

UNIVERSIDAD NACIONAL DEL ALTIPLANO DE PUNO

Escuela Profesional de Ingeniería Estadística e Informática



ESTRUCTURA DE DATOS

Libro académico

Autora: Ruth Jhakelin Huaraya Chambi

Estudiante de Ingeniería Estadística e Informática

Asesor: Fredd Torres Cruz **Curso:** Estructura de Datos

Puno, Perú
2025

Propósito del Libro

El propósito de este libro es proporcionar una guía práctica y teórica sobre estructuras de datos en C++, orientado a estudiantes que buscan fortalecer sus fundamentos de programación. A lo largo del libro se

presentarán explicaciones claras, ejemplos aplicados, códigos comentados y ejercicios que permiten aplicar el conocimiento de forma efectiva.

Cómo Usar el Libro

Cada capítulo está diseñado con teoría breve pero esencial, ejemplos de aplicación en C++, diagramas ilustrativos, sugerencias prácticas, notas destacadas, y al final, una sección de ejercicios propuestos. Se recomienda seguir el orden secuencial, ya que los conceptos se construyen progresivamente.

Índice

- [Datos](#)
 - [Clasificación de los datos en C++](#)
 - [Datos primitivos](#)
 - [Datos derivados](#)
 - [Datos definidos por el usuario](#)
 - [Declaración y asignación de variables](#)
 - [Declaración sin asignación](#)
 - [Declaración múltiple](#)
 - [Inicialización vs asignación](#)
- [Entradas y salidas](#)
 - [Librerías necesarias](#)
 - [Instrucciones de Entrada: cin](#)
 - [Sintaxis](#)
 - [Ejemplo básico](#)
 - [Instrucciones de Salida: cout](#)
 - [Sintaxis](#)
 - [Ejemplo básico](#)
 - [Ejemplo Aplicativo](#)
- [Operadores](#)
 - [Tipos de operadores](#)
 - [Operadores aritméticos](#)
 - [Ejemplo](#)
 - [Operadores relacionales](#)
 - [Ejemplo](#)
 - [Operadores lógicos](#)
 - [Operadores de asignación](#)
 - [Operadores de incremento y decremento](#)
 - [Operadores de bits \(bitwise\)](#)
- [Condicionales](#)

- Sentencia if
 - Sintaxis
- Sentencia if...else
 - Sintaxis
 - Ejemplo
- Sentencia if...else if...else
 - Sintaxis
 - Ejemplo
- Sentencia switch
 - Sintaxis
 - Ejemplo
- Ciclos o bucles
 - Bucle for
 - Sintaxis
 - Ejemplo: Contar del 1 al 5
 - Bucle while
 - Sintaxis
 - Ejemplo: Contar hasta 5
 - Ejemplo Aplicativo: Menú hasta salir
 - Bucle do...while
 - Sintaxis
 - Ejemplo: Leer número positivo
 - Buenas Prácticas
 - Consideraciones Avanzadas
 - Ejemplo: Bucle anidado para una matriz
- Funciones
 - Ventajas del uso de funciones
 - Estructura de una función
 - Partes principales
 - Declaración y definición
 - Ejemplo básico
 - Funciones con retorno void
 - Parámetros por valor vs por referencia
 - Por valor
 - Por referencia
 - Funciones con valores por defecto
 - Funciones sobrecargadas (Overloading)
- Arreglos (Arrays)
 - Sintaxis
 - Acceso a elementos
 - Ejemplo Aplicativo: Promedio de 5 calificaciones
 - Arreglos y constantes

- Inicialización parcial y por defecto
- Ejemplo Aplicativo: Buscar valor en arreglo
- Tabla: Características de los Arreglos
- Punteros y Direcciones de Memoria
 - Direcciones de memoria
 - Tabla: Operadores clave
 - Declaración de punteros
 - Acceso a datos mediante punteros
 - Modificación de valores usando punteros
 - Ejemplo Aplicativo: Intercambio de variables
 - Punteros y arreglos
 - Punteros a funciones
- Listas Enlazadas
 - Concepto Básico de Lista Enlazada
 - Crear e Imprimir una Lista
 - Inserción de Nodos
 - Al inicio
 - Al final
 - Eliminación de Nodos
 - Ejemplo Aplicativo: Lista de Estudiantes
 - Tipos de Listas Enlazadas
 - Visualización y Diagramas
- Pilas (Stacks)
 - Concepto de Pila
 - Implementación de una pila con estructuras
 - Operaciones básicas
 - Aplicación: Verificación de paréntesis balanceados
 - Uso de la STL: std::stack
- Colas (Queues)
 - Concepto de Cola
 - Operaciones fundamentales
 - Uso de la STL: std::queue
 - Aplicación: Sistema de atención de clientes
- Recursión en C++
 - Componentes de una función recursiva
 - Factorial de un número (ejemplo clásico)
 - Secuencia Fibonacci
 - Recursión de cola (Tail Recursion)
 - Aplicación: Potenciación recursiva
 - Comparación: Recursión vs. Iteración

Capítulo 1. Datos

Los datos representan valores que pueden ser procesados por un programa. Toda operación computacional se basa en la manipulación de estos datos, que pueden ser numéricos, lógicos, caracteres o estructuras más complejas.

1.1 Clasificación de los datos en C++

En C++, los datos se clasifican en tres grandes categorías:

Datos primitivos

Son los tipos de datos fundamentales que el lenguaje proporciona de forma nativa:

- `int`: Enteros (positivos o negativos)
- `float`, `double`: Números decimales (precisión simple y doble)
- `char`: Caracteres individuales
- `bool`: Booleanos (verdadero o falso)

Datos derivados

Son construcciones creadas a partir de tipos primitivos:

- Arreglos
- Punteros
- Referencias

Datos definidos por el usuario

Son estructuras personalizadas:

- `struct`: Agrupación de múltiples datos
- `union`: Estructuras de memoria compartida
- `enum`: Conjuntos de constantes con nombre

1.2 Declaración y asignación de variables

En C++, una variable es un espacio en memoria que se reserva para almacenar un valor específico de un tipo determinado. Antes de usar una variable, debe declararse. La sintaxis general es:

```
tipo nombreVariable = valor;
```

Ejemplo básico

```
int edad = 20; // Variable de tipo entero
float estatura = 1.75; // Número con decimales
```

```
char inicial = 'R'; // Carácter individual  
bool aprobado = true; // Valor lógico
```

Declaracion sin asignación

También se puede declarar una variable sin asignarle un valor inmediatamente:

```
int nota;  
nota = 18;
```

Declaracion múltiple

Se pueden declarar varias variables del mismo tipo en una sola línea:

```
int a = 5, b = 10, c = 15;
```

Inicialización vs asignación

Inicialización	Asignación
Se asigna un valor al momento de declarar la variable.	Se declara primero la variable y luego se le asigna un valor.
int edad = 20;	int edad; edad = 20;

Ejemplo aplicativo

Problema: Diseña un programa que declare variables con sus respectivos valores.

```
#include <iostream>  
using namespace std;  
  
int main() {  
    int edad = 20;  
    float altura = 1.60;  
    char inicial = 'R';  
    bool estudiante = true;  
  
    cout << "Edad: " << edad << endl;  
    cout << "Altura: " << altura << endl;  
    cout << "Inicial: " << inicial << endl;  
    cout << "Estudiante: " << estudiante << endl;  
  
    return 0;  
}
```




Siempre verifica que el tipo de dato ingresado corresponda al tipo de variable declarada. Los errores por desajuste de tipo son comunes y pueden causar comportamientos inesperados

Capítulo 2. Entradas y salidas

Las instrucciones de entrada y salida permiten la interacción del programa con el usuario. En C++, la biblioteca "iostream" es fundamental para estas operaciones.

Entrada: recepción de datos del usuario.
Salida: presentación de resultados al usuario.

2.1 Librerías necesarias

```
#include <iostream> // Entrada/Salida estandar
using namespace std;
```

2.2 Instrucciones de Entrada: cin

Sintaxis

```
cin >> variable;
```

Ejemplo básico

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

int main() {
    int edad;
    cout << "Ingrese su edad: ";
    cin >> edad;
    cout << "Tu edad es: " << edad << " años." << endl;
    return 0;
}
```

2.3 Instrucciones de Salida: cout

Sintaxis

```
cout << "Texto a mostrar" << variable;
```

Ejemplo básico

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

int main() {
    string nombre;
    cout << "Ingrese su nombre: ";
    cin >> nombre;
    cout << "Hola, " << nombre << "!" << endl;
    return 0;
}
```

2.4 Ejemplo Aplicativo

Problema: Diseña un programa que lea el nombre, edad y promedio de un estudiante, y los muestre con formato ordenado.

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

int main() {
    string nombre;
    int edad;
    float promedio;

    cout << "Nombre del estudiante: ";
    cin >> nombre;

    cout << "Edad: ";
    cin >> edad;

    cout << "Promedio final: ";
    cin >> promedio;

    cout << "\nReporte de Estudiante\n";
    cout << nombre << endl;
    cout << edad << endl;
    cout << promedio << endl;

    return 0;
}
```

Para evitar errores, siempre inicializa tus variables. En programas grandes, los valores basura pueden causar errores difíciles de rastrear



Capítulo 3. Operadores

Los operadores en C++ permiten realizar operaciones sobre variables y valores. Son fundamentales en toda estructura lógica y aritmética de un programa.

3.1 Tipos de operadores

En C++ existen varios tipos de operadores, clasificados según su funcionalidad:

- Aritméticos
- Relacionales o de comparación
- Lógicos
- De asignación
- Incremento y decremento
- Ternario o condicional
- De bits (bitwise)
- Otros operadores especiales

3.2 Operadores aritméticos

Permiten realizar operaciones matemáticas básicas.

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo (resto de la división)

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;
    cout << "Módulo: " << (a % b) << endl;
    return 0;
}
```

3.4 Operadores relacionales

Compara valores y devuelve un valor booleano.

Operador	Descripción
<code>==</code>	Igual a
<code>!=</code>	Diferente de
<code>></code>	Mayor que
<code><</code>	Menor que
<code>>=</code>	Mayor o igual que
<code><=</code>	Menor o igual que

Ejemplo

```
int x = 5, y = 8;
cout << (x == y) << endl; // false (0)
cout << (x < y) << endl; // true (1)
```

3.5 Operadores lógicos

Se usan para construir expresiones lógicas complejas.

Operador	Descripción
<code>&&</code>	AND lógico
<code> </code>	OR lógico
<code>!</code>	NOT lógico

3.6 Operadores de asignación

Asignan valores a las variables.

Operador	Descripción
<code>=</code>	Asignación
<code>+=</code>	Suma y asignación
<code>-=</code>	Resta y asignación
<code>*=</code>	Multiplicación y asignación
<code>/=</code>	División y asignación
<code>%=</code>	Módulo y asignación

3.7 Operadores de incremento y decremento

Aumentan o disminuyen el valor de una variable en uno.

```
int n = 5;
cout << n++ << endl; // Imprime 5, luego n = 6
cout << ++n << endl; // n = 7, luego imprime 7
```

3.8 Operador ternario

Forma compacta de una estructura \texttt{if-else}.

```
int edad = 20;
string resultado = (edad >= 18) ? "Mayor de edad" : "Menor de edad";
cout << resultado << endl;
```

3.9 Operadores de bits (bitwise)

Manipulan directamente bits. Se usan en programación de bajo nivel.

Operador	Descripción
&	AND bit a bit
	OR bit a bit
^	XOR
~	NOT bit a bit
<<	Desplazamiento a la izquierda
>>	Desplazamiento a la derecha



Comprender bien los operadores te permitirá escribir código más limpio y eficiente. ¡No los subestimes!

Capítulo 4. Condicionales

En programación, una **estructura condicional** permite que un programa tome decisiones y ejecute diferentes bloques de código en función del cumplimiento de una o varias condiciones. En C++, las instrucciones condicionales más comunes son:

- `if`
- `if...else`
- `if...else if...else`
- `switch`

Estas estructuras nos permiten controlar el flujo de ejecución, haciendo nuestros programas más dinámicos e inteligentes.

4.1 Sentencia `if`

La sentencia `if` evalúa una condición booleana. Si es verdadera, se ejecuta el bloque de instrucciones asociado.

Sintaxis

```
if (condición) {
    // Instrucciones si la condición es verdadera
}
```

Ejemplo

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

int main() {
    int edad = 20;

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

    return 0;
}
```

4.2 Sentencia `if...else`

Se utiliza cuando se requiere ejecutar un bloque de código si la condición se cumple y otro diferente si no se cumple.

Sintaxis

```
if (condición) {
    // Código si condición es verdadera
} else {
    // Código si condición es falsa
}
```

Ejemplo

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

int main() {
    int numero;
    cout << "Ingrese un número: ";
    cin >> numero;

    if (numero % 2 == 0) {
        cout << "El número es par." << endl;
    } else {
        cout << "El número es impar." << endl;
    }

    return 0;
}
```

4.3 Sentencia if...else if...else

Permite evaluar múltiples condiciones secuencialmente.

Sintaxis

```
if (condición1) {
    // Código si condición1 es verdadera
} else if (condición2) {
    // Código si condición2 es verdadera
} else {
    // Código si ninguna condición es verdadera
}
```

Ejemplo

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

int main() {
```

```
int nota;
cout << "Ingrese su calificación: ";
cin >> nota;

if (nota >= 90) {
    cout << "Excelente" << endl;
} else if (nota >= 70) {
    cout << "Aprobado" << endl;
} else {
    cout << "Reprobado" << endl;
}

return 0;
}
```

4.4 Sentencia `switch`

La sentencia `switch` permite comparar el valor de una variable con una serie de constantes, de forma más ordenada que múltiples `if`.

Sintaxis

```
switch (expresión) {
    case valor1:
        // Código si expresión == valor1
        break;
    case valor2:
        // Código si expresión == valor2
        break;
    default:
        // Código si no coincide ningún caso
}
```

Nota: La expresión debe ser de tipo entero o enumerado.

Ejemplo

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

int main() {
    int opcion;

    cout << "Menú:\n1. Nuevo\n2. Abrir\n3. Salir\n";
    cout << "Seleccione una opción: ";
    cin >> opcion;

    switch (opcion) {
```

```
case 1:  
    cout << "Seleccionaste Nuevo" << endl;  
    break;  
case 2:  
    cout << "Seleccionaste Abrir" << endl;  
    break;  
case 3:  
    cout << "Saliendo del programa..." << endl;  
    break;  
default:  
    cout << "Opción no válida" << endl;  
}  
  
return 0;  
}
```

Evita condiciones innecesarias, usa llaves siempre, prefiere switch cuando sea apropiado y comenta los bloques complejos.
¡Tu código te lo agradecerá!



Capítulo 5. Ciclos o bucles

Los **bucles** son estructuras fundamentales en programación que permiten repetir una secuencia de instrucciones un número determinado de veces o mientras se cumpla una condición. En C++, los tipos de bucles más comunes son:

- `for`
- `while`
- `do...while`

Estos nos permiten automatizar tareas repetitivas, ahorrar código y mejorar la eficiencia del programa.

5.1 Bucle `for`

El bucle `for` se utiliza cuando conocemos de antemano cuántas veces queremos repetir una instrucción o conjunto de instrucciones.

Sintaxis

```
for (inicialización; condición; actualización) {
    // Instrucciones que se repiten
}
```

Ejemplo: Contar del 1 al 5

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

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

Ejemplo Aplicativo: Tabla de multiplicar

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

int main() {
    int numero;
    cout << "Ingrese un número para ver su tabla de multiplicar: ";
    cin >> numero;

    for (int i = 1; i <= 10; i++) {
```

```
    cout << numero << " x " << i << " = " << numero * i << endl;
}

return 0;
}
```

5.2 Bucle while

El bucle `while` ejecuta un bloque de instrucciones mientras se cumpla una condición booleana. Es ideal cuando no sabemos cuántas veces se debe repetir el proceso.

Sintaxis

```
while (condición) {
    // Instrucciones que se repiten
}
```

Ejemplo: Contar hasta 5

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

int main() {
    int i = 1;

    while (i <= 5) {
        cout << i << " ";
        i++;
    }

    return 0;
}
```

Ejemplo Aplicativo: Menú hasta salir

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

int main() {
    int opcion = 0;

    while (opcion != 3) {
        cout << "\nMenú:\n1. Saludar\n2. Calcular\n3. Salir\n";
        cout << "Seleccione una opción: ";
        cin >> opcion;
    }
}
```

```
if (opcion == 1)
    cout << "Hola!" << endl;
else if (opcion == 2)
    cout << "Vamos a calcular algo..." << endl;
else if (opcion != 3)
    cout << "Opción inválida." << endl;
}

return 0;
}
```

5.3 Bucle `do...while`

Este bucle garantiza al menos una ejecución del bloque de instrucciones, ya que la condición se evalúa al final.

Sintaxis

```
do {
    // Instrucciones que se repiten
} while (condición);
```

Ejemplo: Leer número positivo

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

int main() {
    int numero;

    do {
        cout << "Ingrese un número positivo: ";
        cin >> numero;
    } while (numero <= 0);

    cout << "Correcto: " << numero << endl;
    return 0;
}
```

5.4 Buenas Prácticas

- Utiliza `for` cuando el número de repeticiones es conocido.
- Usa `while` para condiciones abiertas o lectura de datos hasta un estado.
- `do...while` es ideal para validaciones de entrada que requieren al menos una ejecución.
- Evita bucles infinitos: asegúrate de modificar la condición en cada iteración.
- Comenta los bucles complejos para facilitar la lectura y mantenimiento.

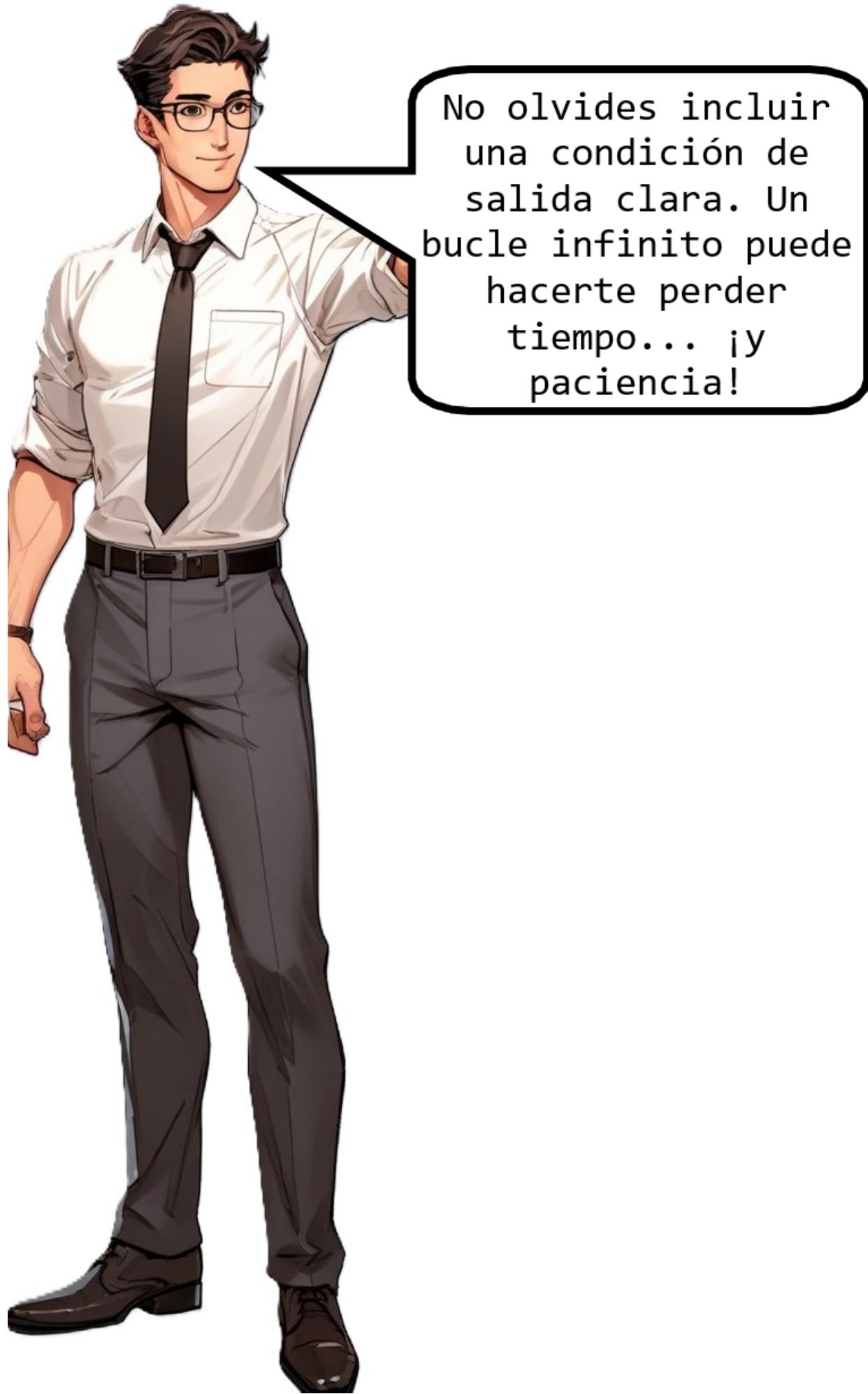
5.5 Consideraciones Avanzadas

- Puedes usar la sentencia `break` para salir de un bucle anticipadamente.
- La sentencia `continue` permite omitir el resto del ciclo y pasar a la siguiente iteración.
- Es posible anidar bucles (bucles dentro de otros bucles) para recorrer matrices, por ejemplo.

Ejemplo: Bucle anidado para una matriz

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

int main() {
    for (int fila = 1; fila <= 3; fila++) {
        for (int col = 1; col <= 3; col++) {
            cout << "(" << fila << "," << col << ") ";
        }
        cout << endl;
    }
    return 0;
}
```



Capítulo 6: Funciones

Una **función** es un bloque de código que realiza una tarea específica y puede ser reutilizado desde diferentes partes del programa. Las funciones mejoran la organización, modularidad, mantenimiento y reutilización del código.

En C++, existen dos tipos principales de funciones:

- **Funciones definidas por el usuario**
- **Funciones predefinidas (de bibliotecas)**

6.1 Ventajas del uso de funciones

- Dividir programas complejos en módulos más manejables
- Reutilizar código sin duplicarlo
- Mejorar legibilidad y depuración
- Facilitar pruebas y mantenimiento

6.2 Estructura de una función

```
<tipo_de_retorno> <nombre_funcion>(<parámetros>) {  
    // Cuerpo de la función  
    return valor;  
}
```

Partes principales:

Parte	Descripción
Tipo de retorno	El tipo de dato que devuelve la función (o void)
Nombre	Identificador de la función
Parámetros	Variables de entrada entre paréntesis
Cuerpo	Instrucciones que ejecuta la función

6.3 Declaración y definición

En C++ se puede declarar la función antes de **main()** y definirla después, o definirla antes directamente.

Ejemplo básico

```
#include <iostream>  
using namespace std;  
  
// Declaración  
int sumar(int a, int b);
```

```
int main() {
    int resultado = sumar(5, 3);
    cout << "La suma es: " << resultado << endl;
    return 0;
}

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

6.4 Funciones con retorno `void`

Las funciones que no devuelven un valor se declaran con el tipo `void`.

```
void saludar() {
    cout << "Hola desde una función!" << endl;
}

int main() {
    saludar();
    return 0;
}
```

6.5 Parámetros por valor vs por referencia

Por valor:

Se copia el valor del argumento. Los cambios no afectan a la variable original.

```
void incrementar(int x) {
    x++;
}

int main() {
    int a = 5;
    incrementar(a);
    cout << a << endl; // Imprime 5
    return 0;
}
```

Por referencia:

Se pasa la dirección de memoria. Los cambios afectan a la variable original.

```
void incrementar(int &x) {
    x++;
}

int main() {
    int a = 5;
    incrementar(a);
    cout << a << endl; // Imprime 6
    return 0;
}
```

6.6 Funciones con valores por defecto

```
int multiplicar(int a, int b = 2) {
    return a * b;
}

int main() {
    cout << multiplicar(5) << endl; // 10
    cout << multiplicar(5, 3) << endl; // 15
    return 0;
}
```

6.7 Funciones sobrecargadas (Overloading)

Permiten usar el mismo nombre de función con diferentes firmas (parámetros).

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

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

int main() {
    cout << sumar(2, 3) << endl; // 5
    cout << sumar(2.5, 4.5) << endl; // 7.0
    return 0;
}
```

Una buena función es como una herramienta bien hecha: debe hacer una sola cosa y hacerla bien. Si tu función se vuelve muy larga o hace muchas tareas, lo mejor es dividirla para que tu código sea más claro y fácil de mantener



Capítulo 7: Arreglos (Arrays)

Un **arreglo** (o *array*) es una estructura de datos que permite almacenar múltiples valores del mismo tipo bajo un solo nombre. Cada valor dentro del arreglo se identifica mediante un índice numérico.

Los arreglos son fundamentales para el manejo eficiente de conjuntos de datos y son ampliamente utilizados en programas que requieren almacenamiento secuencial.

7.1 Declaración de arreglos

Sintaxis

```
<tipo_de_dato> nombre_arreglo[tamano];
```

Ejemplo

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

Se puede inicializar en el momento de la declaración:

```
int numeros[5] = {10, 20, 30, 40, 50};
```

7.2 Acceso a elementos

Los índices comienzan en **0**.

```
cout << numeros[0]; // Accede al primer elemento  
numeros[1] = 25; // Modifica el segundo elemento
```

7.3 Recorrido de arreglos

Usando bucle **for**

```
for (int i = 0; i < 5; i++) {  
    cout << numeros[i] << " ";  
}
```

7.4 Ejemplo Aplicativo: Promedio de 5 calificaciones

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

int main() {
    float notas[5];
    float suma = 0;

    for (int i = 0; i < 5; i++) {
        cout << "Ingrese nota " << i + 1 << ": ";
        cin >> notas[i];
        suma += notas[i];
    }

    float promedio = suma / 5;
    cout << "Promedio: " << promedio << endl;

    return 0;
}
```

7.5 Arreglos y constantes

Se puede usar una constante para definir el tamaño:

```
const int TAM = 10;
int datos[TAM];
```

Esto mejora la flexibilidad del código.

7.6 Inicialización parcial y por defecto

```
int a[5] = {1, 2};
// El resto se inicializa a cero: {1, 2, 0, 0, 0}
```

7.7 Ejemplo Aplicativo: Buscar valor en arreglo

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

int main() {
    int valores[6] = {3, 7, 2, 9, 4, 8};
    int buscado;
    bool encontrado = false;

    cout << "Ingrese valor a buscar: ";
    cin >> buscado;
```

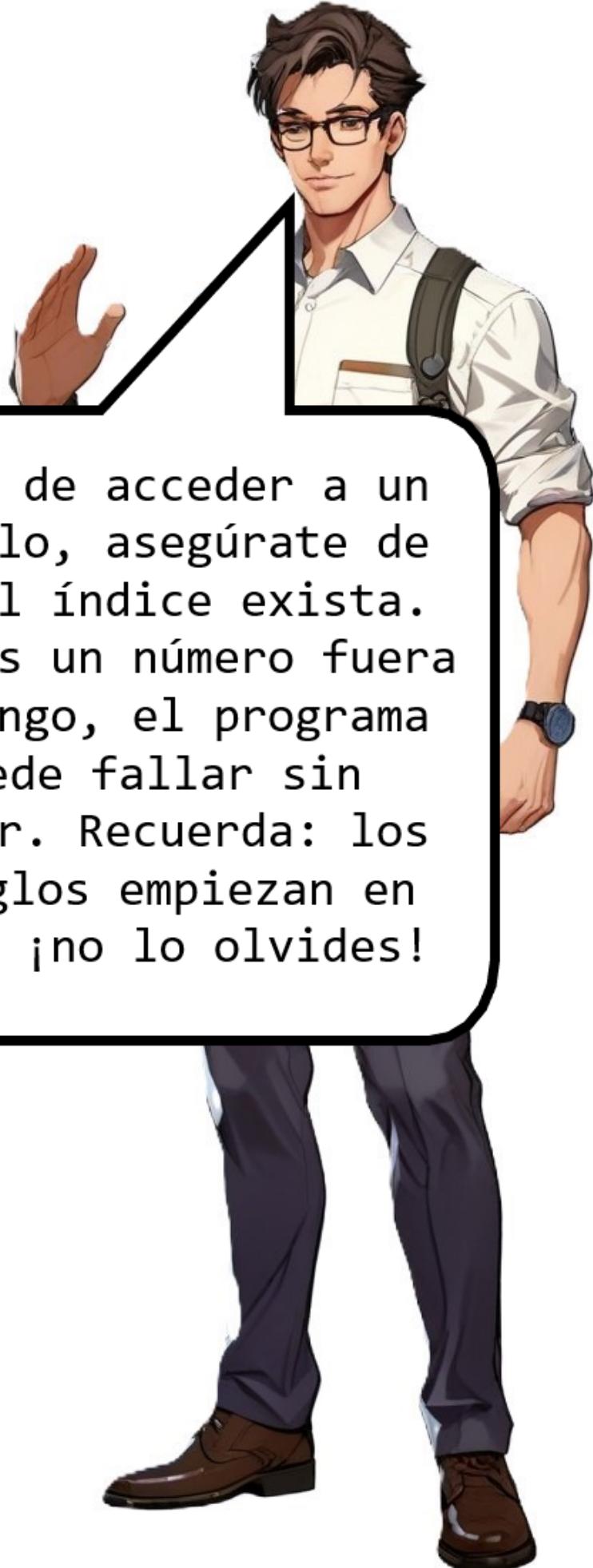
```
for (int i = 0; i < 6; i++) {
    if (valores[i] == buscado) {
        encontrado = true;
        cout << "Encontrado en la posición " << i << endl;
        break;
    }
}

if (!encontrado)
    cout << "No se encontró el valor." << endl;

return 0;
}
```

7.8 Tabla: Características de los Arreglos

Característica	Descripción
Tamaño fijo	Definido al momento de la declaración
Mismo tipo de dato	Todos los elementos deben ser del mismo tipo
Acceso por índice	Se accede mediante índices comenzando desde 0
Memoria contigua	Los elementos se almacenan de forma secuencial



Antes de acceder a un arreglo, asegúrate de que el índice exista. Si usas un número fuera de rango, el programa puede fallar sin avisar. Recuerda: los arreglos empiezan en cero, ¡no lo olvides!

Capítulo 8: Punteros y Direcciones de Memoria

Los **punteros** son variables especiales que almacenan direcciones de memoria. Permiten una manipulación más directa de los datos y son fundamentales en programación de bajo nivel, estructuras dinámicas, y optimización de recursos.

Dominar los punteros es esencial para comprender cómo funciona la memoria y mejorar el rendimiento y control de los programas en C++.

8.1 Direcciones de memoria

Toda variable ocupa una posición específica en la memoria RAM.

```
int a = 10;
cout << &a; // Imprime la dirección de memoria de 'a'
```

Tabla: Operadores clave

Símbolo	Nombre	Descripción
&	Operador de dirección	Devuelve la dirección de memoria de una variable
*	Operador de desreferencia	Accede al valor almacenado en una dirección

8.2 Declaración de punteros

```
int* ptr;
```

Significa que **ptr** es un puntero a una variable de tipo **int**.

```
int a = 5;
int* ptr = &a; // 'ptr' apunta a la dirección de 'a'
```

8.3 Acceso a datos mediante punteros

```
int a = 5;
int* p = &a;

cout << *p; // Imprime 5 (contenido de la dirección apuntada)
```

8.4 Modificación de valores usando punteros

```
int a = 10;
int* p = &a;
*p = 20;
cout << a; // Imprime 20
```

8.5 Ejemplo Aplicativo: Intercambio de variables

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

void intercambiar(int* x, int* y) {
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main() {
    int a = 5, b = 10;
    intercambiar(&a, &b);
    cout << "a = " << a << ", b = " << b << endl;
    return 0;
}
```

8.6 Punteros nulos

Un puntero puede no apuntar a ningún valor válido:

```
int* p = nullptr;
```

Esto evita errores al acceder memoria no inicializada.

8.7 Punteros y arreglos

Un puntero puede recorrer un arreglo:

```
int arr[3] = {10, 20, 30};
int* p = arr; // Equivale a &arr[0]

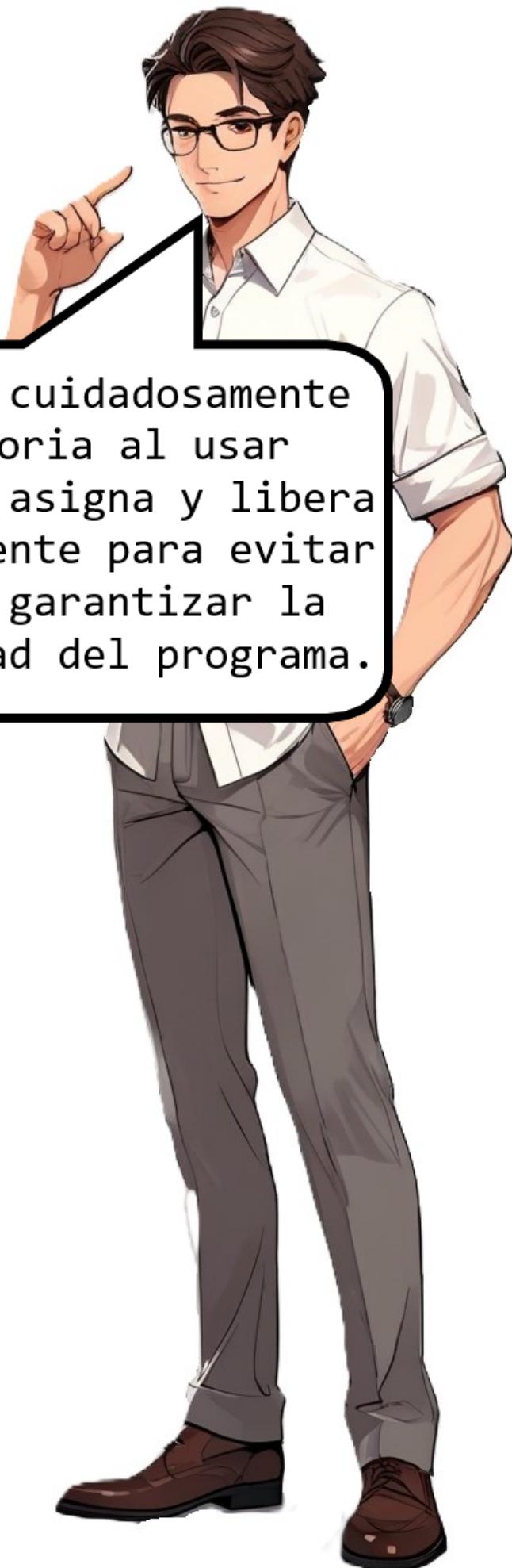
for (int i = 0; i < 3; i++) {
    cout << *(p + i) << " ";
}
```

8.8 Punteros a funciones

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

void saludar() {
    cout << "Hola desde una función!" << endl;
}

int main() {
    void (*ptrFunc)() = saludar;
    ptrFunc();
    return 0;
}
```



Capítulo 9: Listas Enlazadas

Las **listas enlazadas** son estructuras de datos dinámicas que permiten almacenar colecciones de elementos conectados mediante punteros. A diferencia de los arreglos, las listas enlazadas pueden crecer o reducirse durante la ejecución del programa, sin necesidad de redimensionar la memoria.

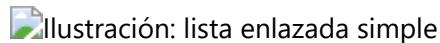
Se usan ampliamente para optimizar inserciones y eliminaciones frecuentes de datos.

9.1 Concepto Básico de Lista Enlazada

Una lista enlazada está compuesta por nodos. Cada **nodo** contiene:

- Un **dato**.
- Un **puntero** al siguiente nodo.

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



9.2 Crear e Imprimir una Lista

```
#include <iostream>  
using namespace std;  
  
struct Nodo {  
    int dato;  
    Nodo* siguiente;  
};  
  
void imprimir(Nodo* cabeza) {  
    Nodo* actual = cabeza;  
    while (actual != nullptr) {  
        cout << actual->dato << " -> ";  
        actual = actual->siguiente;  
    }  
    cout << "NULL" << endl;  
}  
  
int main() {  
    Nodo* n1 = new Nodo{1, nullptr};  
    Nodo* n2 = new Nodo{2, nullptr};  
    Nodo* n3 = new Nodo{3, nullptr};  
  
    n1->siguiente = n2;  
    n2->siguiente = n3;
```

```
    imprimir(n1);
    return 0;
}
```

9.3 Inserción de Nodos

Al inicio:

```
void insertarInicio(Nodo*& cabeza, int valor) {
    Nodo* nuevo = new Nodo{valor, cabeza};
    cabeza = nuevo;
}
```

Al final:

```
void insertarFinal(Nodo*& cabeza, int valor) {
    Nodo* nuevo = new Nodo{valor, nullptr};
    if (cabeza == nullptr) {
        cabeza = nuevo;
    } else {
        Nodo* actual = cabeza;
        while (actual->siguiente != nullptr) {
            actual = actual->siguiente;
        }
        actual->siguiente = nuevo;
    }
}
```

9.4 Eliminación de Nodos

```
void eliminar(Nodo*& cabeza, int valor) {
    Nodo* actual = cabeza;
    Nodo* anterior = nullptr;

    while (actual != nullptr && actual->dato != valor) {
        anterior = actual;
        actual = actual->siguiente;
    }

    if (actual == nullptr) return; // Valor no encontrado

    if (anterior == nullptr) {
        cabeza = actual->siguiente;
    } else {
        anterior->siguiente = actual->siguiente;
    }
}
```

```

    }

    delete actual;
}

```

9.5 Ejemplo Aplicativo: Lista de Estudiantes

```

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

struct Estudiante {
    string nombre;
    Estudiante* siguiente;
};

void imprimirLista(Estudiante* cabeza) {
    while (cabeza != nullptr) {
        cout << cabeza->nombre << endl;
        cabeza = cabeza->siguiente;
    }
}

int main() {
    Estudiante* e1 = new Estudiante{"Ana", nullptr};
    Estudiante* e2 = new Estudiante{"Luis", nullptr};
    Estudiante* e3 = new Estudiante{"Carlos", nullptr};

    e1->siguiente = e2;
    e2->siguiente = e3;

    imprimirLista(e1);
    return 0;
}

```

9.6 Tipos de Listas Enlazadas

Tipo	Característica principal
Lista simplemente enlazada	Cada nodo apunta solo al siguiente nodo
Lista doblemente enlazada	Cada nodo apunta al siguiente y al anterior
Lista circular	El último nodo apunta al primero (forma un ciclo)

9.7 Visualización y Diagramas

- **Lista simple:** 1 → 2 → 3 → NULL
- **Lista doble:** <-> 1 <-> 2 <-> 3 <->
- **Circular:** 1 → 2 → 3 → 1



Cada vez que creas un nodo en una lista enlazada, le estás pidiendo memoria al sistema. No olvides devolverla cuando ya no la uses. Así mantienes tu programa limpio y rápido

Capítulo 10: Pilas (Stacks)

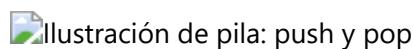
Las **pilas (stacks)** son estructuras de datos lineales que siguen el principio **LIFO**: *Last In, First Out* (el último en entrar es el primero en salir).

Se utilizan en diversos contextos como:

- Llamadas a funciones (stack de ejecución)
- Deshacer operaciones
- Evaluación de expresiones
- Recorrido de estructuras recursivas

10.1 Concepto de Pila

Una pila está compuesta por nodos donde sólo se pueden insertar (push) y eliminar (pop) elementos desde el **tope**.



10.2 Implementación de una pila con estructuras

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

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

class Pila {
private:
    Nodo* tope;

public:
    Pila() : tope(nullptr) {}

    void push(int valor) {
        Nodo* nuevo = new Nodo{valor, tope};
        tope = nuevo;
    }

    void pop() {
        if (tope == nullptr) {
            cout << "Pila vacía." << endl;
            return;
        }
        Nodo* temp = tope;
        tope = tope->siguiente;
        delete temp;
    }
}
```

```

int cima() {
    if (tope == nullptr) {
        throw runtime_error("Pila vacía");
    }
    return tope->dato;
}

bool estaVacia() {
    return tope == nullptr;
}

void imprimir() {
    Nodo* actual = tope;
    while (actual != nullptr) {
        cout << actual->dato << " -> ";
        actual = actual->siguiente;
    }
    cout << "NULL" << endl;
}

~Pila() {
    while (!estaVacia()) pop();
}
};

int main() {
    Pila p;
    p.push(10);
    p.push(20);
    p.push(30);
    p.imprimir();
    p.pop();
    p.imprimir();
    return 0;
}

```

10.3 Operaciones básicas

Operación	Descripción
push(x)	Inserta x en la cima de la pila
pop()	Elimina el elemento en la cima
cima()	Devuelve el valor de la cima
estaVacia()	Verifica si la pila está vacía

10.4 Aplicación: Verificación de paréntesis balanceados

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

bool parentesisBalanceados(const string& expr) {
    stack<char> pila;
    for (char c : expr) {
        if (c == '(') pila.push(c);
        else if (c == ')') {
            if (pila.empty()) return false;
            pila.pop();
        }
    }
    return pila.empty();
}

int main() {
    string expresion = "(a+b)*(c+(d-e))";
    cout << (parentesisBalanceados(expresion) ? "Balanceados" : "No balanceados")
<< endl;
    return 0;
}
```

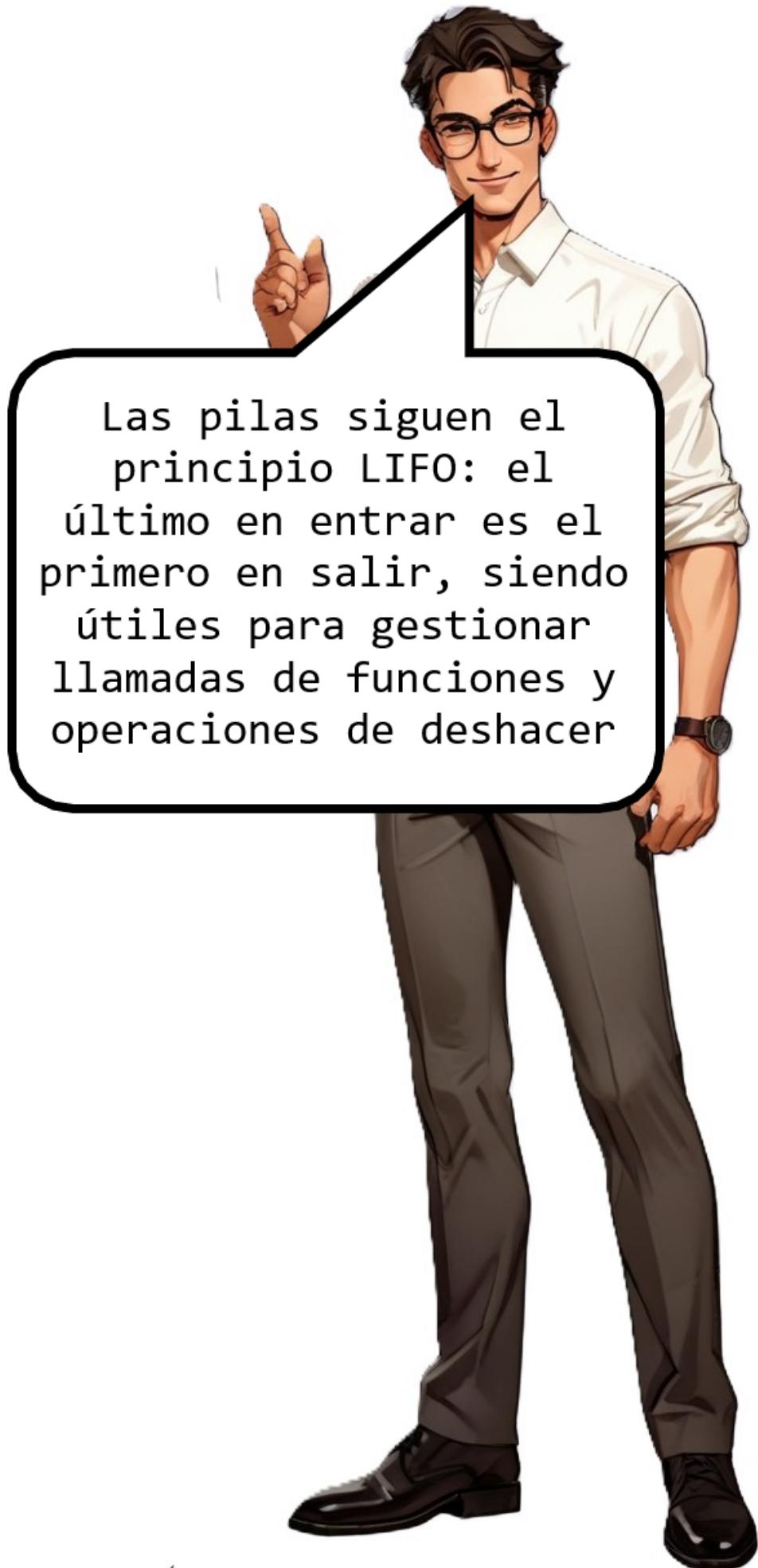
10.5 Uso de la STL: `std::stack`

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

int main() {
    stack<int> pila;
    pila.push(1);
    pila.push(2);
    pila.push(3);

    cout << "Tope: " << pila.top() << endl;
    pila.pop();
    cout << "Nuevo tope: " << pila.top() << endl;

    return 0;
}
```



Las pilas siguen el principio LIFO: el último en entrar es el primero en salir, siendo útiles para gestionar llamadas de funciones y operaciones de deshacer

Capítulo 11: Colas (Queues) en C++

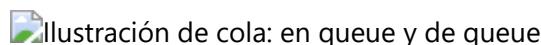
Las **colas** son estructuras de datos lineales que siguen el principio **FIFO**: *First In, First Out* (el primero en entrar es el primero en salir).

Se utilizan en:

- Manejo de procesos en sistemas operativos.
- Simulaciones de filas de espera.
- Transmisiones de datos.
- Impresoras y redes.

11.1 Concepto de Cola

Una cola permite inserciones por un extremo (final) y extracciones por el otro (frente).



11.2 Implementación de una cola con estructuras

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

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

class Cola {
private:
    Nodo* frente;
    Nodo* final;

public:
    Cola() : frente(nullptr), final(nullptr) {}

    void enqueue(int valor) {
        Nodo* nuevo = new Nodo{valor, nullptr};
        if (final == nullptr) {
            frente = final = nuevo;
        } else {
            final->siguiente = nuevo;
            final = nuevo;
        }
    }

    void dequeue() {
        if (frente == nullptr) {
            cout << "Cola vacía." << endl;
            return;
        }
    }
}
```

```

        Nodo* temp = frente;
        frente = frente->siguiente;
        if (frente == nullptr) final = nullptr;
        delete temp;
    }

    int frenteValor() {
        if (frente == nullptr) throw runtime_error("Cola vacía");
        return frente->dato;
    }

    bool estaVacia() {
        return frente == nullptr;
    }

    void imprimir() {
        Nodo* actual = frente;
        while (actual != nullptr) {
            cout << actual->dato << " <- ";
            actual = actual->siguiente;
        }
        cout << "NULL" << endl;
    }

    ~Cola() {
        while (!estaVacia()) dequeue();
    }
};

int main() {
    Cola c;
    c.enqueue(10);
    c.enqueue(20);
    c.enqueue(30);
    c.imprimir();
    c.dequeue();
    c.imprimir();
    return 0;
}

```

11.3 Operaciones fundamentales

Operación	Descripción
enqueue(x)	Inserta x al final de la cola
dequeue()	Elimina el elemento al frente
frenteValor()	Devuelve el valor del frente sin eliminarlo
estaVacia()	Verifica si la cola está vacía

11.4 Uso de la STL: `std::queue`

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

int main() {
    queue<int> cola;
    cola.push(1);
    cola.push(2);
    cola.push(3);

    cout << "Frente: " << cola.front() << endl;
    cola.pop();
    cout << "Nuevo frente: " << cola.front() << endl;

    return 0;
}
```

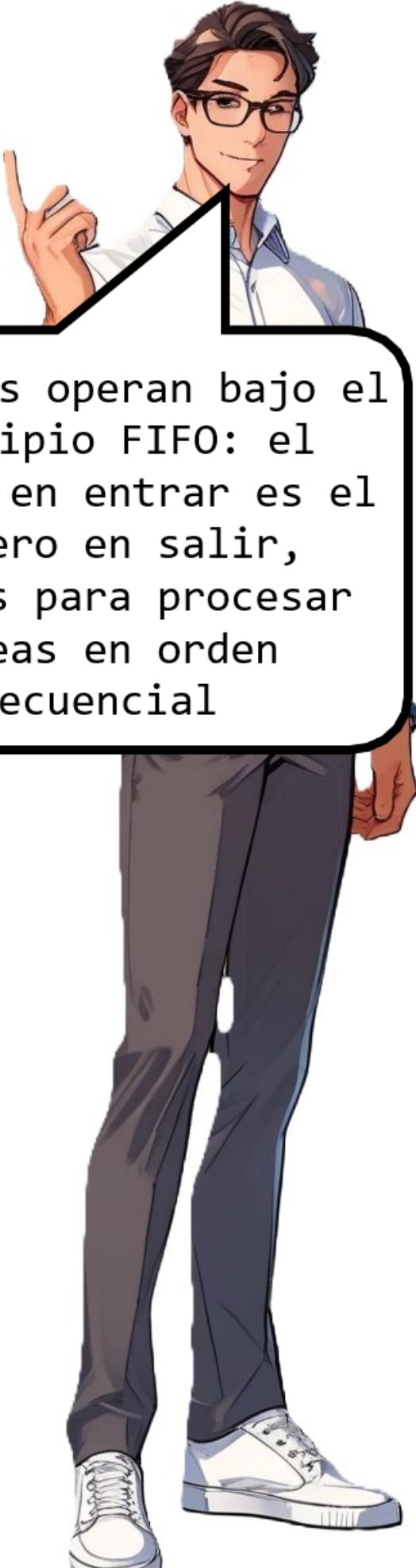
11.5 Aplicación: Sistema de atención de clientes

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

int main() {
    queue<string> clientes;
    clientes.push("Ana");
    clientes.push("Luis");
    clientes.push("María");

    while (!clientes.empty()) {
        cout << "Atendiendo a: " << clientes.front() << endl;
        clientes.pop();
    }

    return 0;
}
```



Las colas operan bajo el principio FIFO: el primero en entrar es el primero en salir, ideales para procesar tareas en orden secuencial

Capítulo 12: Recursión en C++

La **recursión** es una técnica de programación donde una función se llama a sí misma directa o indirectamente para resolver un problema.

Es especialmente útil cuando un problema puede dividirse en subproblemas más pequeños del mismo tipo.

Ejemplos comunes: factoriales, secuencia Fibonacci, recorrido de estructuras de datos, algoritmos de división y conquista.

12.1 Componentes de una función recursiva

Una función recursiva debe tener:

1. **Caso base:** Condición para detener la recursión.
2. **Paso recursivo:** Llamada a sí misma con un valor más simple o reducido.

```
int factorial(int n) {
    if (n == 0) return 1; // Caso base
    else return n * factorial(n - 1); // Paso recursivo
}
```

12.2 Factorial de un número (ejemplo clásico)

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

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

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

Flujo de llamadas:

```
factorial(5)
-> 5 * factorial(4)
-> 4 * factorial(3)
-> 3 * factorial(2)
-> 2 * factorial(1)
-> 1 (caso base)
```

12.3 Secuencia Fibonacci

```
int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Nota: Este código es ineficiente para n grandes. Se recomienda usar memoización o versión iterativa para optimizar.

12.4 Recursión de cola (Tail Recursion)

Una función recursiva es de cola cuando la llamada recursiva es la última instrucción en ejecutarse:

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

Esto facilita la optimización por parte del compilador.

12.5 Aplicación: Potenciación recursiva

```
int potencia(int base, int exponente) {
    if (exponente == 0) return 1;
    return base * potencia(base, exponente - 1);
}
```

Mejora con división y conquista:

```
int potenciaRapida(int base, int exponente) {
    if (exponente == 0) return 1;
    if (exponente % 2 == 0)
        return potenciaRapida(base * base, exponente / 2);
    else
        return base * potenciaRapida(base, exponente - 1);
}
```

12.6 Comparación: Recursión vs. Iteración

Característica	Recursión	Iteración
Simplicidad del código	Más clara para problemas recursivos	Menor uso de memoria
Uso de memoria	Mayor (stack de llamadas)	Menor
Velocidad	Puede ser menor (sin optimización)	Mayor velocidad en general

