    primero->siguiente = new Nodo{"Luis", nullptr};
    primero->siguiente->siguiente = new Nodo{"Carlos", nullptr};

    Nodo* actual = primero;
    while (actual != nullptr) {
        cout << actual->nombre << endl;
        actual = actual->siguiente;
    }

    return 0;
}

```

Ejemplo 2

Crear un array de enteros y calcular su suma en **Python**

```

class Nodo:
    def __init__(self, nombre):
        self.nombre = nombre
        self.siguiente = None

# Crear nodos
n1 = Nodo("Ana")
n2 = Nodo("Luis")
n3 = Nodo("Carlos")

# Enlazar nodos
n1.siguiente = n2
n2.siguiente = n3

# Recorrer lista
actual = n1
while actual:
    print(actual.nombre)
    actual = actual.siguiente

```

Ejemplo 3

Crear un array de enteros y calcular su suma en **Java**

```

class Nodo {
    String nombre;
    Nodo siguiente;
}

```

```
Nodo(String nombre) {
    this.nombre = nombre;
    this.siguiente = null;
}

}

public class ListaEnlazada {
    public static void main(String[] args) {
        Nodo primero = new Nodo("Ana");
        primero.siguiente = new Nodo("Luis");
        primero.siguiente.siguiente = new Nodo("Carlos");

        Nodo actual = primero;
        while (actual != null) {
            System.out.println(actual.nombre);
            actual = actual.siguiente;
        }
    }
}
```

Ejemplo 4

Crear un array de enteros y calcular su suma en **JavaScript**

```
class Nodo {
    constructor(nombre) {
        this.nombre = nombre;
        this.siguiente = null;
    }
}

// Crear nodos
let n1 = new Nodo("Ana");
let n2 = new Nodo("Luis");
let n3 = new Nodo("Carlos");

// Enlazar nodos
n1.siguiente = n2;
n2.siguiente = n3;

// Recorrer lista
let actual = n1;
while (actual !== null) {
    console.log(actual.nombre);
    actual = actual.siguiente;
}
```

*Ejemplo 5

Crear un array de enteros y calcular su suma en **C#**

```
using System;

class Nodo {
    public string Nombre;
    public Nodo Siguiente;

    public Nodo(string nombre) {
        Nombre = nombre;
        Siguiente = null;
    }
}

class Program {
    static void Main() {
        Nodo primero = new Nodo("Ana");
        primero.Siguiente = new Nodo("Luis");
        primero.Siguiente.Siguiente = new Nodo("Carlos");

        Nodo actual = primero;
        while (actual != null) {
            Console.WriteLine(actual.Nombre);
            actual = actual.Siguiente;
        }
    }
}
```

conclusión

Los ejemplos presentados demuestran claramente que las **estructuras de datos**, como las listas enlazadas, son fundamentales en la programación, independientemente del lenguaje utilizado. En todos los casos, se ha implementado una lista enlazada simple que almacena nombres y los imprime, reflejando un patrón común: **un nodo con un valor y una referencia al siguiente**.

Aunque la sintaxis varía entre **C++**, **Python**, **Java**, **JavaScript** y **C#**, la lógica de construcción y recorrido de una lista enlazada permanece constante. Esto pone en evidencia que **la estructura mental del programador es más importante que el lenguaje en sí**.

Puntos Clave

- **Reutilización del concepto:** Todos los lenguajes permiten implementar listas enlazadas, lo cual demuestra la **versatilidad del concepto de nodos y punteros/referencias**.
- **Pensamiento estructurado:** Cada código refleja la importancia de **dividir datos en elementos (nodos)** conectados lógicamente, favoreciendo el diseño de algoritmos eficientes.
- **Independencia del lenguaje:** Una estructura bien entendida puede adaptarse a múltiples lenguajes, lo que fortalece el perfil de un programador polivalente.

Reflexión Final

Aprender estructuras de datos no se trata solo de programar, sino de **entender cómo los datos se organizan y se mueven internamente**. Esto te permite desarrollar software más eficiente, escalable y mantenible, sin importar si usas **C++**, **Python**, **Java** o cualquier otro lenguaje moderno.

Dominar estructuras de datos es dominar el corazón de la programación.

I Operadores

En programación, los **operadores** son símbolos que permiten manipular datos y variables a través de cálculos, comparaciones o asignaciones. Funcionan como el "vocabulario" matemático y lógico que permite expresar acciones y decisiones dentro del código. Hay operadores de distintos tipos: **aritméticos** (+, -, *, /), **lógicos** (&&, ||, !), **relacionales** (>, <, ==) y **de asignación** (=, +=, -=).

El uso adecuado de los operadores permite crear expresiones poderosas y precisas, capaces de tomar decisiones automáticas o ejecutar cálculos complejos con una sola línea de código.

Ejemplo:

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

int main() {
    int a = 10, b = 3;
    cout << "Suma: " << a + b << endl;
    cout << "Resta: " << a - b << endl;
    cout << "Multiplicación: " << a * b << endl;
    cout << "División: " << a / b << endl;
    return 0;
}
```

Explicación paso a paso:

Se declaran dos variables enteras: *a* con valor 10 y *b* con valor 3.

Se realizan cuatro operaciones aritméticas básicas:

$a + b \rightarrow$ Suma: 13

$a - b \rightarrow$ Resta: 7

$a * b \rightarrow$ Multiplicación: 30

$a / b \rightarrow$ División entera: 3 (porque ambos son int, se descarta el decimal)

Los resultados se imprimen en la consola usando cout.

Resultado esperado en pantalla:

```
Suma: 13
Resta: 7
Multiplicación: 30
División: 3
```

Ideas Clave

Los operadores permiten transformar valores y construir lógica. Son la base de cualquier operación matemática o decisión condicional dentro de un programa.

Practica 1

Calcular el promedio de 3 notas.

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

int main() {
    float n1 = 15.5, n2 = 18.0, n3 = 17.5;
    float promedio = (n1 + n2 + n3) / 3;
    cout << "Promedio: " << promedio << endl;
    return 0;
}
```

Practica 2

Verificar si un número es par.

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

int main() {
    int num = 8;
    if (num % 2 == 0)
        cout << "Es par" << endl;
    else
        cout << "Es impar" << endl;
    return 0;
}
```

II Estructuras De Control

Las **estructuras de control** son la columna vertebral de la lógica en la programación. Permiten decidir qué instrucciones se ejecutan, en qué momento y cuántas veces. Gracias a ellas, un programa puede responder a condiciones cambiantes o repetir procesos sin intervención humana.

Existen dos grandes tipos:

- **Condicionales**, como `if`, `else` o `switch`, que permiten elegir entre distintas rutas de ejecución.
- **Repetitivas**, como `for`, `while` y `do-while`, que permiten ejecutar bloques de código múltiples veces, controlando así bucles y ciclos automáticos.

Piensa en esto: *Las estructuras de control le dan inteligencia al software. Le permiten pensar, decidir y repetir por sí solo.*

Ejemplo:

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

int main() {
    int edad = 20;
    if (edad >= 18) {
        cout << "Mayor de edad" << endl;
    } else {
        cout << "Menor de edad" << endl;
    }
    return 0;
}
```

Explicacion paso a paso:

1. Se declara una variable `edad` con el valor 20.
2. La condición `edad >= 18` se evalúa como verdadera.
3. Como es verdadera, se ejecuta `cout << "Mayor de edad"`.

Resultado Esperado en pantalla:

```
Mayor de edad
```

Ideas Clave

Las estructuras de control le dan lógica al código. Le permiten decidir qué hacer según distintas condiciones.

Practica 1

Mostrar los números del 1 al 5.

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

int main() {
    for (int i = 1; i <= 5; i++) {
        cout << i << " ";
    }
    return 0;
}
```

Practica 2

Sumar los primeros 10 números naturales.

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

int main() {
    int suma = 0;
    for (int i = 1; i <= 10; i++) {
        suma += i;
    }
    cout << "Suma total: " << suma << endl;
    return 0;
}
```

Conclusión

Las estructuras de control permiten que un programa no sea simplemente una secuencia lineal de instrucciones, sino una unidad capaz de **tomar decisiones** y **repetir acciones**. Son la esencia de la lógica algorítmica.

Desde un simple `if` hasta bucles anidados, estas estructuras son las que convierten un conjunto de líneas en un programa dinámico, inteligente y útil.

Puntos Clave

- **Decisiones dinámicas:** Permiten elegir qué camino tomar en tiempo de ejecución.
- **Repetición controlada:** Automatizan tareas repetitivas con eficiencia.
- **Flexibilidad lógica:** Hacen que los programas respondan a diferentes condiciones.

Reflexión Final

Las estructuras de control son las que dotan de "mente" al programa. Le permiten **pensar, reaccionar y adaptarse** a diferentes situaciones.

Un programa sin estructuras de control es como un tren sin rieles: avanza, pero no sabe a dónde va.

III Funciones

Una **función** es un bloque de código reutilizable que encapsula una tarea específica. Se comporta como una "caja negra" a la que se le da una entrada (parámetros), realiza una operación interna, y devuelve un resultado (valor de retorno).

El uso de funciones favorece la organización, evita la duplicación de código y hace que un programa grande se divida en módulos más pequeños, claros y manejables. Esto es esencial para desarrollar software escalable, mantenible y más legible.

Así de simple: *Una función es como una receta: la usas, la repites y sabes siempre qué resultado vas a obtener.*

Ejemplo:

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

int sumar(int a, int b) {
    return a + b;
}

int main() {
    cout << "Suma: " << sumar(3, 4) << endl;
    return 0;
}
```

Explicación paso a paso:

1. Se define una función llamada *sumar* que recibe dos enteros y devuelve su suma.
2. En *main()*, se llama a *sumar(3, 4)* y se imprime el resultado.

Resultado esperado en pantalla:

```
Suma: 7
```

Ideas Clave Las funciones dividen un problema en pequeñas tareas que se pueden resolver de forma modular y reutilizable.

Practica 1

Crear una función que devuelva el doble de un número.

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

int doble(int x) {
    return x * 2;
}

int main() {
    cout << "El doble es: " << doble(5) << endl;
    return 0;
}
```

Practica 2

Crear una función que calcule el área de un triángulo.

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

float areaTriangulo(float base, float altura) {
    return (base * altura) / 2;
}

int main() {
    cout << "Área: " << areaTriangulo(10, 5) << endl;
    return 0;
}
```

Conclusión

Las funciones son bloques independientes de código diseñados para realizar tareas específicas. Su existencia no solo mejora la organización del programa, sino que también fomenta la reutilización, la claridad y el mantenimiento del software.

En cualquier lenguaje, las funciones permiten dividir y conquistar: se resuelve un problema complejo en pequeñas partes manejables.

Puntos Clave

- **Reutilización:** Permiten usar el mismo bloque de código varias veces.
- **Mantenimiento:** Facilitan la corrección y actualización del programa.
- **Claridad estructural:** Hacen que el código sea más legible y organizado.

Reflexión Final

Las funciones no solo resuelven tareas, sino que enseñan a **pensar modularmente**. Son pequeñas soluciones que, al combinarse, logran grandes resultados.

Una función bien diseñada es como una herramienta bien forjada: precisa, útil y duradera.

IV Arrays

Un **array** (o arreglo) es una estructura de datos que almacena una colección de elementos del mismo tipo en posiciones contiguas de memoria. Cada elemento del array puede accederse mediante un índice numérico.

Su uso es ideal cuando necesitas trabajar con conjuntos de datos similares, como una lista de números, nombres o registros. Los arrays permiten recorrer todos los elementos usando bucles, lo que los convierte en una de las estructuras más utilizadas.

En pocas palabras: *Un array es como una fila de casillas donde cada una guarda un dato listo para ser usado.*

Ejemplo:

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

int main() {
    int numeros[3] = {1, 2, 3};
    for (int i = 0; i < 3; i++) {
        cout << numeros[i] << endl;
    }
    return 0;
}
```

Explicación paso a paso:

1. Se declara un array `numeros` con 3 *elementos*.
2. Se utiliza un bucle `for` para recorrer cada índice del array.
3. En cada iteración, se imprime el valor correspondiente.

Resultado esperado en pantalla:

```
1
2
3
```

Ideas Clave

Los arrays permiten trabajar con colecciones ordenadas de datos, haciendo posible el uso de ciclos para procesarlos.

Practica 1

Calcular la suma de los elementos de un array.

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

int main() {
    int numeros[5] = {2, 4, 6, 8, 10}, suma = 0;
    for (int i = 0; i < 5; i++) {
        suma += numeros[i];
    }
    cout << "Suma total: " << suma << endl;
    return 0;
}
```

Practica 2

Encontrar el mayor número de un array.

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

int main() {
    int nums[5] = {10, 22, 5, 30, 12};
    int mayor = nums[0];
    for (int i = 1; i < 5; i++) {
        if (nums[i] > mayor) mayor = nums[i];
    }
    cout << "Mayor: " << mayor << endl;
    return 0;
}
```

Conclusión

Los arrays son estructuras de datos que permiten almacenar múltiples elementos del mismo tipo en una sola variable, accediendo a ellos mediante índices. Representan orden, eficiencia y control en el manejo de grandes cantidades de datos.

Usados en casi todos los lenguajes, son la base de estructuras más complejas y algoritmos eficientes.

Puntos Clave

- **Eficiencia:** Acceder por índice es rápido y directo.
- **Organización:** Permiten mantener los datos ordenados secuencialmente.
- **Fundamento de estructuras mayores:** Son la base de matrices, vectores, listas, etc.

Reflexión Final

Dominar los arrays es aprender a **ordenar el caos de los datos**, a clasificarlos, accederlos y manipularlos con precisión.

Un buen uso del array puede ser la diferencia entre un código lento y uno eficiente.

V Punteros y Referencias

Los **punteros** y **referencias** son herramientas que permiten trabajar directamente con la memoria del computador. Un puntero es una variable que almacena la dirección de otra variable, mientras que una referencia es un alias directo a una variable existente.

Con estas herramientas puedes optimizar el rendimiento del programa, acceder a estructuras dinámicas y manipular datos de manera más avanzada. Aunque su uso requiere mayor cuidado, ofrecen un poder de control increíble en el desarrollo de estructuras como listas, árboles y más.

Recuerda esto: *Los punteros te dan acceso al corazón de la memoria. Son el poder y el riesgo en una misma línea.*

Ejemplo:

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

int main() {
    int a = 10;
    int* ptr = &a;
    cout << "Valor: " << *ptr << endl;
    return 0;
}
```

Explicación paso a paso:

1. Se declara una variable `a` con valor 10.
2. `ptr` almacena la dirección de `a`.
3. Usando `*ptr`, se imprime el valor al que apunta: 10.

Resultado esperado en pantalla:

Valor: 10

Ideas Clave

Los punteros te permiten controlar y modificar datos desde cualquier parte del programa.

Practica 1

Mostrar la dirección de memoria de una variable.

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

int main() {
    int numero = 5;
    cout << "Dirección: " << &numero << endl;
    return 0;
}
```

Practica 2

Cambiar el valor de una variable usando puntero.

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

void modificar(int* x) {
    *x = 99;
}

int main() {
    int valor = 10;
    modificar(&valor);
    cout << "Nuevo valor: " << valor << endl;
    return 0;
}
```

Conclusión

Los punteros y referencias permiten trabajar directamente con la memoria del computador. Gracias a ellos, podemos optimizar recursos, crear estructuras dinámicas y manipular datos de forma más avanzada.

Aunque pueden parecer complejos al inicio, una vez dominados abren un mundo de posibilidades técnicas.

Puntos Clave

- **Acceso directo a memoria:** Permiten manipular variables y estructuras por dirección.
- **Eficiencia en recursos:** Reducen copias innecesarias.
- **Necesarios para estructuras dinámicas:** Como listas, árboles o colas.

Reflexión Final

Trabajar con punteros es como manipular los circuitos internos de la máquina: poderoso, pero requiere responsabilidad y precisión.

Quien entiende los punteros, entiende la esencia de cómo funciona la memoria en programación.

VI Operadores tipo & y *

El operador `&` se usa para obtener la dirección de memoria de una variable, es decir, su ubicación exacta en la RAM. El operador `*` permite acceder al valor que se encuentra en esa dirección (desreferenciación). Ambos son indispensables para trabajar con punteros en lenguajes como C y C++.

Estos operadores hacen posible pasar variables por referencia, crear estructuras dinámicas y manipular datos de forma eficiente sin hacer copias innecesarias.

Idea clave: Con `&` y `*` ya no solo usas los datos... accedes a dónde viven y cómo se mueven dentro de la máquina.

Ejemplo:

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

int main() {
    int num = 20;
    int* p = &num;

    cout << "Dirección: " << p << endl;
    cout << "Contenido: " << *p << endl;
    return 0;
}
```

Explicación paso a paso:

1. `p` almacena la dirección de `num`.
2. Se imprime la dirección (`p`) y el contenido (`*p`), que es 20.

Resultado esperado en pantalla:

Dirección: 0x... (varía)
Contenido: 20

Ideas Clave

Los operadores `&` y `*` son esenciales para trabajar con punteros, permitiendo acceder a direcciones de memoria y manipular valores directamente.

Practica 1

Multiplicar dos valores usando punteros.

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

int multiplicar(int* a, int* b) {
    return (*a) * (*b);
}

int main() {
    int x = 4, y = 5;
    cout << "Multiplicación: " << multiplicar(&x, &y) << endl;
    return 0;
}
```

Practica 2

Intercambiar dos variables usando punteros.

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

void intercambiar(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int a = 1, b = 2;
    intercambiar(&a, &b);
    cout << "a = " << a << ", b = " << b << endl;
    return 0;
}
```


Conclusión

Los operadores `&` y `*` son las llaves del manejo de punteros en lenguajes como C++. Con ellos accedemos directamente a la dirección de memoria (`&`) y a su contenido (`*`), permitiendo modificar valores desde su origen.

Son vitales en cualquier estructura dinámica que implique uso de memoria.

Puntos Clave

- **Dirección y contenido:** Permiten trabajar tanto con la ubicación como con el valor.
- **Imprescindibles en punteros:** Son base para entender y usar punteros correctamente.
- **Acceso eficiente:** Facilitan el uso directo y rápido de recursos.

Reflexión Final

Estos operadores no son simples símbolos: son la representación directa del poder de acceder y modificar lo más profundo del sistema.

Aprender `&` y `` es como aprender a programar desde las entrañas del computador.*

VII Listas Enlazadas

Una **lista enlazada** es una estructura de datos dinámica que consiste en nodos. Cada nodo contiene un dato y una referencia (puntero) al siguiente nodo de la lista. A diferencia de los arrays, las listas enlazadas no requieren un bloque contiguo de memoria y permiten insertar o eliminar elementos fácilmente en cualquier posición.

Existen variantes como:

- Lista simplemente enlazada (apunta al siguiente nodo),
- Lista doblemente enlazada (apunta al anterior y al siguiente),
- Lista circular (el último nodo apunta al primero).

Metáfora perfecta: *Una lista enlazada es como una cadena de vagones que puedes expandir o reducir en cualquier momento, sin mover todo el tren.*

Ejemplo:

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

struct Nodo {
    int dato;
    Nodo* siguiente;
};
```

```
int main() {
    Nodo* primero = new Nodo{10, nullptr};
    primero->siguiente = new Nodo{20, nullptr};

    Nodo* actual = primero;
    while (actual != nullptr) {
        cout << actual->dato << endl;
        actual = actual->siguiente;
    }
    return 0;
}
```

Explicación paso a paso:

1. Se crean dos *nodos* con valores 10 y 20.
2. Se conecta el primer nodo al segundo usando el puntero *siguiente*.
3. Se recorre e imprime la lista desde el primero hasta el último *nodo*.

Resultado esperado en pantalla:

```
10
20
```

Ideas Clave

Las listas enlazadas son estructuras dinámicas que permiten almacenar datos de forma flexible, sin necesidad de un tamaño fijo como en los arrays. Cada nodo se conecta al siguiente, formando una cadena que puede crecer o reducirse según sea necesario.

Practica 1

Insertar un nodo al final de la lista.

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

struct Nodo {
    int valor;
    Nodo* siguiente;
};

int main() {
    Nodo* cabeza = new Nodo{1, nullptr};
    Nodo* nuevo = new Nodo{2, nullptr};
    cabeza->siguiente = nuevo;

    Nodo* actual = cabeza;
```

```
while (actual != nullptr) {
    cout << actual->valor << " ";
    actual = actual->siguiente;
}
return 0;
}
```

Practica 2

Contar cuántos nodos hay en la lista.

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

struct Nodo {
    int valor;
    Nodo* siguiente;
};

int main() {
    Nodo* cabeza = new Nodo{1, nullptr};
    cabeza->siguiente = new Nodo{2, nullptr};
    cabeza->siguiente->siguiente = new Nodo{3, nullptr};

    int contador = 0;
    Nodo* actual = cabeza;
    while (actual != nullptr) {
        contador++;
        actual = actual->siguiente;
    }
    cout << "Nodos: " << contador << endl;
    return 0;
}
```

Conclusión

Las listas enlazadas son estructuras dinámicas formadas por nodos, donde cada uno contiene un dato y una referencia al siguiente. A diferencia de los arrays, su tamaño no es fijo y permiten insertar o eliminar elementos con facilidad.

Son la base para implementar pilas, colas, árboles y muchas otras estructuras avanzadas.

Puntos Clave

- **Estructura dinámica:** Pueden crecer o reducirse durante la ejecución.
- **Eficiencia en inserciones:** No necesitan mover otros elementos como en arrays.

- **Uso avanzado:** Son fundamentales en estructuras como árboles y grafos.

Reflexión Final

Entender listas enlazadas es entender cómo construir estructuras vivas, que cambian, se adaptan y se reorganizan en tiempo real.

Una lista enlazada bien implementada es como un organismo digital: dinámico, flexible y eficiente.

VIII Listas Doblemente Enlazadas

Concepto: Una **lista doblemente enlazada** es una estructura dinámica de datos en la que cada nodo contiene tres partes: un dato, un puntero al nodo siguiente y un puntero al nodo anterior. Esta doble conexión permite recorrer la lista en ambos sentidos: hacia adelante y hacia atrás.

Esta estructura es muy útil cuando se necesita un acceso más flexible a los datos, por ejemplo, en editores de texto, navegadores con historial hacia adelante y atrás, o listas de reproducción donde puedes saltar de una canción a otra.

Idea esencial: *Las listas dobles te permiten navegar hacia atrás o hacia adelante sin perderte. Es como tener un mapa bidireccional de tus datos.*

Ejemplo:

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

struct Nodo {
    int dato;
    Nodo* anterior;
    Nodo* siguiente;
};

int main() {
    Nodo* primero = new Nodo{10, nullptr, nullptr};
    Nodo* segundo = new Nodo{20, primero, nullptr};
    primero->siguiente = segundo;

    Nodo* actual = segundo;
    while (actual != nullptr) {
        cout << actual->dato << endl;
        actual = actual->anterior;
    }
    return 0;
}
```

Explicación paso a paso:

1. Se crean *dos nodos* conectados en ambas direcciones.
2. Se imprime la lista desde el último nodo hacia el primero.

Resultado esperado en pantalla:

20

10

Ideas Clave

Las listas doblemente enlazadas permiten un recorrido bidireccional, facilitando la navegación entre nodos en ambas direcciones. Cada nodo tiene un puntero al siguiente y al anterior, lo que las hace más versátiles que las listas simplemente enlazadas.

Practica 1*Recorrer lista doble hacia adelante.*

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

struct Nodo {
    int valor;
    Nodo* ant;
    Nodo* sig;
};

int main() {
    Nodo* n1 = new Nodo{1, nullptr, nullptr};
    Nodo* n2 = new Nodo{2, n1, nullptr};
    n1->sig = n2;

    Nodo* actual = n1;
    while (actual != nullptr) {
        cout << actual->valor << " ";
        actual = actual->sig;
    }
    return 0;
}
```

Practica 2*Insertar nodo al final de una lista doble.*

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

```
struct Nodo {
    int dato;
    Nodo* ant;
    Nodo* sig;
};

int main() {
    Nodo* head = new Nodo{5, nullptr, nullptr};
    Nodo* nuevo = new Nodo{10, head, nullptr};
    head->sig = nuevo;

    Nodo* actual = head;
    while (actual != nullptr) {
        cout << actual->dato << " ";
        actual = actual->sig;
    }
    return 0;
}
```

Conclusión:

Las listas doblemente enlazadas permiten recorrer la información en ambas direcciones, ya que cada nodo contiene un puntero al siguiente y otro al anterior. Son útiles cuando se necesita moverse hacia adelante y hacia atrás sin restricciones, como en reproductores de medios o navegadores web.

Puntos Clave

- **Recorrido bidireccional:** Permite navegar tanto hacia adelante como hacia atrás.
- **Mayor flexibilidad:** Ideal para insertar o eliminar nodos desde cualquier parte de la lista.
- **Útiles en interfaces interactivas:** Navegadores, menús, listas de reproducción.

Reflexión Final

Las listas dobles abren una nueva dimensión de control. No solo avanzas: puedes retroceder, editar y reorganizar en tiempo real.

Son como libros digitales con marcapáginas en cada hoja: puedes ir y volver cuantas veces necesites.

IX Listas Circulares

Concepto: Una **lista circular** es una variante de la lista enlazada donde el último nodo apunta nuevamente al primero, formando un ciclo cerrado. Puede ser simplemente circular (un solo puntero al siguiente) o doblemente circular (también con puntero al anterior).

Se utilizan en situaciones donde el recorrido continuo es necesario, como en sistemas operativos (planificación de procesos), reproductores de música con "repetición" o juegos que requieren recorrer jugadores de forma circular.

Pensamiento clave: *Una lista circular nunca se detiene. Ideal para procesos repetitivos o bucles infinitos controlados.*

Ejemplo:

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

struct Nodo {
    int dato;
    Nodo* siguiente;
};

int main() {
    Nodo* primero = new Nodo{1, nullptr};
    Nodo* segundo = new Nodo{2, nullptr};
    Nodo* tercero = new Nodo{3, nullptr};

    primero->siguiente = segundo;
    segundo->siguiente = tercero;
    tercero->siguiente = primero;

    Nodo* actual = primero;
    int i = 0;
    while (i < 5) {
        cout << actual->dato << " ";
        actual = actual->siguiente;
        i++;
    }
    return 0;
}
```

Explicación paso a paso:

1. Tres nodos se *enlazan circularmente*.
2. Se recorre la lista durante 5 iteraciones, repitiendo el ciclo.

Resultado esperado en pantalla:

```
1 2 3 1 2
```

Ideas Clave

Las listas circulares permiten un recorrido infinito, ya que el último nodo apunta de vuelta al primero. Son ideales para situaciones donde se necesita un flujo continuo de datos o tareas, como en sistemas de tiempo real o buffers circulares.

Practica 1

Mostrar elementos de una lista circular una vuelta completa.

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

struct Nodo {
    int valor;
    Nodo* sig;
};

int main() {
    Nodo* a = new Nodo{10, nullptr};
    Nodo* b = new Nodo{20, nullptr};
    Nodo* c = new Nodo{30, nullptr};

    a->sig = b;
    b->sig = c;
    c->sig = a;

    Nodo* actual = a;
    do {
        cout << actual->valor << " ";
        actual = actual->sig;
    } while (actual != a);
    return 0;
}
```

Practica 2

Contar elementos de una lista circular.

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

struct Nodo {
    int dato;
    Nodo* sig;
};

int main() {
    Nodo* n1 = new Nodo{5, nullptr};
    Nodo* n2 = new Nodo{10, nullptr};
    Nodo* n3 = new Nodo{15, nullptr};

    n1->sig = n2;
    n2->sig = n3;
    n3->sig = n1;
}
```



```
int cont = 0;
Nodo* actual = n1;
do {
    cont++;
    actual = actual->sig;
} while (actual != n1);
cout << "Cantidad: " << cont << endl;
return 0;
}
```

Conclusión:

Las listas circulares son una evolución de las listas enlazadas, donde el último nodo apunta nuevamente al primero. Esto genera un ciclo continuo que nunca termina, ideal para sistemas que requieren un flujo sin fin.

Puntos Clave

- **Recorrido cíclico:** Perfecto para procesos que deben reiniciarse automáticamente.
- **Utilidad en sistemas operativos:** Muy usadas en planificación de tareas y buffers circulares.
- **Sin fin aparente:** El usuario nunca ve el final de la lista.

Reflexión Final

Las listas circulares representan continuidad y estabilidad en sistemas que nunca se detienen.

En un mundo donde todo gira, la lista circular es la estructura perfecta.

X Colas (Queues)

Concepto: Una **cola** es una estructura de datos lineal que sigue el principio **FIFO** (*First In, First Out*), es decir, el primer elemento que entra es el primero en salir. Se asemeja a una fila de personas: el primero en llegar es el primero en ser atendido.

Las colas son ampliamente utilizadas en programación de tareas, procesamiento por lotes, buffers de impresión, manejo de solicitudes de red, entre muchos otros casos.

Resumen visual: *Las colas organizan el acceso justo. Nadie se cuela, todos esperan su turno.*

Ejemplo:

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

int main() {
    queue<int> cola;
    cola.push(1);
}
```

```
cola.push(2);
cola.push(3);

while (!cola.empty()) {
    cout << "Elemento: " << cola.front() << endl;
    cola.pop();
}
return 0;
}
```

Explicación paso a paso:

1. Se agregan 3 elementos a la cola.
2. Se imprimen y eliminan en orden de llegada.

Resultado esperado en pantalla:

```
Elemento: 1
Elemento: 2
Elemento: 3
```

Ideas Clave

Las colas son estructuras de datos que gestionan elementos en orden de llegada, siguiendo el principio FIFO. Son ideales para situaciones donde el orden de procesamiento es crucial, como en sistemas de atención al cliente, impresoras o servidores.

Practica 1

Insertar 5 elementos y mostrar el primero.

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

int main() {
    queue<int> q;
    for (int i = 1; i <= 5; i++) q.push(i);
    cout << "Primero: " << q.front() << endl;
    return 0;
}
```

Practica 2

Simular atención de personas en orden.

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

int main() {
    queue<string> atencion;
    atencion.push("Juan");
    atencion.push("Lucía");
    atencion.push("Carlos");

    while (!atencion.empty()) {
        cout << "Atendiendo a: " << atencion.front() << endl;
        atencion.pop();
    }
    return 0;
}
```

Conclusión:

Las colas son estructuras lineales que gestionan datos en orden de llegada, siguiendo la lógica **Primero en Entrar, Primero en Salir (FIFO)**. Son ideales cuando el orden importa, como en impresoras, filas de atención o servidores web.

Puntos Clave

- **FIFO:** El primero en llegar es el primero en salir.
- **Flujo ordenado:** Evita colisiones y mantiene equidad en el procesamiento.
- **Muy comunes:** Se utilizan en redes, impresoras, servicios en línea.

Reflexión Final

Las colas reflejan justicia y organización. Cada elemento espera su turno y obtiene atención cuando le corresponde.

Una cola bien implementada es como una fila organizada en la vida real: justa, ordenada y eficiente.

XI Pilas (Stacks)

Concepto: Una **pila** es una estructura de datos que sigue el principio **LIFO** (*Last In, First Out*), es decir, el último elemento en entrar es el primero en salir. Es como una pila de platos: solo puedes acceder al que está encima.

Son fundamentales en el manejo de llamadas de funciones, deshacer acciones en editores, evaluar expresiones matemáticas y gestionar memoria (stack de ejecución).

Idea potente: *Las pilas son memoria de corto plazo: lo último que haces es lo primero que recuerdas.*

Ejemplo:

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

int main() {
    stack<int> pila;
    pila.push(5);
    pila.push(10);
    pila.push(15);

    while (!pila.empty()) {
        cout << "Elemento: " << pila.top() << endl;
        pila.pop();
    }
    return 0;
}
```

Explicación paso a paso:

1. Se agregan *tres elementos* a la pila.
2. Se *imprimen y eliminan en orden inverso* al que entraron.

Resultado esperado en pantalla:

```
Elemento: 15
Elemento: 10
Elemento: 5
```

Ideas Clave

Las pilas son estructuras de datos que gestionan elementos en orden inverso al que llegaron, siguiendo el principio **Último en Entrar, Primero en Salir (LIFO)**. Son ideales para situaciones donde necesitas recordar lo último que hiciste, como en deshacer acciones o evaluar expresiones.

Practica 1

Apilar 3 números y mostrar el último.

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

int main() {
    stack<int> p;
    p.push(100);
```

```
p.push(200);
p.push(300);
cout << "Último apilado: " << p.top() << endl;
return 0;
}
```

Practica 2

Simular historial de navegación.

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

int main() {
    stack<string> historial;
    historial.push("Inicio");
    historial.push("Página A");
    historial.push("Página B");

    while (!historial.empty()) {
        cout << "Volver a: " << historial.top() << endl;
        historial.pop();
    }
    return 0;
}
```

Conclusión:

Las pilas son estructuras de tipo **Último en Entrar, Primero en Salir (LIFO)**. El último elemento insertado es el primero en ser retirado. Se usan en sistemas de deshacer/rehacer, recorridos de árboles y ejecución de funciones.

Puntos Clave

- **LIFO:** Lo último que entra es lo primero que sale.
- **Control temporal:** Muy útil para manejar secuencias reversibles.
- **Crucial en el stack de ejecución:** Manejo de llamadas de funciones.

Reflexión Final

Las pilas son la memoria inmediata del programa. Son eficientes, controladas y muy intuitivas.

Son como una pila de platos: solo puedes sacar el de arriba... y eso es exactamente lo que necesitas.

XII Recursión

Concepto: La **recursión** es una técnica de programación en la que una función se llama a sí misma para resolver un problema. Cada llamada se resuelve con un caso más pequeño del problema original, hasta llegar a un caso base que se puede resolver directamente.

La recursión es elegante y poderosa. Se utiliza para resolver problemas que pueden dividirse en subproblemas similares, como factoriales, secuencias de Fibonacci, búsqueda binaria, recorridos en árboles, y más.

Frase que lo define: *La recursión es pensar dentro del espejo: cada solución contiene versiones más pequeñas de sí misma.*

Ejemplo:

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

int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

int main() {
    cout << "Factorial de 5: " << factorial(5) << endl;
    return 0;
}
```

Explicación paso a paso:

1. Se define una función *factorial* que se llama a sí misma hasta llegar al caso base $n == 0$.
2. Cada llamada devuelve $n * factorial(n - 1)$.

Resultado esperado en pantalla:

```
Factorial de 5: 120
```

Ideas Clave

La recursión es una técnica donde una función se llama a sí misma para resolver problemas complejos dividiéndolos en partes más simples. Es especialmente útil para problemas que tienen una estructura repetitiva o jerárquica.

Practica 1

Sumar números del 1 al n recursivamente.

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

int suma(int n) {
    if (n == 0) return 0;
    return n + suma(n - 1);
}

int main() {
    cout << "Suma: " << suma(5) << endl;
    return 0;
}
```

Practica 2

Imprimir cuenta regresiva con recursión.

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

void cuenta(int n) {
    if (n == 0) {
        cout << "¡Despegue!" << endl;
        return;
    }
    cout << n << endl;
    cuenta(n - 1);
}

int main() {
    cuenta(5);
    return 0;
}
```

Conclusión:

La recursión es una técnica poderosa en la que una función se llama a sí misma para resolver problemas grandes dividiéndolos en partes más pequeñas. Elegante, compacta y natural para resolver problemas repetitivos o jerárquicos.

Puntos Clave

- **Divide y vencerás:** Divide un problema grande en problemas más simples.
 - **Se usa en árboles, grafos y matemáticas:** Factorial, Fibonacci, recorridos.
 - **Requiere caso base:** Toda función recursiva necesita un fin claro.
-

Reflexión Final

La recursión no es solo código que se repite: es pensamiento estructurado, organizado y profundo.

Recursión es mirar un espejo dentro de otro espejo... y encontrar la solución en el reflejo más pequeño.

Conclusión




A lo largo de este recorrido por las estructuras de datos y conceptos fundamentales de programación en C++, hemos explorado herramientas esenciales que forman la base de cualquier sistema informático. Desde la manipulación de valores con operadores hasta la construcción de algoritmos recursivos complejos, cada tema ha sido una pieza clave para entender cómo piensa una computadora.

Los **operadores** nos permitieron dar instrucciones básicas. Las **estructuras de control** nos enseñaron a tomar decisiones y repetir procesos. Las **funciones** nos ayudaron a organizar y reutilizar código, mientras que los **arrays** nos permitieron trabajar con grandes volúmenes de datos de forma estructurada.

Con los **punteros y referencias**, dimos un paso hacia el control de memoria y manipulación avanzada de variables. Los operadores `&` y `*` nos abrieron la puerta a comprender cómo se gestiona la información en los niveles más profundos de la máquina.

Más allá, descubrimos el poder de las **listas enlazadas**, **listas dobles** y **listas circulares**, estructuras dinámicas que se adaptan, crecen y se transforman durante la ejecución. Las **colas** y **pilas** nos mostraron cómo modelar el comportamiento de procesos en la vida real, como la atención por turnos o el historial de acciones. Finalmente, la **recursión** nos permitió resolver problemas complejos de forma elegante y matemática, descomponiéndolos en partes más pequeñas.

Puntos Clave

-  Estos temas no solo enseñan a programar, sino a **pensar algorítmicamente**.
 -  Cada estructura y técnica resuelve un tipo de problema específico.
 -  Conocer estas herramientas te convierte en un programador **eficiente, versátil y preparado** para desafíos reales.
-

Reflexión Final

*Aprender estructuras de datos no es solo aprender a codificar: es aprender a **pensar como un ingeniero del futuro**. Es diseñar soluciones, anticiparse a los problemas y construir programas capaces de crecer, adaptarse y resolver situaciones reales.*

Dominar estos conceptos es dominar el lenguaje de la lógica, la eficiencia y la creatividad computacional. Y esto es solo el comienzo.
