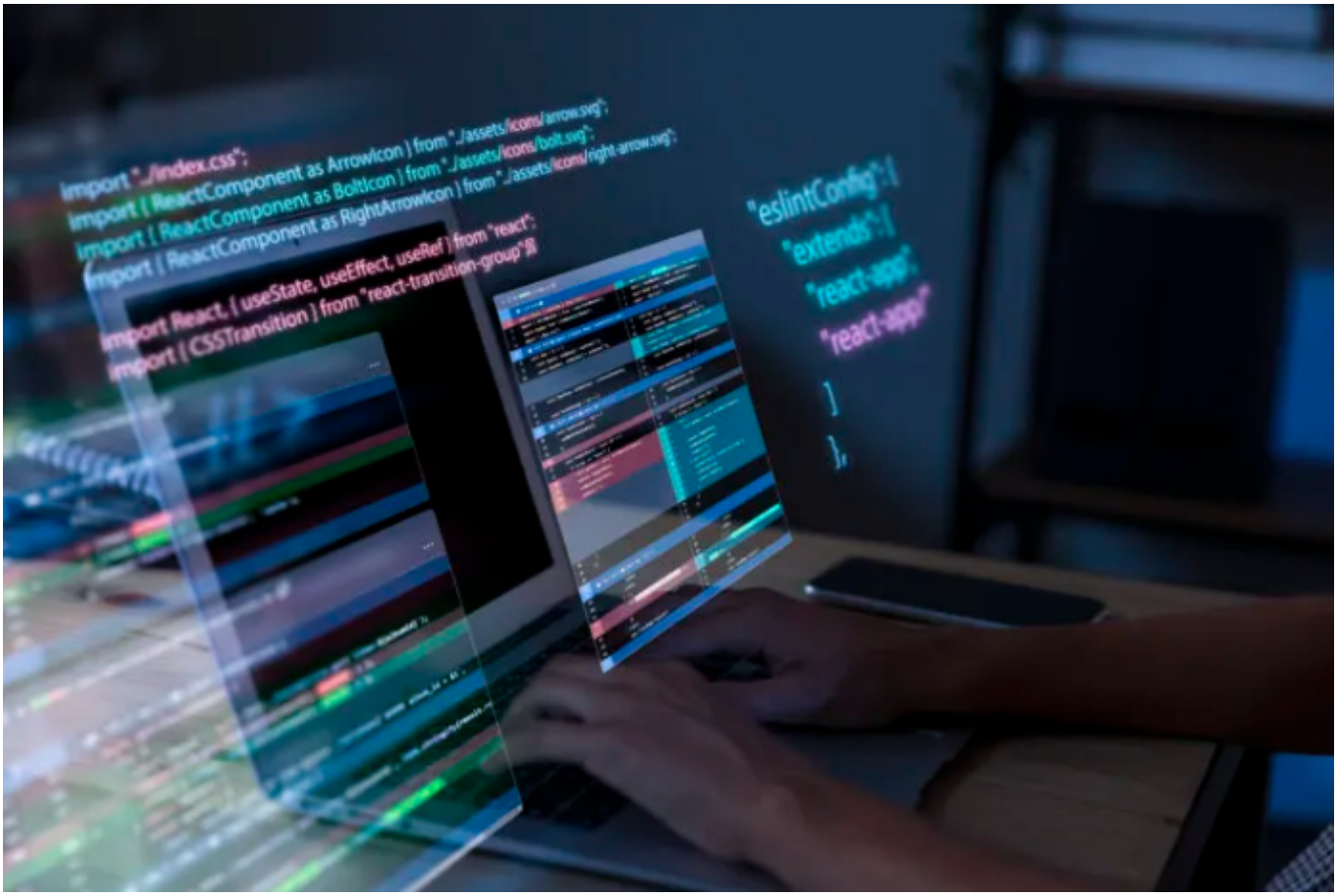
UNIVERSIDAD NACIONAL DEL ALTIPLANO - PUNO

FACULTAD DE INGENIERÍA ESTADÍSTICA E INFORMÁTICA

CÓDIGOS DE PROGRAMACIÓN - ESTRUCTURA DE DATOS



LIBRO DE PROGRAMACIÓN

Alumno: Quisbert Quispe Viviana Mary

Curso: Estructura de Datos

Docente: Fred Torres Cruz

Fecha: 29 de mayo de 2025

"Aprende a programar, crea el mundo que imaginas."

Índice

[Introducción – ¿Qué es programar ?

Programar en C++ es el proceso de escribir instrucciones en el lenguaje de programación C++ para indicarle a una computadora exactamente qué debe hacer y en qué orden. Estas instrucciones deben ser claras, precisas y lógicas, ya que la computadora no puede adivinar ni interpretar como lo haría una persona.

Programar es como enseñarle a una máquina cómo realizar una tarea, paso a paso, tal como lo harías con un robot.

🤖 Ejemplo cotidiano: el robot que hace té Imagina que tienes un robot, y quieres que prepare una taza de té. Para que lo haga correctamente, debes darle las instrucciones en el orden correcto. Sería algo así:

Calentar agua

Colocar una bolsa de té en la taza

Verter el agua caliente en la taza

Esperar 3 minutos

Sacar la bolsa

Servir

Estas instrucciones deben ser exactas. Si no le dices al robot que caliente el agua antes de verterla, por ejemplo, el té no saldrá bien. Lo mismo ocurre cuando se programa: cada paso importa y debe estar bien pensado.

📦 PARTE 1: Bases de la programación

🚀 Primeros pasos para tu primer programa

Crear tu primer programa en C++ es más fácil de lo que parece. Aquí te explico paso a paso cómo hacerlo desde cero.

🧰 1. Prepara tus herramientas

Antes de programar, necesitas tener lo siguiente:

- Un editor de texto o entorno de desarrollo (IDE). Puedes usar:
    - **Code::Blocks**
    - **Dev-C++**
    - **Visual Studio Code**
    - **Replit (en línea)**
- Un compilador de C++, como **g++** (viene con Code::Blocks y otras IDEs).

✍️ 2. Escribe tu primer programa

Abre tu editor y escribe el siguiente código:

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

int main() {
    cout << "Hola, mundo" << endl;
    return 0;
}
```

Línea 1: `#include <iostream>`

Esta línea le dice al programa que queremos usar la librería **iostream**, que contiene las herramientas necesarias para mostrar texto en pantalla (como `cout`) y leer datos del teclado.

Línea 2: `using namespace std;`

Aquí indicamos que vamos a usar el **espacio de nombres estándar** (std). Esto nos permite escribir simplemente `cout` en lugar de `std::cout`.

Línea 3: `int main() {`

Esta es la función principal llamada `main`. Es el punto de inicio de cualquier programa en C++. Cuando ejecutas tu programa, lo primero que hace la computadora es entrar a esta función y ejecutar las instrucciones que están dentro de ella.

Línea 4: `cout << "Hola, mundo" << endl;`

- `cout` es la instrucción que envía información a la pantalla (consola).
- `<<` es el operador que direcciona lo que queremos mostrar hacia `cout`.
- `"Hola, mundo"` es el texto que queremos que aparezca.
- `endl` significa **end line** (fin de línea), que hace un salto de línea para que cualquier cosa que se imprima después salga en una línea nueva.
- La línea termina con `;` que indica el fin de la instrucción.

Esta línea hace que en la pantalla aparezca:

```
Hola, mundo
```

Línea 5: `return 0;`

Con esta instrucción le decimos al sistema que el programa terminó correctamente. El número `0` es un código que representa éxito.

Línea 6: `}`

Esta llave cierra la función `main`, indicando que no hay más instrucciones para ejecutar.

📓 Capítulo 2 – Variables, tipos de datos y entrada/salida

En este capítulo aprenderás tres cosas muy importantes para todo lenguaje de programación:

1. Qué son las **variables**
2. Qué tipos de datos existen
3. Cómo interactuar con el usuario usando **entrada (`cin`)** y **salida (`cout`)**.

🔐 ¿Qué es una variable?

Una **variable** es un espacio en la memoria de la computadora donde se guarda un dato que puede cambiar durante la ejecución del programa.

Es como una cajita con nombre, donde puedes guardar valores.

✳️ Ejemplo:

```
int edad = 18;
```

Aquí:

int indica que es un número entero.

edad es el nombre de la variable.

18 es el valor que estamos guardando en esa variable.

🧮 Tipos de datos en C++ Cada variable debe tener un tipo de dato, que define qué tipo de información va a guardar. Aquí tienes los más usados:

Tipo de dato	Significado	Ejemplo de uso
`int`	Número entero	`int edad = 20;`
`float`	Número decimal corto	`float peso = 48.5;`
`double`	Número decimal largo (más preciso)	`double pi = 3.1416;`
`char`	Un solo carácter	`char letra = 'A';`
`string`	Cadena de texto (palabras o frases)	`string nombre = "Jhonatan";`
`bool`	Valor lógico: verdadero o falso	`bool activo = true;`

🔧 Nota: Para usar string, debes incluir la librería #include

🔄 Entrada y salida de datos

Para que un programa sea útil, debe poder **recibir datos** (entrada) y **mostrar resultados** (salida).

Entrada: `cin`

Usamos `cin` para pedir al usuario que escriba datos en la consola.

```
int edad;
cout << "Escribe tu edad: ";
cin >> edad;
```

Salida: `cout`

Usamos `cout` para mostrar mensajes o resultados en la pantalla.

```
cout << "Tu edad es: " << edad << endl;
```

🧩 Ejemplo completo

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

int main() {
    string nombre;
    int edad;

    cout << "Escribe tu nombre: ";
    getline(cin, nombre);  // Leer texto con espacios

    cout << "Escribe tu edad: ";
    cin >> edad;

    cout << "Hola " << nombre << ", tienes " << edad << " años." << endl;

    return 0;
}
```

🧮 Capítulo 3 – Operadores y expresiones

En este capítulo aprenderás a usar **operadores** en C++, que son los símbolos que permiten realizar operaciones matemáticas, comparaciones y manipulaciones entre datos.

📌 ¿Qué es un operador?

Un **operador** es un símbolo que le dice a la computadora qué operación debe realizar con uno o más valores (también llamados operandos).

Por ejemplo:

```
int suma = 5 + 3;
```

🔢 Tipos de operadores en C++

1. ⌨️ Operadores aritméticos Se usan para operaciones matemáticas básicas:

Operador	Significado	Ejemplo	Resultado
+	Suma	5 + 2	7
-	Resta	5 - 2	3
*	Multiplicación	5 * 2	10
/	División	10 / 2	5
%	Módulo (residuo)	10 % 3	1

1. 🔍 Operadores de comparación Se usan para comparar dos valores. El resultado siempre es true o false (booleano).

Operador	Significado	Ejemplo	Resultado
==	Igual a	5 == 5	true
!=	Distinto de	5 != 3	true
>	Mayor que	5 > 3	true
<	Menor que	3 < 5	true
>=	Mayor o igual que	5 >= 5	true
<=	Menor o igual que	3 <= 5	true

1. 🖇️ Operadores lógicos Se usan para combinar condiciones lógicas:

Operador	Significado	Ejemplo	Resultado
`&&`	Y lógico (AND)	`true && false`	`false`
`||`	O lógico (OR)	`true || false`	`true`
`!`	Negación (NOT)	`!true`	`false`

4. 🧰 Operadores de asignación Se usan para asignar valores a variables.

Operador	Ejemplo	Equivalente a...

Operador	Ejemplo	Equivalente a...
`=`	`x = 10;`	Asignar 10 a `x`
`+=`	`x += 5;`	`x = x + 5`
`-=`	`x -= 2;`	`x = x - 2`
`*=`	`x *= 3;`	`x = x * 3`
`/=`	`x /= 2;`	`x = x / 2`
`%=`	`x %= 4;`	`x = x % 4`

Capítulo 4 – Estructuras de control: if, if-else, switch

¿Qué son las estructuras de control?

Cuando programas, a veces quieres que el computador tome decisiones, como tú lo harías. Por ejemplo: "Si hace frío, me pongo abrigo; si no, me pongo una camiseta." En programación, esas decisiones las hacemos con **estructuras de control**.

En C++, las más usadas para decidir qué hacer son: `if`, `if-else` y `switch`.

---

1. La estructura `if`

¿Para qué sirve?

Para decirle al programa: "Haz algo solo si se cumple una condición."

¿Cómo funciona?

El programa revisa una condición (como si una pregunta fuera verdadera o falsa). Si es verdadera, hace lo que le dices. Si no, no hace nada.

Ejemplo sencillo

```
int edad = 18;

if (edad >= 18) {
    std::cout << "Eres mayor de edad." << std::endl;
}
```

¿Qué pasa aquí? Si la edad es 18 o más, el programa mostrará el mensaje "Eres mayor de edad." Si no, simplemente no pasa nada y sigue.

---

2. La estructura `if-else`

¿Para qué sirve?

Para darle al programa dos caminos: uno si la condición es cierta, y otro si es falsa.

¿Cómo funciona?

Si la condición es verdadera, hace una cosa. Si es falsa, hace otra.

Ejemplo fácil

```
int edad = 16;

if (edad >= 18) {
    std::cout << "Eres mayor de edad." << std::endl;
} else {
    std::cout << "Eres menor de edad." << std::endl;
}
```

¿Qué hace el programa? Si tienes 18 años o más, muestra que eres mayor. Si no, te dice que eres menor. ¡Así de simple!

---

3. La estructura `switch`

¿Para qué sirve?

Cuando tienes varias opciones y quieres que el programa elija una según el valor de una variable.

¿Cómo funciona?

El programa mira el valor y compara con cada opción (llamadas `case`). Cuando encuentra la opción correcta, hace lo que le dices y se detiene.

Si ninguna opción coincide, hace algo por defecto.

Ejemplo práctico

```
char opcion = 'B';

switch (opcion) {
    case 'A':
        std::cout << "Elegiste la opción A." << std::endl;
        break;
    case 'B':
        std::cout << "Elegiste la opción B." << std::endl;
        break;
    case 'C':
        std::cout << "Elegiste la opción C." << std::endl;
        break;
    default:
```

```
        std::cout << "Opción no válida." << std::endl;
}
```

¿Qué pasa aquí? Si la opción es 'A', 'B' o 'C', muestra el mensaje correspondiente. Si no es ninguna de esas, muestra que la opción no es válida.

🔁 PARTE 2: Todos los bucles

Capítulo 5 – Bucle while: repite mientras algo sea cierto

¿Qué es un bucle `while`?

Un **bucle** `while` es una estructura de control que le dice a la computadora: **"Repite un grupo de instrucciones mientras una condición sea verdadera."**

Es como decirle a alguien: *Mientras la luz esté verde, sigue caminando.* Cuando la luz cambie a rojo, deja de caminar.

¿Cómo funciona?

1. Primero, el programa verifica la condición que le diste.
2. Si la condición es verdadera (`true`), ejecuta el código dentro del bucle.
3. Después de ejecutar ese código, vuelve a revisar la condición.
4. Mientras la condición siga siendo verdadera, seguirá repitiendo el código.
5. Cuando la condición se vuelve falsa (`false`), el programa sale del bucle y continúa con lo que sigue.

Ejemplo simple

```
int contador = 1;

while (contador <= 5) {
    std::cout << "Número: " << contador << std::endl;
    contador++;  // Suma 1 al contador en cada vuelta
}
```

¿Qué hace este código?

- Empieza con `contador` igual a 1.
- Mientras `contador` sea menor o igual a 5, imprime el número.
- Después suma 1 al contador.
- Cuando el contador llegue a 6, la condición ya no se cumple y el bucle termina.

Salida en pantalla:

```
```

```
Número: 1
Número: 2
Número: 3
Número: 4
Número: 5
```

Capítulo 6 – Bucle do-while: haz al menos una vez

¿Qué es un bucle `do-while`?

El **bucle `do-while`** es muy parecido al `while`, pero con una diferencia importante: **El código dentro del bucle se ejecuta al menos una vez, antes de verificar la condición.**

Es como cuando pruebas un plato nuevo y solo después decides si te gusta o no. Primero lo pruebas (ejecutas el código), luego decides si seguir o parar (evalúas la condición).

¿Cómo funciona?

1. El programa primero ejecuta el código que está dentro del bloque `do`.
2. Después, verifica la condición que está en el `while`.
3. Si la condición es verdadera (`true`), vuelve a ejecutar el bloque `do`.
4. Si la condición es falsa (`false`), el programa sale del bucle y sigue con lo que sigue.

Ejemplo sencillo

```
int contador = 1;

do {
    std::cout << "Número: " << contador << std::endl;
    contador++;  // Suma 1 al contador en cada vuelta
} while (contador <= 5);
```

¿Qué hace este código?

- Primero imprime el número aunque sea la primera vez.
- Después revisa si `contador` es menor o igual a 5.
- Mientras la condición sea verdadera, sigue repitiendo el código.
- Cuando `contador` es 6, el bucle termina.

Salida en pantalla:

```
Número: 1
Número: 2
```

```
Número: 3
Número: 4
Número: 5
```

Capítulo 7 – Bucle for: repite con conteo claro

¿Qué es un bucle `for`?

El **bucle** `for` es otra forma de repetir un conjunto de instrucciones, pero es ideal cuando sabes cuántas veces quieres repetir algo.

Piensa en contar pasos: *"Voy a dar 10 pasos."* Aquí sabes desde el principio cuántas veces vas a repetir la acción.

¿Cómo funciona?

Un bucle `for` tiene tres partes muy claras:

1. **Inicio:** donde defines una variable que controla el conteo (por ejemplo, `i = 1`).
2. **Condición:** la regla que debe cumplirse para que el bucle siga (por ejemplo, `i <= 10`).
3. **Actualización:** cómo cambia la variable en cada repetición (por ejemplo, `i++`, que suma 1 cada vez).

La estructura básica es así:

```
for (inicialización; condición; actualización) {
    // Código que quieres repetir
}
```

Ejemplo simple

```
for (int i = 1; i <= 5; i++) {
    std::cout << "Número: " << i << std::endl;
}
```

¿Qué hace este código?

- Empieza con `i` igual a 1.
- Mientras `i` sea menor o igual a 5, imprime el número.
- Luego suma 1 a `i`.
- Cuando `i` llega a 6, el bucle termina.

Salida en pantalla:

```
```

```
Número: 1
Número: 2
Número: 3
Número: 4
Número: 5
```

El bucle `for` es muy usado porque organiza todo el conteo en una sola línea y es fácil de leer.

Capítulo 8 – Bucles anidados y control de bucles (break, continue)

¿Qué son los bucles anidados?

Los **bucles anidados** son bucles dentro de otros bucles. Es como cuando haces un recorrido por una cuadrícula: primero recorres las filas, y dentro de cada fila recorres las columnas.

¿Para qué sirven?

Sirven cuando necesitas hacer tareas repetitivas en dos (o más) niveles. Por ejemplo: imprimir una tabla, recorrer una matriz, o hacer combinaciones.

Ejemplo de bucles anidados

```cpp
for (int fila = 1; fila <= 3; fila++) {
    for (int columna = 1; columna <= 4; columna++) {
        std::cout << "Fila " << fila << ", Columna " << columna << std::endl;
    }
}
```

¿Qué hace este código?

- Primero, recorre las filas de 1 a 3.
- Por cada fila, recorre las columnas de 1 a 4.
- Imprime la posición fila-columna en cada paso.

Control de bucles con `break` y `continue`

A veces quieres controlar mejor cuándo salir o saltarte partes de un bucle.

1. `break`

El comando `break` detiene completamente el bucle, saliendo de él sin importar si la condición sigue siendo verdadera.

Ejemplo:

```
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;  // Sale del bucle cuando i es 5
    }
    std::cout << i << std::endl;
}
```

Salida:

```
1
2
3
4
```

Cuando llega a 5, el bucle termina y no imprime más números.

---

2. `continue`

El comando `continue` salta la vuelta actual del bucle y pasa directamente a la siguiente repetición.

Ejemplo:

```
for (int i = 1; i <= 5; i++) {
    if (i == 3) {
        continue;  // Salta el número 3
    }
    std::cout << i << std::endl;
}
```

Salida:

```
1
2
4
5
```

Cuando `i` es 3, el programa no imprime nada y sigue con el siguiente número.

Capítulo 9 – Arreglos (arrays): muchas variables en una sola

¿Qué es un arreglo (array)?

Un **arreglo** es una estructura que permite guardar varias variables del mismo tipo en un solo lugar bajo un mismo nombre. Es como una fila de cajas numeradas donde puedes guardar información similar. Cada caja tiene una posición llamada **índice**, que empieza desde 0.

¿Para qué sirven los arreglos?

Cuando tienes muchos datos relacionados, por ejemplo las notas de varios estudiantes o los precios de productos, en lugar de crear una variable para cada dato, usas un arreglo para tenerlos organizados y poder acceder a ellos fácilmente con un índice.

¿Cómo se declara un arreglo en C++?

Para crear un arreglo necesitas indicar:

- El tipo de dato que almacenará (por ejemplo, `int`, `char`, `float`).
- El nombre del arreglo.
- La cantidad de elementos que tendrá (su tamaño).

Ejemplo:

```
int numeros[5];  // Arreglo de 5 números enteros
```

¿Cómo se accede a los elementos?

Cada elemento tiene un índice empezando en 0. Para acceder o modificar un valor, usas el nombre del arreglo con el índice entre corchetes:

```
numeros[0] = 10;  // Asigna 10 al primer elemento
int x = numeros[2];  // Lee el tercer elemento
```

Ejemplo completo

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

int main() {
    int edades[4];  // Arreglo para 4 edades

    edades[0] = 18;
    edades[1] = 20;
    edades[2] = 22;
    edades[3] = 19;
```

```
    for (int i = 0; i < 4; i++) {
        cout << "Edad en la posición " << i << ": " << edades[i] << endl;
    }

    return 0;
}
```

Explicación:

- Se declara un arreglo llamado `edades` con 4 posiciones.
- Se asignan valores a cada posición del arreglo.
- Se usa un bucle `for` para recorrer el arreglo e imprimir cada edad junto a su posición.

Salida del programa:

```
Edad en la posición 0: 18
Edad en la posición 1: 20
Edad en la posición 2: 22
Edad en la posición 3: 19
```

Capítulo 10 – Cadenas de texto: arrays de caracteres

¿Qué es una cadena de texto?

Una **cadena de texto** es una secuencia de caracteres que forman palabras, frases o cualquier texto. En C++, las cadenas tradicionales se representan como **arrays de caracteres**, es decir, una lista de letras, números o símbolos, uno al lado del otro.

¿Cómo funciona una cadena de texto en C++?

- Cada carácter de la cadena es un elemento del arreglo (array) de tipo `char`.
- La cadena termina con un carácter especial llamado **carácter nulo** (`'\0'`), que indica el final del texto.
- Por ejemplo, la palabra `"Hola"` se almacena como un arreglo con los caracteres `'H'`, `'o'`, `'l'`, `'a'` y `'\0'` al final.

¿Cómo se declara una cadena de texto?

Se declara como un arreglo de `char` con un tamaño suficiente para guardar todos los caracteres más el carácter nulo.

Ejemplo:

```
char saludo[6] = {'H', 'o', 'l', 'a', '!', '\0'};
```

O también se puede declarar así, que es más común y práctico:

```
char saludo[] = "Hola!";
```

¿Cómo se usa una cadena de texto?

Puedes acceder a cada carácter usando su índice, igual que en los arreglos normales.

Ejemplo:

```
cout << saludo[0];  // Imprime 'H'
cout << saludo[4];  // Imprime '!'
```

Ejemplo completo

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

int main() {
    char nombre[] = "Viviana";

    cout << "Mi nombre es: " << nombre << endl;

    // Mostrar cada carácter uno por uno
    for (int i = 0; nombre[i] != '\0'; i++) {
        cout << "Caracter en posición " << i << ": " << nombre[i] << endl;
    }

    return 0;
}
```

Explicación:

- Se crea una cadena `nombre` con el texto `"Marisol"`.
- Se imprime la cadena completa.
- Luego, con un bucle `for` se recorren y muestran todos los caracteres hasta encontrar el carácter nulo (`'\0'`), que indica el final.

Salida del programa:

```
Mi nombre es: Marisol
Caracter en posición 0: M
Caracter en posición 1: a
Caracter en posición 2: r
Caracter en posición 3: i
Caracter en posición 4: s
```

```
Caracter en posición 5: o
Caracter en posición 6: l
```

Capítulo 11 – Estructuras (`struct`): tus propios tipos de datos

🧩 ¿Qué es una estructura (`struct`)?

Una **estructura** en C++ te permite crear **tu propio tipo de dato** combinando varios datos de distintos tipos bajo un mismo nombre. Es como diseñar un "molde" para representar algo más complejo, como una persona, un libro o un producto.

📦 ¿Para qué sirve?

Imagina que quieres guardar información de un estudiante: su nombre, edad y promedio. Podrías usar tres variables separadas, pero eso puede volverse desordenado.

Con `struct`, puedes agrupar todo en una sola unidad:

```
struct Estudiante {
    string nombre;
    int edad;
    float promedio;
};
```

🛠️ ¿Cómo se usa?

1. **Definir la estructura** (normalmente fuera del `main`):

```
struct Estudiante {
    string nombre;
    int edad;
    float promedio;
};
```

2. **Declarar variables del tipo estructurado**:

```
Estudiante alumno1;
```

3. **Asignar valores a los campos**:

```
alumno1.nombre = "manuela ";
alumno1.edad = 18;
alumno1.promedio = 16.5;
```

4. **Mostrar la información**:

```
cout << "Nombre: " << alumno1.nombre << endl;
cout << "Edad: " << alumno1.edad << endl;
cout << "Promedio: " << alumno1.promedio << endl;
```

🧠 Ejemplo completo

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

// Definimos una estructura llamada Estudiante
struct Estudiante {
    string nombre;
    int edad;
    float promedio;
};

int main() {
    Estudiante alumna;

    alumna.nombre = "manuela";
    alumna.edad = 18;
    alumna.promedio = 17.4;

    cout << "Datos de la estudiante:" << endl;
    cout << "Nombre: " << alumna.nombre << endl;
    cout << "Edad: " << alumna.edad << endl;
    cout << "Promedio: " << alumna.promedio << endl;

    return 0;
}
```

💻 Salida esperada:

```
Datos de la estudiante:
Nombre: manuela
Edad: 18
Promedio: 17.4
```

👥 También puedes usar arreglos de estructuras

Si quieres guardar varios estudiantes:

```
Estudiante clase[3];

clase[0].nombre = "Ana";
clase[1].nombre = "Luis";
clase[2].nombre = "Marcos";
```

📎 PARTE 4: Estructuras de datos dinámicas

Capítulo 12 – Punteros: direcciones y memoria

📌 ¿Qué es un puntero?

Un **puntero** en C++ es una variable especial que guarda la **dirección de memoria** de otra variable. Es como si en vez de guardar un número directamente, guardara un "mapa" que te dice dónde encontrar ese número.

🧠 ¿Por qué son importantes?

Los punteros permiten:

- Compartir datos entre funciones sin copiarlos.
- Acceder y modificar datos directamente en memoria.
- Crear estructuras dinámicas como listas o árboles.
- Usar arreglos de forma más eficiente.

📦 ¿Cómo se declara y se usa un puntero?

🧪 Ejemplo con explicación paso a paso

```
#include <iostream> // Importa la librería para usar cout
using namespace std;

int main() {
    int edad = 18; // Creamos una variable llamada 'edad' con valor 18

    int* puntero = &edad;
    // Creamos un puntero a entero llamado 'puntero'
    // Lo igualamos a la dirección de memoria de 'edad' usando '&'

    cout << "Valor de edad: " << edad << endl;
    // Imprime el valor de la variable normalmente

    cout << "Dirección de memoria de edad: " << &edad << endl;
    // Imprime la dirección de memoria de la variable
```

```
    cout << "Valor almacenado en puntero: " << puntero << endl;
    // Imprime lo que contiene el puntero, o sea, la dirección de memoria

    cout << "Valor al que apunta el puntero: " << *puntero << endl;
    // Imprime el valor que está en la dirección que guarda el puntero (18)

    return 0;
}
```

🔁 ¿Qué hacen los símbolos & y *?

Símbolo	Nombre	¿Qué hace?
&	Operador de dirección	Obtiene la dirección de memoria de una variable (&edad)
*	Operador de desreferencia	Accede al valor almacenado en una dirección de memoria (*p)

🛠️ ¿Dónde se usan los punteros?

✔ En funciones (para modificar valores desde fuera):

```
void duplicar(int* numero) {
    *numero = *numero * 2;
}
```

- Se pasa la dirección de la variable.
- Se usa * para modificar el valor original.

✔ En arreglos:

```
int numeros[] = {10, 20, 30};
int* p = numeros;

cout << *p << endl;     // Muestra el primer número: 10
cout << *(p + 1) << endl; // Muestra el segundo número: 20
```

✔ Para reservar memoria dinámica:

```
int* edad = new int; // Reservamos espacio para un entero en memoria
*edad = 25;          // Asignamos el valor 25 a esa dirección
delete edad;         // Liberamos la memoria
```

🔍 Cuadro

Acción	Código	Explicación
Declarar puntero	`int* p;`	Un puntero que guarda dirección de int
Apuntar a una variable	`p = &x;`	Guarda la dirección de `x`
Leer valor apuntado	`*p`	Accede al valor de la dirección guardada
Reservar memoria dinámica	`p = new int;`	Crea un nuevo espacio de memoria
Liberar memoria	`delete p;`	Libera el espacio reservado

Capítulo 13 – Listas enlazadas

🧠 ¿Qué es una lista enlazada ?

Una **lista enlazada simple** es una estructura de datos dinámica en la que cada **elemento (nodo)** contiene:

1. Un **dato**.
2. Un **puntero** que apunta al siguiente nodo de la lista.

💡 A diferencia de los arreglos, no se necesita saber cuántos elementos tendrá la lista al inicio. Se puede **agregar o quitar nodos fácilmente** durante la ejecución.

📦 ¿Cómo está compuesta?

Cada nodo tiene esta forma:

```
struct Nodo {
    int dato;          // Información que guarda
    Nodo* siguiente;   // Apunta al siguiente nodo
};
```

> 🔁 El último nodo siempre apunta a `NULL`, lo que indica el fin de la lista.

✏️ Crear y recorrer una lista enlazada simple (ejemplo completo)

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

// Estructura de un nodo
struct Nodo {
    int dato;
    Nodo* siguiente;
};

int main() {
    // Creamos tres nodos
    Nodo* primero = new Nodo();
```

```
    Nodo* segundo = new Nodo();
    Nodo* tercero = new Nodo();

    // Asignamos datos
    primero->dato = 10;
    segundo->dato = 20;
    tercero->dato = 30;

    // Enlazamos los nodos
    primero->siguiente = segundo;
    segundo->siguiente = tercero;
    tercero->siguiente = NULL; // Fin de la lista

    // Recorremos la lista
    Nodo* actual = primero;
    while (actual != NULL) {
        cout << actual->dato << " -> ";
        actual = actual->siguiente;
    }
    cout << "NULL" << endl;

    return 0;
}
```

🔎 ¿Qué hace este programa?

1. Crea tres nodos dinámicamente.
2. Guarda los valores `10`, `20` y `30`.
3. Enlaza los nodos uno tras otro.
4. Recorre la lista y muestra: `10 -> 20 -> 30 -> NULL`

🛠️ Operaciones básicas con listas enlazadas

Operación	¿Qué hace?
Insertar al inicio	Agrega un nuevo nodo al principio
Insertar al final	Agrega un nuevo nodo al final
Eliminar un nodo	Borra un nodo específico
Recorrer la lista	Muestra todos los datos
Buscar un elemento	Verifica si un dato está en la lista

🧪 Ejemplo: Insertar al inicio

```
Nodo* nuevo = new Nodo();
nuevo->dato = 5;
nuevo->siguiente = primero;
primero = nuevo;
```

> 🔄 Ahora el nodo con valor `5` es el primero de la lista.

📊 Visualización

```
[dato | siguiente] → [dato | siguiente] → [dato | NULL]
   10      ⟶           20      ⟶           30
```

Capítulo 14 – Listas doblemente enlazadas

🧠 ¿Qué es una lista doblemente enlazada?

Una **lista doblemente enlazada** es una estructura de datos dinámica donde **cada nodo tiene tres partes**:

1. Un **dato**.
2. Un **puntero al nodo anterior**.
3. Un **puntero al nodo siguiente**.

🔁 Esto permite **recorrer la lista en ambas direcciones** (hacia adelante y hacia atrás), a diferencia de la lista simple que solo va en un sentido.

📦 Estructura de un nodo

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

> El primer nodo tiene `anterior = NULL` El último nodo tiene `siguiente = NULL`

🖊️ Ejemplo: Crear y recorrer una lista doblemente enlazada

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

// Definición del nodo
struct Nodo {
    int dato;
    Nodo* anterior;
    Nodo* siguiente;
};

int main() {
    // Crear tres nodos
```

```
    Nodo* primero = new Nodo();
    Nodo* segundo = new Nodo();
    Nodo* tercero = new Nodo();

    // Asignar valores
    primero->dato = 10;
    segundo->dato = 20;
    tercero->dato = 30;

    // Enlazar hacia adelante
    primero->siguiente = segundo;
    segundo->siguiente = tercero;
    tercero->siguiente = NULL;

    // Enlazar hacia atrás
    primero->anterior = NULL;
    segundo->anterior = primero;
    tercero->anterior = segundo;

    // Recorrer hacia adelante
    cout << "Recorrido hacia adelante: ";
    Nodo* actual = primero;
    while (actual != NULL) {
        cout << actual->dato << " <-> ";
        actual = actual->siguiente;
    }
    cout << "NULL" << endl;

    // Recorrer hacia atrás
    cout << "Recorrido hacia atrás: ";
    actual = tercero;
    while (actual != NULL) {
        cout << actual->dato << " <-> ";
        actual = actual->anterior;
    }
    cout << "NULL" << endl;

    return 0;
}
```

🔎 ¿Qué hace este código?

1. Crea tres nodos dinámicamente con los datos `10`, `20`, `30`.
2. Los enlaza hacia adelante (`siguiente`) y hacia atrás (`anterior`).
3. Muestra los valores desde el primero al último y luego en orden inverso.

📊 Visualización

```
NULL ← [10] ⇄ [20] ⇄ [30] → NULL
```

⚙️ ¿Dónde se usan las listas doblemente enlazadas?

Se usan cuando necesitas:

- Recorrer datos en ambos sentidos.
- Insertar o eliminar elementos tanto al inicio como al final.
- Implementar **navegadores (botón atrás/adelante)**, **listas de reproducción**, **editores de texto**, etc.
-

Capítulo 15 – Listas circulares

🔁 ¿Qué es una lista circular?

Una **lista circular** es una estructura de datos enlazada donde el **último nodo apunta al primero**, formando un ciclo cerrado.

📌 Puede ser:

- **Simplemente circular**: cada nodo tiene un puntero al siguiente, y el último apunta al primero.
- **Doblemente circular**: cada nodo tiene punteros al anterior y al siguiente, y están conectados en ambos sentidos.

🧠 ¿Para qué sirve una lista circular?

Las listas circulares se usan cuando:

- Necesitas recorrer una lista sin volver a empezar manualmente.
- Quieres un **acceso continuo o cíclico** (por ejemplo, turnos de jugadores, menús circulares, etc.).

📦 Estructura de un nodo (simplemente enlazada)

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

✏️ Ejemplo: Crear una lista circular con 3 nodos

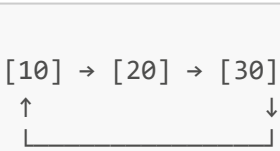
```
#include <iostream>
using namespace std;

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

int main() {
```

```
    Nodo* primero = new Nodo();
    Nodo* segundo = new Nodo();
    Nodo* tercero = new Nodo();

    primero->dato = 10;
    segundo->dato = 20;
    tercero->dato = 30;

    // Enlazar los nodos
    primero->siguiente = segundo;
    segundo->siguiente = tercero;
    tercero->siguiente = primero;  // Cierra el ciclo

    // Recorrer la lista circular (una vuelta)
    Nodo* actual = primero;
    cout << "Recorrido circular: ";
    do {
        cout << actual->dato << " -> ";
        actual = actual->siguiente;
    } while (actual != primero);
    cout << "(vuelve al inicio)" << endl;

    return 0;
}
```

🔍 ¿Qué hace este código?

- Crea tres nodos.
- Los enlaza formando un **ciclo**.
- Usa un bucle `do-while` para recorrerlos **una sola vuelta completa**, porque siempre se regresa al inicio.

🎯 Visualización



📚 PARTE 5: Pilas, colas y recursión

Capítulo 16 – Pilas (Stacks): Último en entrar, primero en salir

🧠 ¿Qué es un *stack*?

Un **stack** (en español, *pila*) es una estructura de datos que funciona como una **torre de objetos**: lo último que colocas es lo primero que se retira.

A esto se le llama principio **LIFO** (*Last In, First Out*), es decir, **el último en entrar es el primero en salir**.

Ejemplo cotidiano:

Imagina una pila de platos:

1. Pones un plato sobre otro → `push()`
2. Para sacar un plato, solo puedes sacar el de arriba → `pop()`

No puedes sacar el de abajo sin antes quitar los de arriba.

¿Para qué sirve un stack?

Los *stacks* son útiles en muchas situaciones, por ejemplo:

- Para **deshacer acciones** (como cuando haces CTRL+Z en Word).
- Para **navegar entre páginas web** (botón "Atrás").
- En **resolución de operaciones matemáticas** o **recursividad**.
- Para **gestionar llamadas de funciones** en un programa.

Operaciones básicas de un stack

Operación	¿Qué hace?
`push()`	Agrega un nuevo elemento a la pila
`pop()`	Elimina el elemento que está arriba
`top()`	Muestra el elemento en la cima, sin eliminarlo
`empty()`	Verifica si la pila está vacía

Ejemplo en C++

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

int main() {
    stack<int> pila;

    pila.push(10);  // Agregamos 10
    pila.push(20);  // Ahora 20 está arriba
    pila.push(30);  // 30 es el tope

    cout << "Elemento en la cima: " << pila.top() << endl;  // Muestra 30

    pila.pop();  // Quitamos el 30

    cout << "Nuevo tope: " << pila.top() << endl;  // Muestra 20
```

```
    return 0;
}
```

🔍 ¿Cómo se ve internamente?

Antes del `pop()`:

```
[30] <- top
[20]
[10]
```

Después del `pop()`:

```
[20] <- top
[10]
```

📓 Capítulo 17 – Colas (Queues): Primero en entrar, primero en salir

🧠 ¿Qué es una cola?

Una **cola (queue)** es una estructura de datos lineal que funciona igual que una **fila de personas esperando en un lugar**.

📌 **Regla principal**:

> El primero que llega es el primero que se va. Esto se llama **FIFO** (*First In, First Out*).

🏫 Ejemplo real

Imagina que estás en una tienda:

1. Llega Ana
2. Luego llega Luis
3. Después María

Entonces, el orden para ser atendidos es:

1. Ana
2. Luis
3. María

Así funciona una cola en programación. No puedes atender a María antes que a Ana, porque **María está al final**.

📦 ¿Para qué se usa una cola en programación?

Las colas se usan cuando los datos deben **mantener su orden de entrada**. Algunos ejemplos:

🔷 **Impresoras**: si mandas varios documentos a imprimir, la impresora los toma uno por uno en el orden en que llegaron.

🔷 **Mensajes**: en aplicaciones como WhatsApp, los mensajes se reciben y procesan en orden.

🔷 **Inteligencia artificial**: para controlar turnos o movimientos en juegos.

🔷 **Sistemas operativos**: para manejar procesos o tareas pendientes.

🛠️ Operaciones básicas

En C++ usamos la librería `#include <queue>` para trabajar con colas. Estas son las funciones más comunes:

Función	¿Qué hace?
`push(x)`	Agrega el elemento `x` al final
`pop()`	Elimina el elemento del frente
`front()`	Muestra el elemento al frente (sin eliminarlo)
`empty()`	Devuelve `true` si la cola está vacía
`size()`	Devuelve cuántos elementos hay

💻 Código explicado paso a paso

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

int main() {
    queue<string> cola;

    cola.push("Ana");     // Ana entra
    cola.push("Luis");    // Luis entra
    cola.push("María");   // María entra

    cout << "Frente de la cola: " << cola.front() << endl;  // Muestra "Ana"

    cola.pop();  // Ana sale de la cola

    cout << "Nuevo frente: " << cola.front() << endl;       // Muestra "Luis"

    return 0;
}
```

¿Qué pasa internamente?

🔷 Después del `push()`:

```
Frente --> [Ana] [Luis] [María] <-- Final
```

🔷 Después del `pop()`:

```
Frente --> [Luis] [María] <-- Final
```

Solo puedes acceder y eliminar al elemento del **frente**, no al del medio o final directamente.

📓 Capítulo 18 – Colas circulares

🧠 ¿Qué es una cola circular?

Una **cola circular** es una estructura de datos muy parecida a una cola normal (como una fila en el banco), pero con una mejora: **cuando llegas al final, puedes volver al inicio**.

> ✨ Imagina que la cola está dibujada en un círculo, por eso se llama "circular".

🎯 ¿Por qué usar una cola circular?

Porque en una **cola normal** (lineal), cuando se llenan los espacios del arreglo y eliminas elementos del frente, **ya no puedes usar ese espacio libre al principio**, aunque esté vacío.

Con una **cola circular**, sí puedes volver a usar esos espacios vacíos. ¡Aprovechas todo el espacio disponible!

🔁 Ejemplo visual

Supón que tienes una cola de 5 espacios:

```
[ _ ][ _ ][ _ ][ _ ][ _ ]   ← Capacidad: 5
```

Agregas tres números:

```
[ 10 ][ 20 ][ 30 ][ _ ][ _ ]
```

Sacas dos elementos del inicio:

```
[ _ ][ _ ][ 30 ][ _ ][ _ ]
         ↑
       frente
```

En una cola normal, ya no puedes usar los espacios vacíos del inicio. ❌ ¡Desperdicio de memoria!

Pero en una **cola circular**, puedes volver al inicio:

```
[ 40 ][ 50 ][ 30 ][ _ ][ _ ]
```

✅ ¡Se aprovecha todo el arreglo!

💡 ¿Dónde se usan las colas circulares?

- En **buffers** de audio o video (como en YouTube).
- En **impresoras**, que guardan trabajos en cola.
- En **sistemas operativos**, para procesar tareas en orden.
- En **microcontroladores**, donde hay poca memoria.

💻 Ejemplo simple en C++

```cpp
#include <iostream>
#define TAM 5

using namespace std;

class ColaCircular {
private:
    int cola[TAM];
    int frente, final, cantidad;

public:
    ColaCircular() {
        frente = 0;
        final = -1;
        cantidad = 0;
    }

    void encolar(int valor) {
        if (cantidad == TAM) {
            cout << "⚠ La cola está llena\n";
            return;
        }
        final = (final + 1) % TAM;
        cola[final] = valor;
        cantidad++;
    }

    void desencolar() {
        if (cantidad == 0) {
            cout << "⚠ La cola está vacía\n";
            return;
        }
        frente = (frente + 1) % TAM;
```

```
        cantidad--;
    }

    void mostrar() {
        cout << "📋 Elementos en la cola: ";
        for (int i = 0; i < cantidad; i++) {
            int indice = (frente + i) % TAM;
            cout << cola[indice] << " ";
        }
        cout << endl;
    }
};

int main() {
    ColaCircular cola;
    cola.encolar(10);
    cola.encolar(20);
    cola.encolar(30);
    cola.desencolar();   // Sale el 10
    cola.encolar(40);
    cola.encolar(50);
    cola.encolar(60);    // No se puede: cola llena
    cola.mostrar();      // Muestra los elementos actuales
    return 0;
}
```

📝 ¿Qué hace este programa?

1. Crea una cola circular de tamaño 5.
2. Agrega 3 elementos: `10`, `20`, `30`.
3. Quita uno (sale el `10`).
4. Agrega otros valores y reutiliza los espacios del inicio.
5. Muestra lo que hay en la cola.

📓 Capítulo 19 – Recursión: funciones que se llaman a sí mismas

🤯 ¿Qué es la recursión?

La recursión es una técnica en programación donde una función se llama a sí misma para resolver un problema paso a paso, resolviendo primero los casos más simples.

Es como cuando te miras en un espejo frente a otro: la imagen se repite una y otra vez, cada vez más pequeña. Así funciona una función recursiva: se repite dentro de sí misma hasta llegar a una solución.

📦 ¿Para qué sirve?

La recursión es útil cuando un problema se puede **dividir en partes más pequeñas** que son similares al problema original. Por ejemplo:

- Calcular el **factorial** de un número
- Generar la **serie Fibonacci**
- Buscar datos en estructuras como **árboles**
- Resolver problemas que siguen un **patrón repetitivo**

🧩 ¿Cómo funciona una función recursiva?

Una función recursiva **debe tener siempre dos partes**:

1. 🟤 **Caso base** → cuando se debe detener la repetición.
2. 🔁 **Llamada recursiva** → la función se llama a sí misma con un valor más pequeño o modificado.

> Si no hay un caso base, la función se ejecutará infinitamente y causará errores (¡mucho cuidado!).

🧪 Ejemplo 1: Factorial de un número

El **factorial** de un número `n` (n!) se calcula así:

```
5! = 5 × 4 × 3 × 2 × 1 = 120
```

🔧 Código en C++:

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

int factorial(int n) {
    if (n == 0)  // caso base
        return 1;
    else
        return n * factorial(n - 1);  // llamada recursiva
}

int main() {
    int numero = 5;
    cout << "El factorial de " << numero << " es: " << factorial(numero);
    return 0;
}
```

🧠 ¿Qué pasa aquí?

- `factorial(5)` necesita el resultado de `factorial(4)`
- `factorial(4)` necesita el resultado de `factorial(3)`
- ...
- Hasta llegar a `factorial(0)` que devuelve `1`, y desde ahí se resuelven todas las operaciones.

🧪 Ejemplo 2: Serie Fibonacci

La **serie Fibonacci** es una secuencia de números donde cada número es la suma de los dos anteriores:

```
0, 1, 1, 2, 3, 5, 8, 13, ...
```

🔧 Código en C++:

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

int fibonacci(int n) {
    if (n == 0) return 0;       // caso base
    if (n == 1) return 1;       // caso base
    return fibonacci(n - 1) + fibonacci(n - 2);  // llamada recursiva
}

int main() {
    int numero = 6;
    cout << "Fibonacci de " << numero << " es: " << fibonacci(numero);
    return 0;
}
```

🌳 PARTE 6: Árboles avanzados

🌳 Capítulo 20 – Árbol Binario Simple

🧠 ¿Qué es un árbol binario?

Un **árbol binario** es una estructura de datos jerárquica compuesta por nodos, en la cual **cada nodo tiene como máximo dos hijos**: uno izquierdo y uno derecho.

Se representa como un grafo **no cíclico y dirigido** que:

- parte desde un nodo raíz,
- ramifica en subárboles izquierdo y derecho,
- y termina en nodos hoja (sin hijos).

📐 Definición formal

Un árbol binario es:

- vacío (nulo), o
- un nodo que tiene un valor, un subárbol izquierdo y un subárbol derecho.

Cada nodo contiene:

```
struct Nodo {
    int valor;
    Nodo* izq;
    Nodo* der;
};
```

🎯 Aplicaciones comunes

- Representación de expresiones matemáticas (árboles de expresión)
- Implementación de árboles de búsqueda binaria (Binary Search Trees – BST)
- Estructuras base para árboles AVL, B-trees, heaps, etc.
- Algoritmos de ordenamiento y búsqueda
- Compiladores y árboles sintácticos

🔍 Propiedades clave

- **Altura**: Longitud del camino más largo desde la raíz hasta una hoja.
- **Tamaño**: Número total de nodos.
- **Nivel**: Distancia desde la raíz (nivel 0).
- **Grado**: Máximo número de hijos (en binarios: 2).
- **Nodos hoja**: Nodos sin hijos.

📊 Ejemplo visual de inserción

Insertamos los elementos: `6, 3, 9, 1, 5, 10`

```
      6
     / \
    3   9
   / \   \
  1   5   10
```

💻 Implementación en C++

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

// Estructura del nodo
struct Nodo {
    int valor;
    Nodo* izq;
    Nodo* der;
};

// Función para crear un nuevo nodo
Nodo* nuevoNodo(int valor) {
```

```
    Nodo* nodo = new Nodo();
    nodo->valor = valor;
    nodo->izq = nodo->der = nullptr;
    return nodo;
}

// Inserta un valor en el árbol binario
Nodo* insertar(Nodo* raiz, int valor) {
    if (raiz == nullptr)
        return nuevoNodo(valor);

    if (valor < raiz->valor)
        raiz->izq = insertar(raiz->izq, valor);
    else
        raiz->der = insertar(raiz->der, valor);

    return raiz;
}

// Recorrido Inorden (izquierda - raíz - derecha)
void inorden(Nodo* raiz) {
    if (raiz != nullptr) {
        inorden(raiz->izq);
        cout << raiz->valor << " ";
        inorden(raiz->der);
    }
}

int main() {
    Nodo* arbol = nullptr;

    int datos[] = {6, 3, 9, 1, 5, 10};
    for (int valor : datos)
        arbol = insertar(arbol, valor);

    cout << "Recorrido Inorden: ";
    inorden(arbol);
    cout << endl;

    return 0;
}
```

🧪 Análisis

Complejidad temporal:

- **Búsqueda, inserción y eliminación:**
  - Mejor caso: `O(log n)` (árbol balanceado)
  - Peor caso: `O(n)` (árbol degenerado, tipo lista)

Ventajas:

- Inserción y búsqueda rápida (en promedio)
- Base para estructuras más complejas (AVL, Heap, etc.)

Desventajas:

- Puede desbalancearse fácilmente si no se controla

🌲 Capítulo 22 – Árboles Balanceados (AVL)

🧠 ¿Qué es un árbol AVL?

Un **árbol AVL** es una mejora del árbol binario de búsqueda (ABB), que se **autobalancea** cada vez que insertamos o eliminamos un elemento.

Fue creado por **Adelson-Velskii y Landis** en 1962, y sus iniciales dan nombre a esta estructura.

📐 ¿Qué significa balanceado?

Cada nodo del árbol guarda una propiedad llamada **altura**, y el árbol está balanceado cuando:

> 🔁 La diferencia entre la altura del subárbol izquierdo y el derecho es -1, 0 o +1.

A esa diferencia se le llama **factor de balance (FB)**.

- Si FB = -1, 0 o 1 → ✅ El nodo está balanceado
- Si FB < -1 o FB > 1 → ❌ El nodo está desequilibrado → se debe rotar

⚙️ Rotaciones para balancear

Cuando un nodo se desbalancea, se hace una rotación para corregirlo. Hay 4 casos posibles:

Caso	Tipo de rotación	Cuándo sucede
1️⃣	Rotación simple a la derecha (RR)	Cuando se inserta en la izquierda de la izquierda
2️⃣	Rotación simple a la izquierda (LL)	Cuando se inserta en la derecha de la derecha
3️⃣	Rotación doble izquierda-derecha (LR)	Cuando se inserta en la derecha de la izquierda
4️⃣	Rotación doble derecha-izquierda (RL)	Cuando se inserta en la izquierda de la derecha

🌳 Ejemplo visual

Insertamos los valores: `10, 5, 3`

```
    10
    /
   5
  /
 3
```

Este árbol está desbalanceado (FB = 2). Aplicamos una **rotación simple a la derecha**, y se convierte en:

```
   5
  / \
 3   10
```

Ahora el árbol está balanceado.

Código en C++ (versión simplificada)

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

struct Nodo {
    int valor;
    Nodo* izq;
    Nodo* der;
    int altura;
};

// Obtener altura de un nodo
int altura(Nodo* nodo) {
    return nodo ? nodo->altura : 0;
}

// Crear nodo nuevo
Nodo* crearNodo(int valor) {
    Nodo* nuevo = new Nodo();
    nuevo->valor = valor;
    nuevo->izq = nuevo->der = nullptr;
    nuevo->altura = 1;
    return nuevo;
}

// Calcular factor de balance
int balance(Nodo* nodo) {
    return nodo ? altura(nodo->izq) - altura(nodo->der) : 0;
}

// Rotación simple a la derecha
Nodo* rotarDerecha(Nodo* y) {
    Nodo* x = y->izq;
    y->izq = x->der;
    x->der = y;

    y->altura = 1 + max(altura(y->izq), altura(y->der));
    x->altura = 1 + max(altura(x->izq), altura(x->der));

    return x;
```

```
}

// Rotación simple a la izquierda
Nodo* rotarIzquierda(Nodo* x) {
    Nodo* y = x->der;
    x->der = y->izq;
    y->izq = x;

    x->altura = 1 + max(altura(x->izq), altura(x->der));
    y->altura = 1 + max(altura(y->izq), altura(y->der));

    return y;
}

// Insertar y balancear
Nodo* insertar(Nodo* nodo, int valor) {
    if (!nodo) return crearNodo(valor);

    if (valor < nodo->valor)
        nodo->izq = insertar(nodo->izq, valor);
    else if (valor > nodo->valor)
        nodo->der = insertar(nodo->der, valor);
    else
        return nodo; // No duplicados

    nodo->altura = 1 + max(altura(nodo->izq), altura(nodo->der));
    int fb = balance(nodo);

    if (fb > 1 && valor < nodo->izq->valor)
        return rotarDerecha(nodo);

    if (fb < -1 && valor > nodo->der->valor)
        return rotarIzquierda(nodo);

    if (fb > 1 && valor > nodo->izq->valor) {
        nodo->izq = rotarIzquierda(nodo->izq);
        return rotarDerecha(nodo);
    }

    if (fb < -1 && valor < nodo->der->valor) {
        nodo->der = rotarDerecha(nodo->der);
        return rotarIzquierda(nodo);
    }

    return nodo;
}

// Recorrido inorden
void inorden(Nodo* raiz) {
    if (raiz) {
        inorden(raiz->izq);
        cout << raiz->valor << " ";
        inorden(raiz->der);
    }
```

```
}

int main() {
    Nodo* arbol = nullptr;
    int valores[] = {10, 5, 3, 8, 15, 20};

    for (int v : valores)
        arbol = insertar(arbol, v);

    cout << "Árbol AVL (inorden): ";
    inorden(arbol);
    cout << endl;

    return 0;
}
```

✅ Ventajas y ❌ Desventajas

✅ **Ventajas**	❌ **Desventajas**
🔄 Se autobalancea automáticamente	⚙️ Código más complejo que un árbol binario simple
⏱️ Operaciones en O(log n) en todos los casos	💾 Mayor uso de memoria (cada nodo guarda altura)
📉 Evita el peor caso del ABB (lista degenerada)	🧠 Las rotaciones pueden ser difíciles de entender al inicio
🔧 Ideal para muchas inserciones y búsquedas	🛠️ No tan usado en bases de datos (se prefiere árbol B)

🗃️ Aplicaciones prácticas

- Sistemas en tiempo real (donde se necesita eficiencia constante)
- Juegos y motores gráficos (para actualizar estados)
- Compiladores y optimización de expresiones
- Indexación de datos dinámicos en estructuras medianas

🌴 Capítulo 23 – Árboles B y B+

🧠 ¿Por qué necesitamos otra estructura?

Imagina que estás trabajando con una **biblioteca enorme**, con millones de libros. Si buscas un título específico, un **árbol binario** o incluso un **AVL** puede ayudarte... **pero se quedan cortos** cuando todo debe guardarse y buscarse desde un **disco duro o base de datos**.

Ahí entran en juego los **Árboles B y B+**: estructuras creadas para trabajar con **grandes cantidades de información**, de manera **rápida y eficiente**. Son como bibliotecarios expertos que saben exactamente en qué estante, fila y cajón está lo que buscas.

🌳 ¿Qué es un árbol B?

Un **árbol B** es una especie de "súper árbol" que, a diferencia de los árboles binarios (que solo tienen dos ramas por nodo), puede tener muchas ramas. Piensa en él como un menú con varias opciones en cada página.

Por ejemplo:

- Un árbol binario te da solo **izquierda o derecha**.
- Un árbol B te da 3, 4 o hasta **10 caminos posibles** en un solo nodo.

¿Y por qué esto es útil?

Porque al tener más caminos por nodo, **el árbol crece más a lo ancho que a lo alto**, y eso significa que necesitas **menos pasos para llegar a lo que buscas**.

📐 ¿Qué reglas siguen los árboles B?

1. Todos los **datos están ordenados**.
2. Todos los **niveles de hojas están alineados** (no hay ramas más largas).
3. Cada nodo puede tener **varios valores** y **varios hijos**.
4. El número de claves e hijos depende del **orden** `m` del árbol:
    - Un nodo puede tener hasta `m - 1` claves y `m` hijos.

🧱 Ejemplo: árbol B de orden 3

Supón que insertas los números: `10`, `20`, `30`. Como solo caben 2 claves en un nodo, al insertar la tercera, **se divide**:

```
[10 20 30] → se divide en:

    [20]
   /    \
[10]   [30]
```

¡Y listo! El árbol se balancea solo, sin rotaciones complicadas como en AVL.

🌴 ¿Qué es un árbol B+?

Es como el primo mejorado del árbol B. La diferencia está en **dónde se almacenan los datos** y **cómo se enlazan las hojas**.

Árbol B	Árbol B+
Las claves pueden estar en cualquier nodo	Las claves **solo están en las hojas**
Las hojas no están enlazadas	Las hojas **están conectadas** (como una lista)

¿Y esto qué permite?

✅ Puedes hacer **recorridos ordenados muy rápidos** (perfecto para bases de datos). ✅ Buscar rangos como "todos los datos entre 100 y 500" es facilísimo.

🛠️ ¿Dónde se usan?

Mucho más de lo que crees:

- 🗂️ En sistemas de archivos (Windows, Linux, Android...)
- 💾 En bases de datos (MySQL, Oracle, PostgreSQL...)
- 📚 En índices de libros digitales
- 📊 En buscadores y motores de almacenamiento

Son la **columna vertebral del acceso rápido a datos** en casi todo lo que usas a diario.

💻 Código didáctico de árbol B (simulado)

Aquí te muestro un pequeño ejemplo de cómo **se vería** un árbol B al insertar claves:

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

const int ORDEN = 3;

struct NodoB {
    vector<int> claves;
    vector<NodoB*> hijos;
    bool esHoja;

    NodoB(bool hoja) {
        esHoja = hoja;
    }
};

void mostrar(NodoB* nodo, int nivel = 0) {
    if (!nodo) return;
    cout << string(nivel * 4, ' ') << "[ ";
    for (int clave : nodo->claves) cout << clave << " ";
    cout << "]\n";
    for (NodoB* hijo : nodo->hijos)
        mostrar(hijo, nivel + 1);
}

NodoB* ejemplo() {
    NodoB* raiz = new NodoB(false);
    raiz->claves = {20};
    NodoB* izq = new NodoB(true); izq->claves = {10};
    NodoB* der = new NodoB(true); der->claves = {30};
```

```
    raiz->hijos = {izq, der};
    return raiz;
}

int main() {
    NodoB* arbol = ejemplo();
    cout << "Árbol B de ejemplo:\n";
    mostrar(arbol);
    return 0;
}
```

⚠ Este código **no implementa inserción completa**, solo **una estructura ya armada** para ilustrar el concepto. Una implementación real es más extensa, pero puedes incluirla en apéndices avanzados.

Ventajas y Desventajas

✅ Ventajas	❌ Desventajas
Ideal para trabajar con disco o archivos grandes	Implementación compleja comparado con ABB
Búsqueda rápida, incluso en millones de datos	Usa más memoria por nodo
Óptimo para recorridos ordenados (B+)	Divisiones y fusiones complican el código
Eficiente para rangos o filtros de valores	Menos intuitivo para estudiantes nuevos

¿Y cuándo uso AVL o B/B+?

Necesitas...	Usa...
Estructura ligera en RAM	Árbol AVL
Manejar datos en disco o archivo	Árbol B
Buscar valores ordenados o hacer filtros por rango	Árbol B+

**

Capítulo 24 – Árbol Heap y algoritmo HeapSort

¿Qué es un Heap?

Un **Heap** (o **montículo**) es una estructura de árbol **completa** que se usa sobre todo para **ordenar elementos rápidamente** o **encontrar el valor más grande o más pequeño** de manera eficiente.

> 🎯 Se usa mucho en algoritmos de prioridad, colas y ordenamiento eficiente.

Existen dos tipos principales:

- **Max-Heap**: el valor más grande está arriba.
- **Min-Heap**: el valor más pequeño está arriba.

📦 ¿Qué lo hace diferente de otros árboles?

- Siempre es **un árbol binario completo** (todos los niveles llenos menos el último).
- Cada **padre cumple una regla** respecto a sus hijos:
    - En **Max-Heap**: el padre es mayor que sus hijos.
    - En **Min-Heap**: el padre es menor que sus hijos.

No importa si el árbol está balanceado como AVL. Aquí lo importante es el orden entre padres e hijos, y que se llene nivel por nivel.

🔨 Ejemplo visual de un Max-Heap

Insertamos los números: `40, 20, 35, 10, 15, 30`.

Representación en forma de árbol:

```
      40
     /  \
   20    35
  / \    /
10  15 30
```

Aquí el 40 está arriba porque es el más grande. En cada subárbol, el padre es mayor que sus hijos.

🔁 ¿Y qué es HeapSort?

Es un algoritmo de ordenamiento que **usa un heap** para ordenar los datos.

🧩 ¿Cómo funciona?

1. Se construye un **Max-Heap** con los datos.
2. Se intercambia el primer valor (el más grande) con el último.
3. Se reduce el tamaño del heap (como si lo “sacaras”) y se ajusta el árbol.
4. Se repite hasta que todos estén ordenados.

💻 Código en C++ explicado

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

// Intercambiar dos valores
void intercambiar(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}
```

```cpp
// Función para hacer heap (ajustar el árbol)
void heapify(int arr[], int n, int i) {
    int mayor = i;        // Suponemos que el más grande es la raíz
    int izquierda = 2 * i + 1;
    int derecha = 2 * i + 2;

    if (izquierda < n && arr[izquierda] > arr[mayor])
        mayor = izquierda;

    if (derecha < n && arr[derecha] > arr[mayor])
        mayor = derecha;

    if (mayor != i) {
        intercambiar(arr[i], arr[mayor]);
        heapify(arr, n, mayor); // Recursivamente ajustar
    }
}

// HeapSort
void heapSort(int arr[], int n) {
    // Paso 1: construir el heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Paso 2: uno por uno sacar elementos del heap
    for (int i = n - 1; i > 0; i--) {
        intercambiar(arr[0], arr[i]); // Mover raíz al final
        heapify(arr, i, 0);           // Reajustar el heap
    }
}

// Mostrar arreglo
void mostrar(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int datos[] = {40, 20, 35, 10, 15, 30};
    int n = sizeof(datos) / sizeof(datos[0]);

    cout << "Original: ";
    mostrar(datos, n);

    heapSort(datos, n);

    cout << "Ordenado con HeapSort: ";
    mostrar(datos, n);

    return 0;
}
```

✅ Ventajas y ❌ Desventajas del Heap

✅ **Ventajas**	❌ **Desventajas**
🔝 Acceder rápido al mayor o menor elemento	❌ No es tan rápido para búsquedas generales
⚖️ Siempre es un árbol binario completo	❌ Más difícil de implementar que burbuja o selección
🔁 Funciona bien en ordenamientos grandes	❌ No es estable (puede desordenar elementos iguales)
🧠 Se puede implementar con solo un arreglo	🌡️ No es tan intuitivo como un árbol BST o AVL

🎯 Aplicaciones reales

- 🔔 **Colas de prioridad** (por ejemplo, en una sala de emergencias: el paciente más grave va primero).
- 🕹️ **Sistemas de eventos** en juegos (eventos más urgentes tienen prioridad).
- 💻 **Ordenamiento de datos** cuando se necesita eficiencia sin estructuras complejas.
- 🌐 **Algoritmos como Dijkstra** (para encontrar caminos más cortos).

🧷 Capítulo 25 – Árbol Rojo-Negro

🧠 ¿Qué es un Árbol Rojo-Negro?

Un **Árbol Rojo-Negro (Red-Black Tree)** es un **árbol binario de búsqueda** especial que **se auto-balancea**, pero con reglas distintas a un AVL.

👉 ¿Por qué existe si ya hay AVL?

Porque aunque **AVL es más estricto** (siempre lo más balanceado posible), los árboles Rojo-Negro prefieren ser **más relajados pero más rápidos** en inserciones y eliminaciones. Es como alguien que no es tan perfeccionista, pero termina más rápido.

🎨 ¿Por qué se llama “Rojo-Negro”?

Cada nodo del árbol está **pintado de rojo 🔴 o negro ⚫**. Esto no es un adorno: los colores ayudan a mantener el **balance del árbol**.

📏 Reglas clave

1. Cada nodo es **rojo o negro**.
2. La **raíz siempre es negra**.
3. Las **hojas nulas** (punteros vacíos) se consideran **negras**.
4. Un **nodo rojo nunca puede tener un hijo rojo**. ❌🔴🔴
5. Desde cualquier nodo, todas las rutas hasta sus hojas tienen **la misma cantidad de nodos negros** (esto asegura el balance).

🌱 Ejemplo visual

Insertamos: `10, 20, 30`

Paso 1:

```
10⚫
  \
  20🔴
```

Paso 2: Insertar 30. Hay dos rojos seguidos (20🔴 y 30🔴).

¡Esto rompe la regla! Se arregla con **rotación y recoloreo**:

```
    20⚫
   /   \
10🔴  30🔴
```

Todo volvió a estar en orden ✅

🔧 ¿Por qué se usa?

Este tipo de árbol es ideal cuando necesitas mantener muchos datos **ordenados y balanceados automáticamente**, pero con un costo **moderado de procesamiento**.

Por eso lo usan en cosas como:

- 🗂️ Implementaciones de `map`, `set` en C++
- 🐘 Bases de datos como MongoDB
- 💾 Sistemas de archivos (ext3/ext4)
- 🧠 Árboles sintácticos en compiladores

💻 Código C++ básico – Inserción Rojo-Negro (estructura inicial)

El código completo es extenso, así que aquí va una **estructura didáctica simplificada** para comenzar a visualizarlo:

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

enum Color { ROJO, NEGRO };

struct Nodo {
    int valor;
    Color color;
    Nodo* izq;
    Nodo* der;
    Nodo* padre;

    Nodo(int val) {
        valor = val;
```

```
        color = ROJO; // Nuevo nodo siempre comienza rojo
        izq = der = padre = nullptr;
    }
};

// Mostrar color como texto
string colToStr(Color c) {
    return (c == ROJO) ? "🔴" : "⚫";
}

// Mostrar árbol (modo didáctico)
void mostrar(Nodo* nodo, int nivel = 0) {
    if (nodo) {
        mostrar(nodo->der, nivel + 1);
        cout << string(nivel * 5, ' ') << nodo->valor << colToStr(nodo->color) <<
endl;
        mostrar(nodo->izq, nivel + 1);
    }
}

// Ejemplo manual: árbol estático
Nodo* ejemploManual() {
    Nodo* raiz = new Nodo(20);
    raiz->color = NEGRO;
    raiz->izq = new Nodo(10);
    raiz->izq->padre = raiz;
    raiz->der = new Nodo(30);
    raiz->der->padre = raiz;
    return raiz;
}

int main() {
    Nodo* arbol = ejemploManual();
    cout << "Árbol Rojo-Negro (ejemplo manual):\n";
    mostrar(arbol);
    return 0;
}
```

📌 Este ejemplo solo **crea un árbol Rojo-Negro estático** con nodos pintados a mano. Para una implementación real con inserciones y rotaciones automáticas, se necesita más código (ideal para un capítulo de apéndice o nivel avanzado).

✅ Ventajas y ❌ Desventajas

✅ **Ventajas**	❌ **Desventajas**
⚖️ Se auto-balancea con reglas simples	🤯 Más difícil de entender al principio
⚡ Inserciones y eliminaciones más rápidas	📦 Requiere más espacio (por el color extra)
💻 Muy usado en la STL (`map`, `set`)	❌ Puede parecer menos intuitivo que AVL
🔐 Útil para mantener orden con buena velocidad	❗ Menos eficiente que AVL en búsquedas

🎯 ¿Rojo-Negro o AVL?

Necesitas…	Usa…
Inserciones y eliminaciones frecuentes	Árbol Rojo-Negro
Búsquedas súper rápidas	Árbol AVL
Implementación en la STL de C++	Rojo-Negro (por defecto)

🧨 Capítulo 26 – Comparación de los arboles

Tipo de Árbol	¿Qué es?	¿Cómo funciona?	Ventajas ☑	Desventajas ✕	¿Dónde se usa? 📌
Árbol Binario Simple (ABB)	Árbol donde cada nodo tiene máx. 2 hijos: izq. si es menor, der. si mayor	Organiza los datos comparándolos al insertar: menor a la izquierda, mayor a la derecha	✔ Fácil de entender ✔ Útil para ordenar o buscar	✕ Puede desbalancearse ✕ Se vuelve lento si se inserta mal	Para aprender árboles Pequeños programas
Árbol AVL	Árbol que se balancea solo usando rotaciones	Ajusta su forma cada vez que una rama se hace más larga que la otra	✔ Muy eficiente para búsquedas ✔ Siempre balanceado	✕ Más difícil de implementar ✕ Muchas rotaciones	Apps con muchas búsquedas Sistemas dinámicos
Árbol B / B+	Árbol con muchos hijos, ideal para manejar grandes volúmenes de datos	Divide nodos en varios caminos. El B guarda en todos; el B+ guarda solo en hojas conectadas	✔ Excelente para bases de datos ✔ Muy rápido en discos grandes	✕ Implementación compleja ✕ Menos intuitivo para estudiantes	Bases de datos Sistemas de archivos
Árbol Heap	Árbol que mantiene el orden: raíz mayor (Max-Heap) o menor (Min-Heap)	Reorganiza los valores para que el más importante esté arriba. Se usa para ordenar con HeapSort	✔ Ideal para prioridades ✔ Muy eficiente para ordenar	✕ No es estable ✕ No es bueno para buscar elementos específicos	HeapSort Colas de prioridad Juegos y eventos

Tipo de Árbol	¿Qué es?	¿Cómo funciona?	Ventajas ✅	Desventajas ❌	¿Dónde se usa? 📌
Árbol Rojo-Negro	Árbol auto-balanceado que usa colores para mantener orden	Usa reglas con colores: no hay 2 rojos seguidos, misma cantidad de negros en cada camino	✔ Inserta y elimina rápido ✔ Se usa en estructuras reales como `map`	❌ Difícil de entender al principio ❌ Lógica de colores puede confundir	STL de C++ (`map`, `set`) Bases de datos Sistemas OS