Manual Completo de C: De Cero a Experto

Portada

MANUAL COMPLETO DE C

De Cero a Experto

Una guía didáctica y progresiva para dominar el lenguaje de programación C, desde los fundamentos hasta las estructuras de datos dinámicas y la programación a bajo nivel.

Elaborado por:

Basdonax AI

Innovación en Inteligencia Artificial y Educación Tecnológica

Instructor Experto: Manus AI

Versión: 1.0

Fecha: Noviembre 2025

Índice General

Nivel 1: Básico - Fundamentos de la Programación Estructurada

Nivel 2: Intermedio - Estructuras de Datos Simples y Modularidad

- **Módulo 5: Arreglos (Arrays)**
  - 5.1. Arreglos Unidimensionales
  - 5.2. Arreglos Multidimensionales
- **Módulo 6: Cadenas de Caracteres**
  - 6.1. Cadenas como Arreglos de `char`
  - 6.2. Funciones de la Librería `string.h`
- **Módulo 7: Estructuras y Uniones**
  - 7.1. Definición de Estructuras (`struct`)
  - 7.2. Arreglos de Estructuras
  - 7.3. Uniones (`union`) y Enumeraciones (`enum`)
- **Módulo 8: Preprocesador y Modularidad**
  - 8.1. El Preprocesador de C
  - 8.2. Compilación Modular (Archivos `.h` y `.c`)
  - 8.3. Clases de Almacenamiento
- **Resumen del Módulo Intermedio**
- **Evaluación Parcial: Nivel Intermedio**

Nivel 3: Avanzado - Punteros, Memoria Dinámica y Archivos

- **Módulo 9: Punteros (Pointers)**
  - 9.1. Fundamentos de Punteros
  - 9.2. Aritmética de Punteros
  - 9.3. Punteros y Arreglos
  - 9.4. Punteros a Punteros y Punteros a Funciones
- **Módulo 10: Memoria Dinámica**
  - 10.1. Asignación Dinámica (`malloc` , `calloc` , `realloc`)
  - 10.2. Liberación de Memoria (`free`)
  - 10.3. Arreglos Dinámicos
- **Módulo 11: Manejo de Archivos**
  - 11.1. Archivos de Texto (ASCII)
  - 11.2. Archivos Binarios
  - 11.3. Manejo de Errores en Archivos
- **Resumen del Módulo Avanzado**
- **Evaluación Parcial: Nivel Avanzado**

Nivel 4: Experto - Estructuras de Datos Complejas y Optimización

- **Módulo 12: Estructuras de Datos Dinámicas**
  - 12.1. Listas Enlazadas Simples
  - 12.2. Listas Doblemente Enlazadas y Circulares
  - 12.3. Pilas (Stacks) y Colas (Queues)
- **Módulo 13: Programación a Bajo Nivel**
  - 13.1. Operaciones a Nivel de Bit
  - 13.2. Manipulación de Memoria (`memcpy`, `memset`)
  - 13.3. Programación con `void *` (Punteros Genéricos)
- **Módulo 14: Optimización y Debugging**
  - 14.1. Técnicas de Optimización
  - 14.2. Debugging con GDB
  - 14.3. Manejo de Errores Avanzado
- **Resumen del Módulo Experto**
- **Evaluación Parcial: Nivel Experto**

Proyectos Integradores Finales

- **Proyecto 1:** Gestor de Tareas (To-Do List)
- **Proyecto 2:** Sistema de Registro de Estudiantes
- **Proyecto 3:** Juego de Adivinanza con Números
- **Proyecto 4:** Calculadora de Matrices

Contenido del Manual

Nivel 1: Básico

$(/home/ubuntu/c_manual/nivel_basico.md)

Nivel 2: Intermedio

$(/home/ubuntu/c_manual/nivel_intermedio.md)

Nivel 3: Avanzado

$(/home/ubuntu/c_manual/nivel_avanzado.md)

Nivel 4: Experto

$(/home/ubuntu/c_manual/nivel_experto.md)

Proyectos Integradores

$(/home/ubuntu/c_manual/proyectos_integradores.md)

Nivel 1: Básico - Fundamentos de la Programación Estructurada

¡Bienvenido/a al mundo de C! Este nivel es tu punto de partida. Aquí estableceremos las bases sólidas que te permitirán construir cualquier programa. Nos enfocaremos en la sintaxis esencial, el manejo de datos simples y el control de flujo.

Módulo 1: Introducción

1.1. ¿Qué es C y por qué aprenderlo?

Teoría: El lenguaje de programación **C** es un lenguaje de propósito general, imperativo y estructurado, desarrollado por Dennis Ritchie entre 1969 y 1973 en los Laboratorios Bell. Es la base de muchos sistemas operativos (como Linux y Windows), compiladores, bases de datos y lenguajes de programación modernos (como C++, C#, Java y Python).

Aprender C te da una comprensión profunda de cómo funciona una computadora, ya que te permite trabajar cerca del *hardware* y gestionar la memoria de forma manual. Esto es crucial para el desarrollo de sistemas embebidos, *firmware* y aplicaciones de alto rendimiento.

El estándar más común hoy en día es **C11** (ISO/IEC 9899:2011), aunque muchos proyectos ya utilizan **C18** o el más reciente **C23**. Nos enfocaremos en código compatible con C11 o superior.

Notas del Instructor:

> *C es como aprender a manejar un auto con caja manual. Es más difícil al principio, pero te da un control total sobre el motor y te enseña cómo funciona realmente la máquina. ¡La paciencia es clave!*

1.2. Configuración del Entorno y Primer Programa

Teoría: Para escribir y ejecutar código C, necesitamos un **compilador** (que traduce nuestro código a lenguaje de máquina) y un **editor de texto** o un **IDE** (Entorno de Desarrollo Integrado). El compilador más popular es **GCC** (GNU Compiler Collection).

El ciclo de compilación de C tiene cuatro fases:

1. **Preprocesamiento:** Se procesan las directivas (`#include`, `#define`).
2. **Compilación:** El código C se traduce a código ensamblador.
3. **Ensamblado:** El código ensamblador se convierte en código objeto.
4. **Linkeado (Enlazado):** Se combinan los archivos objeto con las librerías necesarias para crear el ejecutable final.

Código de Ejemplo Explicado: El programa clásico "Hola Mundo".

```c
// hola_mundo.c
#include <stdio.h> // Directiva de preprocesador: incluye la librería estándar de entrada/salida

int main() { // La función principal: el punto de inicio de todo programa C
    // printf es una función de la librería stdio.h que imprime texto en la consola
    printf("Hola, mundo en C!\n");

    // return 0 indica que el programa finalizó exitosamente
    return 0;
}
```

Explicación Línea por Línea:

- `#include <stdio.h>`: Le dice al preprocesador que incluya el contenido del archivo de cabecera `stdio.h`, que contiene la declaración de la función `printf`.

- `int main()`: Define la función principal. `int` es el tipo de dato que devuelve la función (un entero), y `main` es el nombre que el sistema operativo busca para empezar.
- `{ ... }`: Delimitan el cuerpo de la función `main`.
- `printf("Hola, mundo en C!\n");`: La función que imprime el texto. `\n` es un carácter de escape que inserta un salto de línea.
- `return 0;`: Devuelve el valor entero 0 al sistema operativo, indicando que todo salió bien.

Ejercicio Práctico: Modificá el programa anterior para que imprima tu nombre completo y tu ciudad de origen en dos líneas separadas.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    printf("Mi nombre es [Tu Nombre Completo].\n");
    printf("Soy de [Tu Ciudad].\n");
    return 0;
}
```

Módulo 2: Datos y Operaciones

2.1. Variables y Tipos de Datos

Teoría: Una **variable** es un espacio de memoria reservado para almacenar un valor. En C, todas las variables deben ser declaradas con un **tipo de dato** antes de ser usadas.

Tipo de Dato	Descripción	Rango (Típico)	Especificador `printf` / `scanf`
`int`	Enteros (números sin decimales)	-2 mil millones a +2 mil millones	`%d` o `%i`
`float`	Números de punto flotante (con decimales)	Precisión simple (6-7 dígitos)	`%f`
`double`	Números de punto flotante de doble precisión	Precisión doble (15 dígitos)	`%lf` (para `scanf`), `%f` o `%lf` (para `printf`)
`char`	Un solo carácter (letra, número, símbolo)	-128 a 127 (código ASCII)	`%c`

Código de Ejemplo Explicado:

```c
#include <stdio.h>

int main() {
    // Declaración e inicialización de variables
    int edad = 30;
    float altura = 1.75f; // La 'f' indica que es un float, no un double
    char inicial = 'J';

    // Reasignación de valor
    edad = 31;

    printf("Edad: %d años\n", edad);
    printf("Altura: %.2f metros\n", altura); // .2f limita a dos decimales
    printf("Inicial: %c\n", inicial);

    return 0;
}
```

Ejercicio Práctico: Declarar una variable `double` para almacenar el valor de Pi (3.14159) y una variable `int` para el radio (5). Imprimí el área de un círculo (Área = Pi * radio * radio).

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    double pi = 3.14159;
    int radio = 5;
    double area;

    area = pi * radio * radio;

    printf("El área del círculo con radio %d es: %.4f\n", radio, area);

    return 0;
}
```

2.2. Entrada y Salida Básica (`printf` y `scanf`)

Teoría:

- `printf()` : Se usa para mostrar datos en la consola. Utiliza **especificadores de formato** (`%d` , `%f` , `%c`) para indicar el tipo de dato que se va a imprimir.
- `scanf()` : Se usa para leer datos ingresados por el usuario desde el teclado. **¡ATENCIÓN!** `scanf` requiere la **dirección de memoria** de la variable donde se almacenará el dato, por eso usamos el operador `&` (ampersand) antes del nombre de la variable.

Código de Ejemplo Explicado:

```c
#include <stdio.h>

int main() {
    int numero;

    printf("Ingresá un número entero: ");
    // El & es crucial: le dice a scanf dónde está la variable 'numero' en
la memoria
    scanf("%d", &numero);

    printf("El número que ingresaste es: %d\n", numero);

    return 0;
}
```

Notas del Instructor:

> ***Error Común:*** *Olvidar el `&` en `scanf()`. Si lo olvidás, el programa intentará escribir el dato en una dirección de memoria aleatoria, lo que casi siempre causa un error de segmentación o un comportamiento inesperado.*

Ejercicio Práctico: Escribí un programa que le pida al usuario su peso (en `float`) y su edad (en `int`), y luego imprima ambos valores en una sola línea.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    float peso;
    int edad;

    printf("Ingresá tu peso (ej. 75.5): ");
    scanf("%f", &peso);

    printf("Ingresá tu edad: ");
    scanf("%d", &edad);

    printf("Tenés %d años y pesás %.1f kg.\n", edad, peso);

    return 0;
}
```

2.3. Operadores

Teoría: Los operadores son símbolos que le indican al compilador que realice manipulaciones matemáticas o lógicas específicas.

Tipo	Operador	Descripción	Ejemplo
Aritméticos	`+`, `-`, `*`, `/`	Suma, resta, multiplicación, división.	`a + b`
	`%`	Módulo (resto de la división entera).	`10 % 3` es `1`
Relacionales	`==`, `!=`	Igual a, distinto de.	`a == b`
	`>`, `<`, `>=`, `<=`	Mayor que, menor que, mayor o igual, menor o igual.	`a > b`
Lógicos	`&&`	AND lógico (ambas condiciones deben ser verdaderas).	`(a > 0) && (b < 10)`
	`||`	OR lógico (al menos una condición debe ser verdadera).	`(a == 5) || (b == 5)`
	`!`	NOT lógico (niega la condición).	`!(a == b)`
Asignación	`=`	Asigna el valor de la derecha a la variable de la izquierda.	`a = 10`
	`+=`, `-=`, `*=`…	Asignación compuesta (ej: `a += 5` es igual a `a = a + 5`).	`a += 5`

Código de Ejemplo Explicado:

```c
#include <stdio.h>

int main() {
    int a = 10;
    int b = 3;
    int resultado;

    // Operador Aritmético: Módulo
    resultado = a % b; // 10 dividido 3 es 3, resto 1
    printf("Módulo (10 %% 3): %d\n", resultado);

    // Operador de Asignación Compuesta
    a *= 2; // a ahora es 20 (10 * 2)
    printf("a despues de a *= 2: %d\n", a);

    // Operador Lógico: AND
    // (20 > 15) es verdadero (1) Y (3 < 5) es verdadero (1) -> Resultado es
1 (Verdadero)
    int logico = (a > 15) && (b < 5);
    printf("Resultado Lógico (AND): %d\n", logico);

    return 0;
}
```

Ejercicio Práctico: Creá dos variables enteras, `x = 7` e `y = 2`. Calculá el resultado de la siguiente expresión: `(x * y) + (x % y)`. Usá un operador de asignación compuesta para incrementar `x` en 3 unidades.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    int x = 7;
    int y = 2;
    int resultado;

    // (7 * 2) + (7 % 2) = 14 + 1 = 15
    resultado = (x * y) + (x % y);
    printf("Resultado de la expresión: %d\n", resultado);

    x += 3; // x ahora es 10
    printf("Nuevo valor de x: %d\n", x);

    return 0;
}
```

Módulo 3: Control de Flujo

3.1. Estructuras Condicionales (`if`, `else`, `else if`)

Teoría: Las estructuras condicionales permiten que el programa tome decisiones y ejecute diferentes bloques de código según si una condición es verdadera o falsa.

- `if`: Ejecuta un bloque de código si la condición es verdadera.
- `else`: Ejecuta un bloque de código si la condición del `if` es falsa.
- `else if`: Permite evaluar una segunda (o tercera, etc.) condición si la anterior fue falsa.

Código de Ejemplo Explicado:

```c
#include <stdio.h>

int main() {
    int nota = 85;

    if (nota >= 90) {
        printf("¡Excelente! Aprobaste con A.\n");
    } else if (nota >= 70) {
        printf("Buen trabajo. Aprobaste con B.\n");
    } else {
        printf("Necesitás practicar más. Desaprobaste.\n");
    }

    return 0;
}
```

Notas del Instructor:

> ***Operador Ternario:*** *Es una forma corta de escribir un* `if-else` *simple.* `condicion ? expresion_si_verdadera : expresion_si_falsa;` *Ejemplo:* `int max = (a > b) ? a : b;`

Ejercicio Práctico: Pedile al usuario que ingrese un número entero. Usá una estructura `if-else` para determinar si el número es par o impar. (Pista: usá el operador módulo `%`).

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    int num;

    printf("Ingresá un número: ");
    scanf("%d", &num);

    // Si el resto de la división por 2 es 0, es par
    if (num % 2 == 0) {
        printf("El número %d es PAR.\n", num);
    } else {
        printf("El número %d es IMPAR.\n", num);
    }

    return 0;
}
```

3.2. Estructuras de Repetición (Bucles)

Teoría: Los bucles permiten ejecutar un bloque de código repetidamente.

Bucle	Descripción	Cuándo usarlo
`for`	Repite un bloque un número **fijo** de veces.	Ideal para iterar sobre arreglos o cuando se conoce el número de repeticiones.
`while`	Repite un bloque **mientras** una condición sea verdadera.	Ideal cuando el número de repeticiones es desconocido (ej: leer un archivo hasta el final).
`do-while`	Similar a `while`, pero garantiza que el bloque se ejecute **al menos una vez**.	Ideal para menús de usuario o validación de entrada.

Código de Ejemplo Explicado (`for`):

```c
#include <stdio.h>

int main() {
    // Inicialización (int i = 1); Condición (i <= 5); Incremento (i++)
    for (int i = 1; i <= 5; i++) {
        printf("Iteración número: %d\n", i);
    }

    return 0;
}
```

Código de Ejemplo Explicado (`while`):

```c
#include <stdio.h>

int main() {
    int contador = 0;

    while (contador < 3) { // Mientras contador sea menor a 3
        printf("Contador: %d\n", contador);
        contador++; // ¡Importante! Si no incrementamos, es un bucle
infinito
    }

    return 0;
}
```

Ejercicio Práctico: Usá un bucle `while` para pedirle al usuario que ingrese números hasta que ingrese el número 0. Al final, imprimí un mensaje de despedida.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    int numero = -1; // Inicializamos con un valor distinto de 0 para entrar al bucle

    printf("Ingresá números (0 para salir):\n");

    while (numero != 0) {
        printf("Número: ");
        scanf("%d", &numero);
    }

    printf("¡Saliste del programa! ¡Adiós!\n");

    return 0;
}
```

3.3. Sentencias de Salto (`break` y `continue`)

Teoría:

- `break`: Termina inmediatamente el bucle más interno (`for`, `while`, `do-while` o `switch`) y el control del programa salta a la sentencia que sigue al bucle.
- `continue`: Salta la iteración actual del bucle y pasa directamente a la siguiente iteración.

Código de Ejemplo Explicado:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            continue; // Salta la iteración 5
        }

        if (i == 8) {
            break; // Termina el bucle cuando i es 8
        }

        printf("Número: %d\n", i);
    }

    // El programa continúa aquí después del break
    printf("Bucle terminado.\n");

    return 0;
}
// Salida esperada: 1, 2, 3, 4, 6, 7, Bucle terminado.
```

Ejercicio Práctico: Escribí un bucle `for` que itere del 1 al 10. Usá `continue` para no imprimir los números pares (2, 4, 6, 8, 10).

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    for (int i = 1; i <= 10; i++) {
        // Si el número es par (resto de la división por 2 es 0)
        if (i % 2 == 0) {
            continue; // Pasa a la siguiente iteración sin ejecutar el
printf
        }

        printf("Número impar: %d\n", i);
    }

    return 0;
}
```

Módulo 4: Funciones

4.1. Concepto de Función

Teoría: Una **función** es un bloque de código que realiza una tarea específica. Las funciones permiten dividir un programa grande en partes más pequeñas y manejables (modularidad), lo que facilita la lectura, el *debugging* y la reutilización del código.

Toda función tiene:

1. **Tipo de Retorno:** El tipo de dato que la función devuelve (`int` , `float` , `void` si no devuelve nada).
2. **Nombre:** Un identificador único.
3. **Parámetros:** Los valores de entrada que la función necesita (entre paréntesis).

Código de Ejemplo Explicado:

```c
#include <stdio.h>

// 1. Declaración (Prototipo): Le dice al compilador que esta función existe
int sumar(int a, int b);

int main() {
    int num1 = 5;
    int num2 = 3;
    int resultado;

    // 3. Llamada a la función
    resultado = sumar(num1, num2);

    printf("La suma es: %d\n", resultado);

    return 0;
}

// 2. Definición (Implementación): El cuerpo de la función
int sumar(int a, int b) {
    int total = a + b;
    return total; // Devuelve el resultado (tipo int)
}
```

Ejercicio Práctico: Creá una función llamada `multiplicar` que reciba dos números de tipo `double` y devuelva su producto. Llamá a la función desde `main` e imprimí el resultado.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

// Prototipo
double multiplicar(double x, double y);

int main() {
    double val1 = 4.5;
    double val2 = 2.0;
    double producto;

    producto = multiplicar(val1, val2);

    printf("El producto es: %.2f\n", producto);

    return 0;
}

// Definición
double multiplicar(double x, double y) {
    return x * y;
}
```

4.2. Paso de Argumentos y Ámbito de Variables

Teoría:

- **Paso por Valor:** En C, los argumentos se pasan a las funciones **por valor**. Esto significa que la función recibe una *copia* del valor original. Cualquier cambio que la función haga a sus parámetros internos no afecta a la variable original en la función que la llamó (`main`, en nuestro caso).
- **Ámbito (Scope):** Define dónde es accesible una variable.
  - **Local:** Declarada dentro de una función o bloque (`{}`). Solo es accesible dentro de ese bloque.
  - **Global:** Declarada fuera de cualquier función. Es accesible desde cualquier parte del programa. **Se recomienda evitar variables globales** para mantener el código modular y fácil de depurar.

Código de Ejemplo Explicado:

```c
#include <stdio.h>

// Variable global (evitar su uso excesivo)
int contador_global = 0;

void incrementar(int num) {
    num = num + 10; // Solo cambia la COPIA de 'num'
    contador_global++; // Cambia la variable global
    printf("Dentro de la función: num = %d\n", num);
}

int main() {
    int mi_numero = 5; // Variable local a main

    incrementar(mi_numero);

    // mi_numero sigue siendo 5 porque se pasó por valor
    printf("Fuera de la función: mi_numero = %d\n", mi_numero);
    printf("Contador global: %d\n", contador_global);

    return 0;
}
```

Ejercicio Práctico: Creá una función llamada `es_positivo` que reciba un entero y devuelva 1 si es positivo o 0 si es cero o negativo. Usá esta función para evaluar un número ingresado por el usuario.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int es_positivo(int num) {
    if (num > 0) {
        return 1;
    } else {
        return 0;
    }
}

int main() {
    int valor;

    printf("Ingresá un valor: ");
    scanf("%d", &valor);

    if (es_positivo(valor) == 1) {
        printf("El número es positivo.\n");
    } else {
        printf("El número es cero o negativo.\n");
    }

    return 0;
}
```

Resumen del Módulo Básico

Concepto Clave	Descripción
Sintaxis	Todo programa C comienza con `main()`. Las sentencias terminan con `;`.
Tipos de Datos	`int`, `float`, `double`, `char`. Se usan para declarar variables.
I/O	`printf()` para salida, `scanf()` con `&` para entrada.
Operadores	Aritméticos (`+`, `%`), Relacionales (`==`, `>`), Lógicos (`&&`, `
Control de Flujo	`if-else` para decisiones, `for`, `while`, `do-while` para repeticiones.
Funciones	Bloques de código reutilizables. Se pasan argumentos **por valor**.

Evaluación Parcial: Nivel Básico

Instrucciones: Resolvé los siguientes ejercicios en un solo programa.

1. **Calculadora Simple:** Pedile al usuario dos números enteros (`a` y `b`) y una operación (`+` , `-` , `*` , `/`). Usá estructuras `if-else if` para realizar la operación e imprimir el resultado.
2. **Suma de Rango:** Usá un bucle `for` para calcular la suma de todos los números enteros desde 1 hasta 100. Imprimí el resultado final.
3. **Función de Potencia:** Creá una función llamada `potencia` que reciba una base (`int`) y un exponente (`int`), y devuelva el resultado de la potencia (base^exponente). Usá un bucle `while` dentro de la función.

Solución de la Evaluación Parcial:

```c
#include <stdio.h>

// Función para el Ejercicio 3
int potencia(int base, int exp) {
    int resultado = 1;
    int i = 0;

    if (exp < 0) {
        printf("Error: Exponente negativo no soportado en este nivel.\n");
        return 0;
    }

    while (i < exp) {
        resultado *= base;
        i++;
    }
    return resultado;
}

int main() {
    // --- Ejercicio 1: Calculadora Simple ---
    int a, b;
    char op;

    printf("--- Calculadora Simple ---\n");
    printf("Ingresá el primer número: ");
    scanf("%d", &a);
    printf("Ingresá el segundo número: ");
    scanf("%d", &b);
    printf("Ingresá la operación (+, -, *, /): ");
    // El espacio antes de %c es para consumir el salto de línea anterior
    scanf(" %c", &op);

    if (op == '+') {
        printf("Resultado: %d + %d = %d\n", a, b, a + b);
    } else if (op == '-') {
        printf("Resultado: %d - %d = %d\n", a, b, a - b);
    } else if (op == '*') {
        printf("Resultado: %d * %d = %d\n", a, b, a * b);
    } else if (op == '/') {
        if (b != 0) {
            printf("Resultado: %d / %d = %.2f\n", a, b, (float)a / b);
        } else {
            printf("Error: División por cero.\n");
        }
```

```
    } else {
        printf("Operación no válida.\n");
    }

    // --- Ejercicio 2: Suma de Rango ---
    int suma = 0;
    for (int i = 1; i <= 100; i++) {
        suma += i;
    }
    printf("\n--- Suma de Rango ---\n");
    printf("La suma de 1 a 100 es: %d\n", suma); // Resultado esperado: 5050

    // --- Ejercicio 3: Función de Potencia ---
    int base = 2;
    int exp = 10;
    int res_potencia = potencia(base, exp);

    printf("\n--- Función de Potencia ---\n");
    printf("%d elevado a la %d es: %d\n", base, exp, res_potencia); //
Resultado esperado: 1024

    return 0;
}
```

Nivel 2: Intermedio - Estructuras de Datos Simples y Modularidad

¡Felicitaciones! Ya dominás los fundamentos de C. En este nivel, aprenderemos a manejar colecciones de datos de manera eficiente (arreglos y cadenas), a crear tipos de datos personalizados (estructuras) y a organizar nuestro código en múltiples archivos (modularidad).

Módulo 5: Arreglos (Arrays)

5.1. Arreglos Unidimensionales

Teoría: Un **arreglo** (o *array*) es una colección de elementos del **mismo tipo de dato** almacenados en posiciones de memoria contiguas. Esto permite acceder a cualquier elemento de forma rápida mediante un **índice**. En C, los índices de los arreglos siempre comienzan en **0**.

Declaración: `tipo nombre_arreglo[tamaño];`

Código de Ejemplo Explicado:

```c
#include <stdio.h>

int main() {
    // Declaración de un arreglo de 5 enteros
    int numeros[5];

    // Inicialización de elementos (el primer elemento es el índice 0)
    numeros[0] = 10;
    numeros[1] = 20;
    numeros[2] = 30;
    numeros[3] = 40;
    numeros[4] = 50; // Último elemento (índice 4)

    // Acceso a un elemento
    printf("El tercer elemento (índice 2) es: %d\n", numeros[2]); // Imprime
30

    // Recorrido del arreglo usando un bucle for
    printf("Todos los elementos:\n");
    for (int i = 0; i < 5; i++) {
        printf("Elemento en índice %d: %d\n", i, numeros[i]);
    }

    return 0;
}
```

Notas del Instructor:

> ***Error Común: Desbordamiento de Buffer (Buffer Overflow):*** *C no verifica si el índice que usás está dentro de los límites del arreglo. Si intentás acceder a* `numeros[5]` *en el ejemplo anterior, estarías leyendo o escribiendo en una zona de memoria que no pertenece al arreglo, lo que puede causar fallos de seguridad o errores impredecibles. ¡Siempre respetá los límites!*

Ejercicio Práctico: Creá un arreglo de 4 notas de tipo `float`. Inicializalo con los valores `7.5`, `8.0`, `9.5` y `6.0`. Luego, calculá y mostrá el promedio de las notas.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    float notas[4] = {7.5, 8.0, 9.5, 6.0};
    float suma = 0.0;
    int i;

    for (i = 0; i < 4; i++) {
        suma += notas[i];
    }

    float promedio = suma / 4;

    printf("El promedio de las notas es: %.2f\n", promedio);

    return 0;
}
```

5.2. Arreglos Multidimensionales

Teoría: Un arreglo multidimensional es esencialmente un "arreglo de arreglos". El más común es el arreglo bidimensional, que se usa para representar **matrices** o tablas.

Declaración: `tipo nombre_arreglo[filas][columnas];`

Código de Ejemplo Explicado:

```c
#include <stdio.h>

int main() {
    // Matriz de 3 filas y 2 columnas
    int matriz[3][2] = {
        {1, 2}, // Fila 0
        {3, 4}, // Fila 1
        {5, 6}  // Fila 2
    };

    // Acceso a un elemento (Fila 1, Columna 0)
    printf("Elemento [1][0]: %d\n", matriz[1][0]); // Imprime 3

    // Recorrido de la matriz usando bucles anidados
    printf("\nContenido de la matriz:\n");
    for (int i = 0; i < 3; i++) { // Bucle para las filas
        for (int j = 0; j < 2; j++) { // Bucle para las columnas
            printf("%d ", matriz[i][j]);
        }
        printf("\n"); // Salto de línea al terminar cada fila
    }

    return 0;
}
```

Ejercicio Práctico: Creá una matriz de 2x3. Pedile al usuario que ingrese los 6 valores de la matriz y luego imprimí la matriz completa en formato de tabla.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    int matriz[2][3];
    int i, j;

    printf("Ingresá 6 valores para la matriz 2x3:\n");

    // Lectura de datos
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++) {
            printf("Elemento [%d][%d]: ", i, j);
            scanf("%d", &matriz[i][j]);
        }
    }

    // Impresión de datos
    printf("\nLa matriz ingresada es:\n");
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 3; j++) {
            printf("%4d", matriz[i][j]); // %4d para alinear en 4 espacios
        }
        printf("\n");
    }

    return 0;
}
```

Módulo 6: Cadenas de Caracteres

6.1. Cadenas como Arreglos de `char`

Teoría: En C, una **cadena de caracteres** (*string*) es simplemente un arreglo unidimensional de tipo `char`. La característica fundamental es que la cadena debe terminar con el carácter nulo, `\0` (cuyo valor ASCII es 0). Este carácter es el que indica el **fin de la cadena** a las funciones de C.

Declaración e Inicialización:

1. Como arreglo de `char`: `char nombre[10] = {'J', 'u', 'a', 'n', '\0'};`

2. Como literal de cadena (más común): `char nombre[] = "Juan";` (El compilador agrega automáticamente el `\0`).

Código de Ejemplo Explicado:

```c
#include <stdio.h>

int main() {
    // El compilador reserva 5 bytes: 'H', 'o', 'l', 'a', '\0'
    char saludo[] = "Hola";

    // Acceso como arreglo
    printf("Primer caracter: %c\n", saludo[0]); // H

    // Recorrido hasta encontrar el caracter nulo
    int i = 0;
    while (saludo[i] != '\0') {
        printf("%c", saludo[i]);
        i++;
    }
    printf("\n");

    // Imprimir la cadena completa (printf sabe detenerse en '\0')
    printf("Cadena completa: %s\n", saludo); // %s es el especificador para
cadenas

    return 0;
}
```

Ejercicio Práctico: Declarar una cadena con tu apellido. Recorré la cadena e imprimí cada carácter en una línea separada.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    char apellido[] = "Perez";
    int i = 0;

    while (apellido[i] != '\0') {
        printf("%c\n", apellido[i]);
        i++;
    }

    return 0;
}
```

6.2. Funciones de la Librería `string.h`

Teoría: La librería estándar `string.h` proporciona funciones esenciales para manipular cadenas de caracteres de forma segura y eficiente.

Función	Descripción
`strlen(cadena)`	Devuelve la longitud de la cadena (sin contar el `\0`).
`strcpy(destino, origen)`	Copia la cadena `origen` en la cadena `destino`. **¡Cuidado con el tamaño!**
`strcat(destino, origen)`	Concatena (une) la cadena `origen` al final de la cadena `destino`.
`strcmp(cadena1, cadena2)`	Compara dos cadenas. Devuelve 0 si son iguales, < 0 si `cadena1` es menor, > 0 si `cadena1` es mayor.

Código de Ejemplo Explicado:

```c
#include <stdio.h>
#include <string.h> // Necesario para usar las funciones de cadena

int main() {
    char nombre[20] = "Ana"; // Reservamos espacio suficiente (20)
    char apellido[] = "Gomez";

    // 1. Longitud
    printf("Longitud de 'Ana': %zu\n", strlen(nombre)); // %zu para size_t

    // 2. Concatenación
    strcat(nombre, " "); // nombre ahora es "Ana "
    strcat(nombre, apellido); // nombre ahora es "Ana Gomez"
    printf("Nombre completo: %s\n", nombre);

    // 3. Comparación
    char clave1[] = "secreto";
    char clave2[] = "secreto";

    if (strcmp(clave1, clave2) == 0) {
        printf("Las claves son iguales.\n");
    } else {
        printf("Las claves son diferentes.\n");
    }

    return 0;
}
```

Notas del Instructor:

> ***Seguridad:*** *Las funciones* `strcpy` *y* `strcat` *son peligrosas porque no verifican el tamaño del arreglo destino, lo que puede llevar a un buffer overflow. En código moderno, se recomienda usar sus versiones seguras:* `strncpy` *y* `strncat`*, o funciones de librerías más recientes.*

Ejercicio Práctico: Creá dos cadenas: `prefijo = "Progra"` y `sufijo = "mación"`. Usá `strcat` para unirlas en una tercera cadena llamada `palabra` (asegurate de que `palabra` tenga el tamaño suficiente). Imprimí la longitud de la cadena resultante.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>
#include <string.h>

int main() {
    char prefijo[] = "Progra";
    char sufijo[] = "mación";
    // 15 es suficiente: 6 (Progra) + 7 (mación) + 1 (\0) = 14
    char palabra[15];

    // Inicializar palabra con el prefijo
    strcpy(palabra, prefijo);

    // Concatenar el sufijo
    strcat(palabra, sufijo);

    printf("Palabra resultante: %s\n", palabra);
    printf("Longitud: %zu\n", strlen(palabra)); // 13

    return 0;
}
```

Módulo 7: Estructuras y Uniones

7.1. Definición de Estructuras (`struct`)

Teoría: Una **estructura** (`struct`) es un tipo de dato definido por el usuario que permite agrupar variables de **diferentes tipos de datos** bajo un solo nombre. Esto es fundamental para modelar entidades del mundo real (por ejemplo, un `Estudiante` con nombre, edad y nota).

Declaración:

```
struct NombreEstructura {
    tipo_dato miembro1;
    tipo_dato miembro2;
    // ...
};
```

Código de Ejemplo Explicado:

```
#include <stdio.h>
#include <string.h>

// Definición de la estructura
struct Producto {
    int id;
    char nombre[50];
    float precio;
};

int main() {
    // Declaración e inicialización de una variable de tipo struct Producto
    struct Producto laptop;

    // Acceso y asignación de valores a los miembros usando el operador punto (.)
    laptop.id = 101;
    strcpy(laptop.nombre, "Laptop Gamer");
    laptop.precio = 1250.50;

    printf("ID: %d\n", laptop.id);
    printf("Nombre: %s\n", laptop.nombre);
    printf("Precio: %.2f\n", laptop.precio);

    return 0;
}
```

Ejercicio Práctico: Definí una estructura llamada `Punto` con dos miembros `int`: `x` e `y`. Creá una variable de tipo `Punto`, asignale los valores `x=5` e `y=10`, e imprimí las coordenadas.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

struct Punto {
    int x;
    int y;
};

int main() {
    struct Punto p1;

    p1.x = 5;
    p1.y = 10;

    printf("Coordenadas del punto: (%d, %d)\n", p1.x, p1.y);

    return 0;
}
```

7.2. Arreglos de Estructuras

Teoría: Así como podemos tener arreglos de enteros o flotantes, podemos tener arreglos de estructuras. Esto nos permite crear listas o bases de datos simples de objetos complejos.

Declaración: `struct NombreEstructura nombre_arreglo[tamaño];`

Código de Ejemplo Explicado:

```c
#include <stdio.h>
#include <string.h>

struct Libro {
    char titulo[100];
    int anio;
};

int main() {
    // Arreglo de 3 estructuras Libro
    struct Libro biblioteca[3];

    // Inicialización del primer libro
    strcpy(biblioteca[0].titulo, "El Quijote");
    biblioteca[0].anio = 1605;

    // Inicialización del segundo libro
    strcpy(biblioteca[1].titulo, "Cien Años de Soledad");
    biblioteca[1].anio = 1967;

    // Recorrido e impresión
    for (int i = 0; i < 2; i++) {
        printf("Libro %d: %s (Año: %d)\n", i + 1, biblioteca[i].titulo,
biblioteca[i].anio);
    }

    return 0;
}
```

Ejercicio Práctico: Usando la estructura `Producto` del tema anterior, creá un arreglo para almacenar 2 productos. Pedile al usuario que ingrese el nombre y el precio de ambos productos. Luego, imprimí el nombre del producto más caro.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>
#include <string.h>

struct Producto {
    char nombre[50];
    float precio;
};

int main() {
    struct Producto inventario[2];
    int i;

    for (i = 0; i < 2; i++) {
        printf("Producto %d:\n", i + 1);
        printf("  Nombre: ");
        scanf("%s", inventario[i].nombre); // scanf funciona bien para una sola palabra
        printf("  Precio: ");
        scanf("%f", &inventario[i].precio);
    }

    // Encontrar el más caro
    int indice_mas_caro = 0;
    if (inventario[1].precio > inventario[0].precio) {
        indice_mas_caro = 1;
    }

    printf("\nEl producto más caro es: %s (%.2f)\n",
           inventario[indice_mas_caro].nombre,
           inventario[indice_mas_caro].precio);

    return 0;
}
```

7.3. Uniones (`union`) y Enumeraciones (`enum`)

Teoría:

- **Uniones (`union`):** Similar a una estructura, pero todos sus miembros comparten la **misma ubicación de memoria**. El tamaño de la unión es igual al tamaño de su miembro más grande. Solo podés usar un miembro a la vez. Se usa para ahorrar

memoria cuando sabés que solo uno de varios campos será relevante en un momento dado.

- **Enumeraciones (`enum`):** Permite definir un conjunto de constantes enteras con nombres simbólicos, lo que hace el código más legible. Por defecto, el primer nombre tiene el valor 0, el segundo 1, y así sucesivamente.

Código de Ejemplo Explicado:

```c
#include <stdio.h>

// 1. Definición de la Unión
union Dato {
    int entero;
    float flotante;
    char caracter;
};

// 2. Definición de la Enumeración
enum DiaSemana {
    LUNES, // 0
    MARTES, // 1
    MIERCOLES, // 2
    JUEVES, // 3
    VIERNES, // 4
    SABADO, // 5
    DOMINGO // 6
};

int main() {
    // Uso de la Unión
    union Dato d;
    d.entero = 10;
    printf("Entero: %d\n", d.entero);

    d.flotante = 3.14; // Sobrescribe el valor del entero
    printf("Flotante: %.2f\n", d.flotante);

    // Uso de la Enumeración
    enum DiaSemana hoy = MIERCOLES;

    if (hoy == MIERCOLES) {
        printf("Hoy es el día %d de la semana.\n", hoy); // Imprime 2
    }

    return 0;
}
```

Ejercicio Práctico: Definí una enumeración para los colores primarios (`ROJO`, `VERDE`, `AZUL`). Imprimí el valor numérico de `AZUL`.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

enum Color {
    ROJO, // 0
    VERDE, // 1
    AZUL // 2
};

int main() {
    enum Color color_favorito = AZUL;

    printf("El valor numérico de AZUL es: %d\n", color_favorito);

    return 0;
}
```

Módulo 8: Preprocesador y Modularidad

8.1. El Preprocesador de C

Teoría: El **preprocesador** es la primera fase del ciclo de compilación. Procesa el código fuente antes de que el compilador real lo vea. Sus directivas comienzan con el símbolo `#`.

Directiva	Descripción
`#include`	Inserta el contenido de otro archivo (cabecera) en el punto donde se encuentra la directiva.
`#define`	Define una **macro** o una **constante simbólica**. El preprocesador reemplaza todas las ocurrencias del nombre por su valor antes de la compilación.
`#undef`	Elimina una definición de macro.
`#ifdef`, `#ifndef`	Compilación condicional: incluye un bloque de código solo si una macro está definida (`#ifdef`) o no definida (`#ifndef`).

Código de Ejemplo Explicado:

```c
#include <stdio.h>

// Definición de una constante simbólica
#define PI 3.14159
// Definición de una macro con argumentos
#define CUADRADO(x) ((x) * (x))

int main() {
    float radio = 5.0;

    // El preprocesador reemplaza PI por 3.14159
    float area = PI * CUADRADO(radio);

    printf("El área es: %.2f\n", area);

    return 0;
}
```

Notas del Instructor:

> ***Buenas Prácticas con Macros:*** *Siempre encerrá los argumentos de las macros entre paréntesis, como en* `((x) * (x))`*, para evitar errores de precedencia de operadores.*

Ejercicio Práctico: Definí una macro llamada `MAX(a, b)` que devuelva el valor más grande entre `a` y `b` usando el operador ternario. Usala para encontrar el máximo entre 15 y 25.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

// Macro que devuelve el máximo entre a y b
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int main() {
    int num1 = 15;
    int num2 = 25;

    int maximo = MAX(num1, num2);

    printf("El máximo entre %d y %d es: %d\n", num1, num2, maximo);

    return 0;
}
```

8.2. Compilación Modular (Archivos `.h` y `.c`)

Teoría: La **modularidad** es la práctica de dividir un programa grande en archivos más pequeños y manejables. En C, esto se logra separando la **declaración** de las funciones (en archivos de cabecera `.h`) de su **implementación** (en archivos fuente `.c`).

- **Archivo `.h` (Cabecera):** Contiene prototipos de funciones, definiciones de estructuras (`struct`), uniones (`union`) y macros (`#define`). Es la "interfaz" pública del módulo.
- **Archivo `.c` (Fuente):** Contiene la implementación real de las funciones.

Protección de Cabecera (*Include Guards*): Para evitar que el mismo archivo de cabecera se incluya varias veces en un mismo archivo fuente (lo que causaría errores de redefinición), se usan las directivas de compilación condicional:

```
// archivo_ejemplo.h
#ifndef ARCHIVO_EJEMPLO_H // Si no está definida la macro
#define ARCHIVO_EJEMPLO_H // Definila
// ... Contenido de la cabecera ...
#endif // Fin de la protección
```

Ejercicio Práctico (Conceptual): Explicá brevemente por qué es necesario usar la protección de cabecera (`#ifndef` , `#define` , `#endif`) en los archivos `.h` .

Espacio de Práctica:

> *Respuesta:*

Solución Esperada:

> *La protección de cabecera es necesaria para evitar que el compilador procese el mismo archivo `.h` varias veces durante la compilación. Si un archivo `.h` se incluye en dos o más archivos `.c` que luego se enlazan, o si un archivo `.h` incluye a otro, las definiciones de estructuras o prototipos de funciones se repetirían, causando errores de redefinición. Las directivas `#ifndef` , `#define` y `#endif` aseguran que el contenido del archivo `.h` solo se incluya una vez.*

8.3. Clases de Almacenamiento

Teoría: Las **clases de almacenamiento** definen el **ámbito** (*scope*), la **visibilidad** y el **tiempo de vida** de una variable o función.

Clase	Descripción	Ámbito	Tiempo de Vida
`auto`	Por defecto para variables locales.	Local (dentro de la función).	Mientras la función se ejecuta.
`extern`	Indica que la variable o función está definida en **otro archivo fuente**. Se usa para compartir variables globales entre módulos.	Global (en todos los archivos).	Todo el programa.
`static`	**Variables Locales:** Mantiene su valor entre llamadas a la función. **Variables Globales/Funciones:** Limita su visibilidad al archivo donde está definida.	Local o de archivo.	Todo el programa.
`register`	Sugiere al compilador que almacene la variable en un registro del CPU para un acceso más rápido (raramente usado hoy en día).	Local.	Mientras la función se ejecuta.

Código de Ejemplo Explicado (`static`):

```c
#include <stdio.h>

void contador_llamadas() {
    // 'static' hace que 'contador' mantenga su valor entre llamadas
    static int contador = 0;
    contador++;
    printf("Llamada número: %d\n", contador);
}

int main() {
    contador_llamadas(); // 1
    contador_llamadas(); // 2
    contador_llamadas(); // 3

    return 0;
}
```

Ejercicio Práctico: Explicá la diferencia entre una variable local declarada con `auto` y una declarada con `static` dentro de una función.

Espacio de Práctica:

> *Respuesta:*

Solución Esperada:

> *Una variable local `auto` (por defecto) se crea cada vez que se llama a la función y se destruye al salir, por lo que su valor se reinicia en cada llamada. Una variable local `static` se inicializa solo la primera vez que se llama a la función y **mantiene su valor** entre llamadas sucesivas. Su tiempo de vida es el de todo el programa, aunque su ámbito sigue siendo local a la función.*

Resumen del Módulo Intermedio

Concepto Clave	Descripción
Arreglos	Colecciones de elementos del mismo tipo, indexados desde 0.
Cadenas	Arreglos de `char` terminados con el carácter nulo (`\0`).
`string.h`	Librería para manipulación de cadenas (`strlen` , `strcpy` , `strcat` , `strcmp`).
Estructuras	`struct` agrupa variables de diferentes tipos bajo un solo nombre.
Uniones/Enum	`union` comparte memoria; `enum` define constantes simbólicas.
Preprocesador	Primera fase de compilación. Usa `#include` , `#define` , `#ifndef` .
Modularidad	Separación de código en archivos `.h` (declaración) y `.c` (implementación).
`static`	Limita la visibilidad a un archivo o mantiene el valor de una variable local.

Evaluación Parcial: Nivel Intermedio

Instrucciones: Resolvé los siguientes ejercicios en un solo programa.

1. **Inversión de Cadena:** Escribí una función que reciba una cadena de caracteres y la imprima al revés.
2. **Búsqueda en Arreglo de Estructuras:** Usando la estructura `Libro` (título, año), creá un arreglo de 3 libros. Pedile al usuario un año y mostrá el título de todos los libros publicados en ese año.
3. **Contador Estático:** Usá la función `contador_llamadas` (con variable `static`) del tema 8.3 y llamala 5 veces.

Solución de la Evaluación Parcial:

```c
#include <stdio.h>
#include <string.h>

// Estructura para Ejercicio 2
struct Libro {
    char titulo[100];
    int anio;
};

// Función para Ejercicio 1
void invertir_cadena(char cadena[]) {
    int longitud = strlen(cadena);
    printf("Cadena invertida: ");
    for (int i = longitud - 1; i >= 0; i--) {
        printf("%c", cadena[i]);
    }
    printf("\n");
}

// Función para Ejercicio 3
void contador_llamadas() {
    static int contador = 0;
    contador++;
    printf("Llamada al contador estático: %d\n", contador);
}

int main() {
    // --- Ejercicio 1: Inversión de Cadena ---
    printf("--- Ejercicio 1 ---\n");
    char texto[] = "Basdonax AI";
    printf("Cadena original: %s\n", texto);
    invertir_cadena(texto);

    // --- Ejercicio 2: Búsqueda en Arreglo de Estructuras ---
    printf("\n--- Ejercicio 2 ---\n");
    struct Libro biblioteca[3] = {
        {"El Principito", 1943},
        {"1984", 1949},
        {"Un Mundo Feliz", 1932}
    };

    int anio_buscado;
    printf("Ingresá un año para buscar libros: ");
    scanf("%d", &anio_buscado);
```

```
    printf("Libros publicados en %d:\n", anio_buscado);
    int encontrados = 0;
    for (int i = 0; i < 3; i++) {
        if (biblioteca[i].anio == anio_buscado) {
            printf("- %s\n", biblioteca[i].titulo);
            encontrados++;
        }
    }
    if (encontrados == 0) {
        printf("No se encontraron libros en ese año.\n");
    }

    // --- Ejercicio 3: Contador Estático ---
    printf("\n--- Ejercicio 3 ---\n");
    for (int i = 0; i < 5; i++) {
        contador_llamadas();
    }

    return 0;
}
```

Nivel 3: Avanzado - Punteros, Memoria Dinámica y Archivos

¡Llegaste al corazón de C! Este nivel es crucial, ya que abordaremos el concepto más poderoso y, a la vez, más complejo del lenguaje: los **punteros**. Dominar los punteros te dará un control total sobre la memoria de tu programa, permitiéndote crear estructuras de datos dinámicas y manejar archivos de forma eficiente.

Módulo 9: Punteros (Pointers)

9.1. Fundamentos de Punteros

Teoría: Un **puntero** es una variable que almacena la **dirección de memoria** de otra variable. En lugar de almacenar un valor directamente, almacena la ubicación (la "dirección") donde ese valor está guardado.

Para trabajar con punteros, necesitamos dos operadores clave:

1. **Operador de Dirección (`&` - *Address-of*):** Devuelve la dirección de memoria de una variable.
2. **Operador de Indirección (`*` - *Dereference*):** Accede al valor almacenado en la dirección de memoria apuntada por el puntero.

Declaración: `tipo_dato *nombre_puntero;`

Código de Ejemplo Explicado:

```c
#include <stdio.h>

int main() {
    int valor = 42; // Variable entera
    int *puntero_a_valor; // Puntero a un entero

    // 1. Asignar la dirección de 'valor' al puntero
    puntero_a_valor = &valor;

    printf("Valor de 'valor': %d\n", valor); // 42
    printf("Dirección de 'valor' (con &): %p\n", &valor); // Imprime la
dirección (ej: 0x7ffe...)
    printf("Valor del puntero (la dirección): %p\n", puntero_a_valor); //
Imprime la misma dirección

    // 2. Acceder al valor a través del puntero (Indirección)
    printf("Valor apuntado (con *): %d\n", *puntero_a_valor); // 42

    // 3. Modificar el valor a través del puntero
    *puntero_a_valor = 99;
    printf("Nuevo valor de 'valor': %d\n", valor); // 99

    return 0;
}
```

Notas del Instructor:

> ***El `*` tiene doble función:*** *Se usa para* ***declarar*** *un puntero (`int *p;`) y para* ***desreferenciar****lo (acceder al valor: `*p = 10;`). ¡No los confundas!*

Ejercicio Práctico: Declarar una variable `float` llamada `pi` con valor `3.14159`. Declarar un puntero a `float` y hacer que apunte a `pi`. Usar el puntero para cambiar el valor de `pi` a `3.14`. Imprimir el valor final de `pi` usando la variable original.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    float pi = 3.14159;
    float *p_pi;

    p_pi = &pi;

    printf("Valor inicial de pi: %.5f\n", pi);

    // Modificar a través del puntero
    *p_pi = 3.14;

    printf("Valor final de pi: %.2f\n", pi);

    return 0;
}
```

9.2. Aritmética de Punteros

Teoría: La **aritmética de punteros** es la capacidad de sumar o restar enteros a un puntero. Cuando sumás `1` a un puntero, el compilador no le suma 1 byte, sino que le suma el **tamaño del tipo de dato** al que apunta.

Si `p` es un puntero a `int` (que ocupa 4 bytes), entonces `p + 1` apunta a la siguiente posición de memoria donde podría comenzar el próximo entero.

Operaciones Válidas:

- Suma/Resta de un entero a un puntero (`p + 1` , `p - 2`).
- Resta de dos punteros del mismo tipo (`p2 - p1`). El resultado es el número de elementos entre ellos.
- Comparación de punteros (`p1 == p2` , `p1 < p2`).

Código de Ejemplo Explicado:

```
#include <stdio.h>

int main() {
    int numeros[] = {10, 20, 30, 40};
    int *p = numeros; // 'p' apunta al primer elemento (numeros[0])

    printf("Dirección inicial de p: %p\n", p);
    printf("Valor inicial de *p: %d\n", *p); // 10

    // Aritmética de punteros: p + 1
    p = p + 1; // 'p' ahora apunta a numeros[1]

    printf("Dirección de p + 1: %p\n", p);
    printf("Valor de *(p + 1): %d\n", *p); // 20

    // Otra forma de acceder al siguiente elemento
    printf("Valor de *(p + 1): %d\n", *(p + 1)); // 30 (p ya está en el 20,
+1 es el 30)

    return 0;
}
```

Ejercicio Práctico: Usá la aritmética de punteros para recorrer el arreglo `char letras[] = {'A', 'B', 'C', 'D'}` e imprimir sus elementos.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    char letras[] = {'A', 'B', 'C', 'D'};
    char *p = letras;

    for (int i = 0; i < 4; i++) {
        // Accedemos al valor en la posición p + i
        printf("Elemento %d: %c\n", i, *(p + i));
    }

    return 0;
}
```

9.3. Punteros y Arreglos

Teoría: En C, el nombre de un arreglo (sin corchetes) es, en la mayoría de los contextos, tratado como un **puntero constante** al primer elemento del arreglo. Esta es la razón por la que la aritmética de punteros funciona tan bien con arreglos.

Notación de Arreglo	Notación de Puntero	Significado
`arreglo[i]`	`*(arreglo + i)`	El valor del elemento en el índice `i`.
`arreglo`	`&arreglo[0]`	La dirección del primer elemento.

Código de Ejemplo Explicado:

```c
#include <stdio.h>

// Función que recibe un arreglo (en realidad, recibe un puntero)
void imprimir_arreglo(int *arr, int tamano) {
    for (int i = 0; i < tamano; i++) {
        // Podemos usar notación de arreglo o de puntero
        printf("%d ", arr[i]); // Es equivalente a *(arr + i)
    }
    printf("\n");
}

int main() {
    int datos[] = {100, 200, 300};

    // El nombre del arreglo 'datos' se convierte en un puntero a int
    imprimir_arreglo(datos, 3);

    return 0;
}
```

Ejercicio Práctico: Creá una función llamada `sumar_elementos` que reciba un puntero a un entero y el tamaño del arreglo. La función debe sumar todos los elementos del arreglo usando **solo notación de punteros** (`*p` , `p++`).

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int sumar_elementos(int *p_arr, int tamano) {
    int suma = 0;

    for (int i = 0; i < tamano; i++) {
        suma += *p_arr; // Suma el valor apuntado
        p_arr++; // Mueve el puntero al siguiente elemento
    }

    return suma;
}

int main() {
    int numeros[] = {5, 10, 15, 20};
    int total = sumar_elementos(numeros, 4);

    printf("La suma total es: %d\n", total); // 50

    return 0;
}
```

9.4. Punteros a Punteros y Punteros a Funciones

Teoría:

- **Puntero a Puntero (Doble Indirección):** Una variable que almacena la dirección de otro puntero. Se declara con doble asterisco: `int **pp;`. Se usa comúnmente para modificar un puntero dentro de una función (ver Módulo 10) o para arreglos de cadenas.
- **Puntero a Función:** Un puntero que almacena la dirección de una función. Permite pasar funciones como argumentos a otras funciones (*callbacks*).

Declaración de Puntero a Función: `tipo_retorno (*nombre_puntero)(tipo_parametro1, tipo_parametro2, ...);`

Código de Ejemplo Explicado (Puntero a Función):

```c
#include <stdio.h>

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

int restar(int a, int b) {
    return a - b;
}

// Función que acepta un puntero a función como argumento
int operar(int x, int y, int (*operacion)(int, int)) {
    return operacion(x, y);
}

int main() {
    // Declaración e inicialización del puntero a función
    int (*p_func)(int, int) = sumar;

    // Llamada a través del puntero
    int resultado_suma = p_func(10, 5); // Llama a sumar(10, 5)
    printf("Resultado Suma: %d\n", resultado_suma); // 15

    // Reasignamos el puntero para que apunte a 'restar'
    p_func = restar;
    int resultado_resta = operar(10, 5, p_func); // Llama a operar(10, 5,
restar)
    printf("Resultado Resta (con operar): %d\n", resultado_resta); // 5

    return 0;
}
```

Ejercicio Práctico: Declarar un puntero a puntero a `char` (`char **pp_cadena`). Asignarle la dirección de un puntero a `char` que apunta a la cadena "Doble Puntero". Imprimir la cadena usando doble desreferenciación.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    char *p_cadena = "Doble Puntero";
    char **pp_cadena = &p_cadena; // pp_cadena apunta a p_cadena

    // *pp_cadena es p_cadena (la dirección de la cadena)
    // **pp_cadena es el primer caracter de la cadena ('D')

    printf("La cadena es: %s\n", *pp_cadena); // Imprime la cadena completa
    printf("El primer caracter es: %c\n", **pp_cadena); // Imprime 'D'

    return 0;
}
```

Módulo 10: Memoria Dinámica

10.1. Asignación Dinámica (`malloc`, `calloc`, `realloc`)

Teoría: La **memoria dinámica** es memoria que se asigna en tiempo de ejecución (mientras el programa se está ejecutando), a diferencia de la memoria estática o de pila que se asigna en tiempo de compilación. Esta memoria se conoce como **Heap** o **Montón**.

Las funciones clave de la librería `stdlib.h` son:

- `void *malloc(size_t size)`: Asigna un bloque de memoria del tamaño especificado en bytes y devuelve un puntero `void *` al inicio del bloque. El contenido de la memoria no está inicializado.
- `void *calloc(size_t num, size_t size)`: Asigna memoria para un arreglo de `num` elementos, cada uno de tamaño `size`, e **inicializa todos los bytes a cero**.
- `void *realloc(void *ptr, size_t size)`: Cambia el tamaño del bloque de memoria apuntado por `ptr` al nuevo `size`.

Código de Ejemplo Explicado (`malloc`):

```c
#include <stdio.h>
#include <stdlib.h> // Necesario para malloc y free

int main() {
    int *p_entero;

    // Asignar memoria para un solo entero (sizeof(int) bytes)
    p_entero = (int *)malloc(sizeof(int));

    // Verificar si la asignación fue exitosa
    if (p_entero == NULL) {
        printf("Error: No se pudo asignar memoria.\n");
        return 1;
    }

    // Usar la memoria asignada
    *p_entero = 100;
    printf("Valor asignado dinámicamente: %d\n", *p_entero);

    // Liberar la memoria (¡CRUCIAL!)
    free(p_entero);
    p_entero = NULL; // Buena práctica: evitar punteros colgantes

    return 0;
}
```

Ejercicio Práctico: Usá `calloc` para asignar memoria para 5 números enteros. Imprimí el valor de los 5 elementos para demostrar que `calloc` los inicializa a cero.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p_numeros;
    int cantidad = 5;

    // Asignar memoria para 5 enteros e inicializar a 0
    p_numeros = (int *)calloc(cantidad, sizeof(int));

    if (p_numeros == NULL) {
        printf("Error de memoria.\n");
        return 1;
    }

    printf("Valores inicializados por calloc:\n");
    for (int i = 0; i < cantidad; i++) {
        printf("Elemento %d: %d\n", i, p_numeros[i]);
    }

    free(p_numeros);

    return 0;
}
```

10.2. Liberación de Memoria (`free`)

Teoría: La función `free(void *ptr)` libera el bloque de memoria apuntado por `ptr` que fue previamente asignado con `malloc`, `calloc` o `realloc`.

Fugas de Memoria (*Memory Leaks*): Ocurren cuando asignás memoria dinámica pero olvidás liberarla con `free`. El sistema operativo no puede reutilizar esa memoria hasta que el programa termina, lo que puede agotar los recursos del sistema en programas de larga ejecución.

Punteros Colgantes (*Dangling Pointers*): Ocurren cuando liberás la memoria con `free(p)` pero el puntero `p` sigue apuntando a esa zona de memoria liberada. Si intentás desreferenciar `p` después de `free`, el comportamiento es indefinido y peligroso. **Buena práctica:** Asignar `NULL` al puntero después de liberarlo (`p = NULL;`).

Código de Ejemplo Explicado (Puntero Colgante):

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *p = (int *)malloc(sizeof(int));

    if (p == NULL) return 1;

    *p = 50;
    printf("Antes de free: %d\n", *p);

    free(p); // La memoria es liberada

    // *p = 100; // ¡PELIGRO! Esto es un puntero colgante. No se debe hacer.

    p = NULL; // ¡Buena práctica!

    if (p == NULL) {
        printf("Puntero seguro (NULL).\n");
    }

    return 0;
}
```

Ejercicio Práctico: Escribí una función que asigne memoria para una cadena de 100 caracteres, la inicialice con un mensaje y luego la libere. Asegurate de asignar `NULL` al puntero después de `free`.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void manejar_memoria() {
    char *mensaje = (char *)malloc(100 * sizeof(char));

    if (mensaje == NULL) {
        printf("Fallo al asignar memoria.\n");
        return;
    }

    strcpy(mensaje, "Memoria asignada y lista para usar.");
    printf("Mensaje: %s\n", mensaje);

    // Liberación
    free(mensaje);
    printf("Memoria liberada.\n");

    // Prevención de puntero colgante
    mensaje = NULL;
}

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

10.3. Arreglos Dinámicos

Teoría: La asignación dinámica es ideal para crear arreglos cuyo tamaño no se conoce hasta que el programa se ejecuta (por ejemplo, el usuario ingresa el tamaño).

Creación: `int *arr = (int *)malloc(cantidad * sizeof(int));`

Redimensionamiento: La función `realloc` permite cambiar el tamaño de un bloque de memoria previamente asignado. Si el nuevo tamaño es mayor, el contenido anterior se mantiene. Si es menor, se trunca.

Código de Ejemplo Explicado (`realloc`):

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *arr = (int *)malloc(3 * sizeof(int)); // Arreglo de 3 elementos

    if (arr == NULL) return 1;

    arr[0] = 1;
    arr[1] = 2;
    arr[2] = 3;

    printf("Arreglo inicial (3 elementos): %d, %d, %d\n", arr[0], arr[1],
arr[2]);

    // Redimensionar a 5 elementos
    int *nuevo_arr = (int *)realloc(arr, 5 * sizeof(int));

    if (nuevo_arr == NULL) {
        free(arr);
        return 1;
    }

    arr = nuevo_arr; // El puntero original ahora apunta al nuevo bloque
    arr[3] = 4;
    arr[4] = 5;

    printf("Arreglo redimensionado (5 elementos): %d, %d, %d, %d, %d\n",
           arr[0], arr[1], arr[2], arr[3], arr[4]);

    free(arr);

    return 0;
}
```

Ejercicio Práctico: Pedile al usuario que ingrese la cantidad de números que desea almacenar. Asigná memoria dinámicamente para ese número de enteros, leé los números y luego imprimilos.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int cantidad;
    int *numeros;

    printf("¿Cuántos números querés almacenar? ");
    scanf("%d", &cantidad);

    // Asignación dinámica
    numeros = (int *)malloc(cantidad * sizeof(int));

    if (numeros == NULL) {
        printf("Error de asignación de memoria.\n");
        return 1;
    }

    // Lectura de datos
    for (int i = 0; i < cantidad; i++) {
        printf("Ingresá el número %d: ", i + 1);
        scanf("%d", &numeros[i]);
    }

    // Impresión de datos
    printf("\nLos números ingresados son: ");
    for (int i = 0; i < cantidad; i++) {
        printf("%d ", numeros[i]);
    }
    printf("\n");

    free(numeros);

    return 0;
}
```

Módulo 11: Manejo de Archivos

11.1. Archivos de Texto (ASCII)

Teoría: El manejo de archivos permite que los programas almacenen datos de forma persistente. En C, se utiliza la librería `stdio.h` y el tipo de dato `FILE` para manejar archivos.

Pasos para el Manejo de Archivos:

1. **Abrir:** Usar `FILE *fopen(const char *filename, const char *mode)` para abrir el archivo.
2. **Leer/Escribir:** Usar funciones como `fprintf`, `fscanf`, `fputc`, `fgetc`.
3. **Cerrar:** Usar `int fclose(FILE *stream)` para liberar el recurso.

Modo	Descripción
`"r"`	Lectura. El archivo debe existir.
`"w"`	Escritura. Crea el archivo si no existe, o lo trunca (borra el contenido) si existe.
`"a"`	Anexar (Append). Crea el archivo si no existe, o agrega datos al final si existe.
`"r+"`	Lectura y escritura. El archivo debe existir.

Código de Ejemplo Explicado (Escritura y Lectura):

```c
#include <stdio.h>

int main() {
    FILE *archivo;
    char linea[100];

    // 1. Escritura (modo "w")
    archivo = fopen("datos.txt", "w");
    if (archivo == NULL) {
        printf("Error al abrir el archivo para escritura.\n");
        return 1;
    }

    fprintf(archivo, "Hola, Basdonax AI.\n");
    fprintf(archivo, "Este es un ejemplo de escritura en C.\n");
    fclose(archivo);
    printf("Datos escritos en datos.txt\n");

    // 2. Lectura (modo "r")
    archivo = fopen("datos.txt", "r");
    if (archivo == NULL) {
        printf("Error al abrir el archivo para lectura.\n");
        return 1;
    }

    printf("\nContenido del archivo:\n");
    // Leer línea por línea hasta el final del archivo (EOF)
    while (fgets(linea, 100, archivo) != NULL) {
        printf("%s", linea);
    }

    fclose(archivo);

    return 0;
}
```

Ejercicio Práctico: Escribí un programa que pida al usuario su nombre y su edad, y guarde esa información en un archivo llamado `registro.txt` en modo de anexar (`"a"`).

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    FILE *archivo;
    char nombre[50];
    int edad;

    printf("Ingresá tu nombre: ");
    scanf("%s", nombre);
    printf("Ingresá tu edad: ");
    scanf("%d", &edad);

    // Abrir en modo anexar ("a")
    archivo = fopen("registro.txt", "a");

    if (archivo == NULL) {
        printf("Error al abrir el archivo.\n");
        return 1;
    }

    // Escribir en el archivo
    fprintf(archivo, "Nombre: %s, Edad: %d\n", nombre, edad);

    fclose(archivo);
    printf("Registro guardado exitosamente.\n");

    return 0;
}
```

11.2. Archivos Binarios

Teoría: Los **archivos binarios** almacenan los datos exactamente como están representados en la memoria de la computadora (bytes sin formato). Son más eficientes en espacio y velocidad que los archivos de texto, y son ideales para guardar estructuras de datos complejas.

Modos de Apertura: Se agrega una `b` al modo de texto (ej: `"wb"`, `"rb"`, `"ab"`).

Funciones Clave:

- `size_t fwrite(const void *ptr, size_t size, size_t count, FILE *stream)`: Escribe `count` elementos, cada uno de tamaño `size` bytes, desde la ubicación de memoria `ptr` al archivo.
- `size_t fread(void *ptr, size_t size, size_t count, FILE *stream)`: Lee `count` elementos, cada uno de tamaño `size` bytes, desde el archivo a la ubicación de memoria `ptr`.

Código de Ejemplo Explicado (Estructura en Binario):

```c
#include <stdio.h>
#include <string.h>

struct Persona {
    char nombre[20];
    int edad;
};

int main() {
    struct Persona p1 = {"Carlos", 35};
    struct Persona p_leida;
    FILE *archivo;

    // 1. Escritura Binaria ("wb")
    archivo = fopen("persona.bin", "wb");
    if (archivo == NULL) return 1;

    // Escribir la estructura completa
    fwrite(&p1, sizeof(struct Persona), 1, archivo);
    fclose(archivo);
    printf("Estructura escrita en persona.bin\n");

    // 2. Lectura Binaria ("rb")
    archivo = fopen("persona.bin", "rb");
    if (archivo == NULL) return 1;

    // Leer la estructura completa
    fread(&p_leida, sizeof(struct Persona), 1, archivo);
    fclose(archivo);

    printf("Datos leídos:\n");
    printf("Nombre: %s, Edad: %d\n", p_leida.nombre, p_leida.edad);

    return 0;
}
```

Ejercicio Práctico: Usá `fseek` para posicionar el puntero de lectura/escritura al inicio de un archivo binario. (Conceptual: ¿Qué función usarías para saber la posición actual del puntero?)

Espacio de Práctica:

> *Respuesta:*

Solución Esperada:

> *La función que usaría para saber la posición actual del puntero de archivo es* `long ftell(FILE *stream)`*.*
>
> ***Ejemplo de*** `fseek`***:***

```
> #include <stdio.h>
>
> int main() {
>     FILE *archivo = fopen("test.bin", "wb+"); // Abrir para
lectura/escritura binaria
>     if (archivo == NULL) return 1;
>
>     // Escribir algo
>     fputc('A', archivo);
>     fputc('B', archivo);
>
>     // Mover el puntero al inicio (offset 0 desde el inicio)
>     fseek(archivo, 0, SEEK_SET);
>
>     // Leer el primer carácter
>     printf("Carácter leído después de fseek: %c\n", fgetc(archivo)); //
Imprime 'A'
>
>     fclose(archivo);
>     return 0;
> }
> ```

11.3. Manejo de Errores en Archivos

Teoría:
Es fundamental verificar si las operaciones de archivo fueron exitosas.

*   **Verificación de Apertura:** `fopen` devuelve `NULL` si el archivo no se pudo abrir (ej: no existe en modo `"r"`, o no hay permisos).
*   **Fin de Archivo (EOF):** La función `int feof(FILE *stream)` devuelve un valor distinto de cero si se alcanzó el final del archivo.
*   **Indicador de Error:** La función `int ferror(FILE *stream)` devuelve un valor distinto de cero si ocurrió un error de lectura/escritura.
*   **Información de Error:** Las funciones `perror()` y la variable global `errno` (de `errno.h`) proporcionan detalles sobre el último error del sistema.

Código de Ejemplo Explicado (`perror`):
```c
#include <stdio.h>
#include <errno.h> // Para la variable errno
```
```

```c
#include <string.h> // Para strerror

int main() {
    FILE *archivo;

    // Intentar abrir un archivo en un modo que requiere permisos de escritura en una ruta protegida
    // (o simplemente un archivo que no existe en modo "r")
    archivo = fopen("archivo_inexistente.txt", "r");

    if (archivo == NULL) {
        // Imprime un mensaje de error descriptivo basado en el valor de errno
        perror("Error al intentar abrir el archivo");

        // También se puede usar strerror(errno)
        printf("Detalle del error: %s\n", strerror(errno));

        return 1;
    }

    // Si se abre correctamente, se cierra
    fclose(archivo);

    return 0;
}
```

Ejercicio Práctico: Escribí un fragmento de código que intente leer un archivo llamado `prueba.txt`. Si `fopen` devuelve `NULL`, usá `perror` para informar al usuario.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>
#include <errno.h>

int main() {
    FILE *archivo = fopen("prueba.txt", "r");

    if (archivo == NULL) {
        perror("Error de Apertura");
        return 1;
    }

    printf("Archivo abierto exitosamente.\n");
    fclose(archivo);

    return 0;
}
```

Resumen del Módulo Avanzado

Concepto Clave	Descripción
Punteros	Variables que almacenan direcciones de memoria. Operadores: `&` (dirección) y `*` (valor).
Aritmética	`p + 1` avanza el puntero el tamaño del tipo de dato.
Arreglos	El nombre del arreglo es un puntero constante al primer elemento.
Memoria Dinámica	Asignación en tiempo de ejecución (Heap). Funciones: `malloc`, `calloc`, `realloc`.
Liberación	`free()` es obligatorio para evitar fugas de memoria. Asignar `NULL` después.
Archivos	Persistencia de datos. Se usa el tipo `FILE *`. Modos: `"r"`, `"w"`, `"a"`, etc.
I/O Archivos	Texto: `fprintf`, `fscanf`. Binario: `fwrite`, `fread`.
Errores	Verificar `fopen` contra `NULL`. Usar `perror` para mensajes de error del sistema.

Evaluación Parcial: Nivel Avanzado

Instrucciones: Resolvé los siguientes ejercicios en un solo programa.

1. **Intercambio con Punteros:** Escribí una función llamada `intercambiar` que reciba las **direcciones** de dos enteros y use punteros para intercambiar sus valores.
2. **Arreglo Dinámico de Flotantes:** Pedile al usuario la cantidad de números flotantes que desea ingresar. Asigná memoria dinámicamente con `malloc`, leé los valores y calculá su suma. Liberá la memoria al finalizar.
3. **Contador de Líneas en Archivo:** Escribí un programa que lea el archivo `datos.txt` (creado en el ejemplo 11.1) y cuente cuántas líneas contiene.

Solución de la Evaluación Parcial:

```c
#include <stdio.h>
#include <stdlib.h>

// Función para Ejercicio 1: Intercambio con Punteros
void intercambiar(int *a, int *b) {
    int temp = *a; // Guarda el valor apuntado por a
    *a = *b;       // Asigna el valor apuntado por b al valor apuntado por a
    *b = temp;     // Asigna el valor temporal (original de a) al valor
apuntado por b
}

int main() {
    // --- Ejercicio 1: Intercambio con Punteros ---
    printf("--- Ejercicio 1 ---\n");
    int x = 10, y = 20;
    printf("Antes del intercambio: x = %d, y = %d\n", x, y);
    intercambiar(&x, &y); // Pasamos las direcciones
    printf("Después del intercambio: x = %d, y = %d\n", x, y);

    // --- Ejercicio 2: Arreglo Dinámico de Flotantes ---
    printf("\n--- Ejercicio 2 ---\n");
    int cantidad;
    float *numeros;
    float suma = 0.0;

    printf("Cantidad de flotantes a ingresar: ");
    scanf("%d", &cantidad);

    numeros = (float *)malloc(cantidad * sizeof(float));

    if (numeros == NULL) {
        printf("Error de memoria.\n");
        return 1;
    }

    for (int i = 0; i < cantidad; i++) {
        printf("Ingresá flotante %d: ", i + 1);
        scanf("%f", &numeros[i]);
        suma += numeros[i];
    }

    printf("La suma de los %.2f números es: %.2f\n", (float)cantidad, suma);
    free(numeros);

    // --- Ejercicio 3: Contador de Líneas en Archivo ---
```

```c
    printf("\n--- Ejercicio 3 ---\n");
    FILE *archivo;
    int contador_lineas = 0;
    char caracter;

    // Creamos el archivo de prueba si no existe (para que el ejemplo funcione)
    archivo = fopen("datos.txt", "w");
    if (archivo != NULL) {
        fprintf(archivo, "Linea 1\nLinea 2\nLinea 3\n");
        fclose(archivo);
    }

    archivo = fopen("datos.txt", "r");

    if (archivo == NULL) {
        perror("Error al abrir datos.txt");
        return 1;
    }

    // Contar líneas: cada '\n' es una nueva línea
    while ((caracter = fgetc(archivo)) != EOF) {
        if (caracter == '\n') {
            contador_lineas++;
        }
    }

    // Si el archivo no está vacío y no termina en '\n', se cuenta la última línea
    if (contador_lineas == 0 && ferror(archivo) == 0) {
        contador_lineas = 1;
    } else if (contador_lineas > 0 && caracter != '\n') {
        contador_lineas++;
    }

    printf("El archivo datos.txt tiene %d líneas.\n", contador_lineas);

    fclose(archivo);

    return 0;
}
```

Nivel 4: Experto - Estructuras de Datos Complejas y Optimización

¡Felicidades! Llegaste al nivel Experto. Aquí aplicarás todo lo aprendido sobre punteros y memoria dinámica para construir estructuras de datos complejas, manipular la memoria a bajo nivel y optimizar el rendimiento de tu código. Este conocimiento es fundamental para el desarrollo de sistemas operativos, compiladores y aplicaciones de alto rendimiento.

Módulo 12: Estructuras de Datos Dinámicas

12.1. Listas Enlazadas Simples

Teoría: Una **Lista Enlazada Simple** es una colección lineal de elementos de datos llamados **nodos**. A diferencia de los arreglos, los nodos no están almacenados en posiciones de memoria contiguas. Cada nodo contiene dos partes:

1. El **dato** que almacena.
2. Un **puntero** al siguiente nodo de la secuencia.

La lista se mantiene mediante un puntero principal llamado `cabeza` o `inicio`, que apunta al primer nodo. El último nodo apunta a `NULL`.

Estructura del Nodo:

```
struct Nodo {
    int dato;
    struct Nodo *siguiente; // Puntero al siguiente nodo
};
```

Código de Ejemplo Explicado (Inserción al Inicio):

```c
#include <stdio.h>
#include <stdlib.h>

// Definición de la estructura del nodo
struct Nodo {
    int dato;
    struct Nodo *siguiente;
};

// Función para insertar un nuevo nodo al inicio de la lista
void insertar_inicio(struct Nodo **cabeza, int nuevo_dato) {
    // 1. Asignar memoria para el nuevo nodo
    struct Nodo *nuevo_nodo = (struct Nodo *)malloc(sizeof(struct Nodo));

    if (nuevo_nodo == NULL) {
        printf("Error de memoria.\n");
        return;
    }

    // 2. Asignar el dato
    nuevo_nodo->dato = nuevo_dato;

    // 3. El puntero 'siguiente' del nuevo nodo apunta a la cabeza actual
    nuevo_nodo->siguiente = *cabeza;

    // 4. La cabeza de la lista ahora es el nuevo nodo
    *cabeza = nuevo_nodo;
}

// Función para imprimir la lista
void imprimir_lista(struct Nodo *nodo) {
    while (nodo != NULL) {
        printf("%d -> ", nodo->dato);
        nodo = nodo->siguiente;
    }
    printf("NULL\n");
}

int main() {
    struct Nodo *cabeza = NULL; // La lista comienza vacía

    insertar_inicio(&cabeza, 10); // Lista: 10 -> NULL
    insertar_inicio(&cabeza, 20); // Lista: 20 -> 10 -> NULL
    insertar_inicio(&cabeza, 30); // Lista: 30 -> 20 -> 10 -> NULL
```

```
    printf("Lista enlazada: ");
    imprimir_lista(cabeza);

    // Nota: La liberación de memoria (free) es crucial en producción.
    // Se debe recorrer la lista y liberar cada nodo.

    return 0;
}
```

Notas del Instructor:

> ***Operador Flecha (`->`):*** *Cuando trabajás con punteros a estructuras, usá el operador flecha (`->`) para acceder a sus miembros. `puntero->miembro` es una abreviatura de `(*puntero).miembro`.*

Ejercicio Práctico: Escribí una función llamada `contar_nodos` que reciba la cabeza de una lista enlazada y devuelva el número total de nodos que contiene.

Espacio de Práctica:

```
// Escribí tu código aquí
```

Solución Esperada:

```c
// ... (Definición de struct Nodo) ...

int contar_nodos(struct Nodo *cabeza) {
    int contador = 0;
    struct Nodo *actual = cabeza;

    while (actual != NULL) {
        contador++;
        actual = actual->siguiente;
    }

    return contador;
}

// ... (main con lista creada) ...

int main() {
    // ... (Creación de lista con 3 nodos) ...

    printf("La lista tiene %d nodos.\n", contar_nodos(cabeza)); // 3

    // ... (Liberación de memoria) ...
    return 0;
}
```

12.2. Listas Doblemente Enlazadas y Circulares

Teoría:

- **Lista Doblemente Enlazada:** Cada nodo tiene dos punteros: uno al nodo **siguiente** y otro al nodo **anterior**. Esto permite recorrer la lista en ambas direcciones.
- **Lista Circular:** El puntero `siguiente` del último nodo apunta al primer nodo (`cabeza`), creando un ciclo.

Estructura del Nodo Doblemente Enlazado:

```c
struct NodoDoble {
    int dato;
    struct NodoDoble *anterior; // Puntero al nodo anterior
    struct NodoDoble *siguiente; // Puntero al nodo siguiente
};
```

Ventajas y Desventajas:

Tipo de Lista	Ventajas	Desventajas
Simple	Más simple de implementar, menor uso de memoria por nodo.	Solo se puede recorrer hacia adelante.
Doble	Recorrido bidireccional, eliminación más eficiente.	Mayor complejidad de implementación, mayor uso de memoria por nodo.
Circular	Acceso rápido al inicio desde el final, ideal para *buffers* circulares.	Bucle infinito si no se maneja correctamente el punto de parada.

Ejercicio Práctico (Conceptual): En una lista doblemente enlazada, ¿cuál es el valor del puntero `anterior` del primer nodo y el valor del puntero `siguiente` del último nodo?

Espacio de Práctica:

> *Respuesta:*

Solución Esperada:

> *En una lista doblemente enlazada **no circular**:*
>
> - *El puntero `anterior` del primer nodo (`cabeza`) debe ser `NULL`.*
> - *El puntero `siguiente` del último nodo debe ser `NULL`.*

12.3. Pilas (Stacks) y Colas (Queues)

Teoría: Las Pilas y Colas son estructuras de datos abstractas que imponen restricciones sobre cómo se pueden agregar y eliminar elementos.

- **Pila (Stack):** Sigue el principio **LIFO** (*Last In, First Out* - Último en Entrar, Primero en Salir). Las operaciones principales son:
  - `push`: Agregar un elemento a la cima.
  - `pop`: Eliminar y devolver el elemento de la cima.
- **Cola (Queue):** Sigue el principio **FIFO** (*First In, First Out* - Primero en Entrar, Primero en Salir). Las operaciones principales son:
  - `enqueue`: Agregar un elemento al final.
  - `dequeue`: Eliminar y devolver el elemento del frente.

Implementación: Ambas estructuras se pueden implementar eficientemente usando **Listas Enlazadas** o **Arreglos Dinámicos**.

Código de Ejemplo Explicado (Implementación de Pila con Lista Enlazada):

```c
#include <stdio.h>
#include <stdlib.h>

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

// Push (Insertar al inicio de la lista)
void push(struct PilaNodo **cima, int dato) {
    struct PilaNodo *nuevo_nodo = (struct PilaNodo *)malloc(sizeof(struct
PilaNodo));
    if (nuevo_nodo == NULL) return;

    nuevo_nodo->dato = dato;
    nuevo_nodo->siguiente = *cima;
    *cima = nuevo_nodo;
}

// Pop (Eliminar del inicio de la lista)
int pop(struct PilaNodo **cima) {
    if (*cima == NULL) {
        printf("Error: Pila vacía.\n");
        return -1;
    }

    struct PilaNodo *temp = *cima;
    int dato_sacado = temp->dato;

    *cima = temp->siguiente; // Mover la cima al siguiente nodo
    free(temp); // Liberar el nodo eliminado

    return dato_sacado;
}

int main() {
    struct PilaNodo *pila = NULL;

    push(&pila, 10);
    push(&pila, 20);
    push(&pila, 30); // Cima: 30

    printf("Pop: %d\n", pop(&pila)); // 30
    printf("Pop: %d\n", pop(&pila)); // 20
```

```
    return 0;
}
```

Ejercicio Práctico: Explicá cómo se implementaría la operación `dequeue` (sacar de la cola) si la cola se implementa usando una lista enlazada simple.

Espacio de Práctica:

> *Respuesta:*

Solución Esperada:

> *Para implementar `dequeue` en una cola basada en una lista enlazada simple, se debe mantener un puntero al **frente** y otro al **final** de la lista. La operación `dequeue` consiste en:*
>
> 1. *Devolver el dato del nodo apuntado por el puntero `frente`.*
> 2. *Mover el puntero `frente` al nodo `siguiente`.*
> 3. *Liberar la memoria del nodo que acaba de ser eliminado.*
>
> *Esto asegura que el elemento que entró primero (el que está en el `frente`) sea el primero en salir.*

Módulo 13: Programación a Bajo Nivel

13.1. Operaciones a Nivel de Bit

Teoría: Las operaciones a nivel de bit permiten manipular los bits individuales de un entero. Son esenciales para la programación de sistemas embebidos, *drivers* y para optimizar el uso de memoria.

Operador	Nombre	Descripción
`&`	AND a nivel de bit	Pone a 1 solo si ambos bits son 1.
`|`	OR a nivel de bit	Pone a 1 si al menos un bit es 1.
`^`	XOR a nivel de bit	Pone a 1 si los bits son diferentes.
`~`	NOT a nivel de bit	Invierte todos los bits (complemento a uno).
`<<`	Desplazamiento a la izquierda	Multiplica por 2 por cada posición desplazada.
`>>`	Desplazamiento a la derecha	Divide por 2 por cada posición desplazada.

Código de Ejemplo Explicado (Uso de Máscaras de Bits):

```c
#include <stdio.h>

// Definición de banderas (flags) usando potencias de 2
#define FLAG_LECTURA  0b0001 // 1
#define FLAG_ESCRITURA 0b0010 // 2
#define FLAG_EJECUCION 0b0100 // 4

int main() {
    unsigned char permisos = 0b0000; // Permisos iniciales (0)

    // 1. Activar un bit (OR)
    permisos = permisos | FLAG_LECTURA; // Activa el bit de lectura
    permisos |= FLAG_ESCRITURA; // Forma abreviada

    printf("Permisos (decimal): %u\n", permisos); // 3 (1 + 2)

    // 2. Verificar si un bit está activo (AND)
    if (permisos & FLAG_LECTURA) {
        printf("El permiso de LECTURA está activo.\n");
    }

    // 3. Desactivar un bit (AND con NOT)
    permisos &= ~FLAG_LECTURA; // Desactiva el bit de lectura

    // 4. Desplazamiento
    int a = 5; // 00000101
    int b = a << 2; // Desplaza 2 posiciones a la izquierda: 00010100 (20)
    printf("5 << 2 = %d\n", b);

    return 0;
}
```

Ejercicio Práctico: Usá el operador XOR (`^`) para intercambiar los valores de dos variables enteras (`a` y `b`) sin usar una variable temporal.

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>

int main() {
    int a = 10; // 1010
    int b = 5;  // 0101

    printf("Antes: a = %d, b = %d\n", a, b);

    a = a ^ b; // a = 1010 ^ 0101 = 1111 (15)
    b = a ^ b; // b = 1111 ^ 0101 = 1010 (10) -> b ahora tiene el valor original de a
    a = a ^ b; // a = 1111 ^ 1010 = 0101 (5)  -> a ahora tiene el valor original de b

    printf("Después: a = %d, b = %d\n", a, b);

    return 0;
}
```

13.2. Manipulación de Memoria (`memcpy`, `memset`)

Teoría: Las funciones de manipulación de memoria de la librería `string.h` (aunque manipulan memoria, no solo cadenas) son cruciales para mover y rellenar bloques de memoria de forma eficiente.

- `void *memcpy(void *destino, const void *origen, size_t n)`: Copia `n` bytes desde el área de memoria `origen` al área de memoria `destino`. Es la forma más rápida de copiar bloques de datos.
- `void *memset(void *s, int c, size_t n)`: Rellena los primeros `n` bytes del área de memoria apuntada por `s` con el valor constante `c` (interpretado como un `unsigned char`). Se usa comúnmente para inicializar bloques de memoria a 0.

Código de Ejemplo Explicado:

```c
#include <stdio.h>
#include <string.h> // Para memcpy y memset

int main() {
    int origen[] = {1, 2, 3, 4, 5};
    int destino[5];

    // 1. Inicializar el arreglo destino a 0 usando memset
    // Rellenar 5 * sizeof(int) bytes con el valor 0
    memset(destino, 0, 5 * sizeof(int));

    printf("Destino inicializado: %d\n", destino[0]); // 0

    // 2. Copiar 5 enteros (20 bytes) de origen a destino
    memcpy(destino, origen, 5 * sizeof(int));

    printf("Destino después de memcpy: %d, %d, %d\n", destino[0],
destino[1], destino[4]);

    return 0;
}
```

Ejercicio Práctico: Creá un arreglo de 10 enteros. Usá `memset` para inicializar solo los primeros 5 elementos a 0. Luego, imprimí el valor del elemento en el índice 6 (que no debería ser 0).

Espacio de Práctica:

```c
// Escribí tu código aquí
```

Solución Esperada:

```c
#include <stdio.h>
#include <string.h>

int main() {
    int arreglo[10];

    // Inicializar los primeros 5 enteros (5 * 4 bytes = 20 bytes) a 0
    memset(arreglo, 0, 5 * sizeof(int));

    // Inicializar el resto manualmente para demostrar que no fueron
afectados
    for (int i = 5; i < 10; i++) {
        arreglo[i] = 99;
    }

    printf("Elemento en índice 0 (memset): %d\n", arreglo[0]);
    printf("Elemento en índice 4 (memset): %d\n", arreglo[4]);
    printf("Elemento en índice 5 (manual): %d\n", arreglo[5]);
    printf("Elemento en índice 6 (manual): %d\n", arreglo[6]);

    return 0;
}
```

13.3. Programación con `void *` (Punteros Genéricos)

Teoría: Un puntero de tipo `void *` es un **puntero genérico** que puede apuntar a cualquier tipo de dato. Se utiliza para escribir funciones de utilidad que pueden operar con datos de cualquier tipo, como `malloc`, `memcpy` o las funciones de ordenamiento `qsort`.

Restricciones:

- No se puede desreferenciar directamente un `void *`. Primero debe ser **convertido** (*cast*) al tipo de puntero correcto.
- No se puede realizar aritmética de punteros con un `void *` sin convertirlo, ya que el compilador no sabe qué tamaño de dato debe saltar.

Código de Ejemplo Explicado (Función Genérica de Intercambio):

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Función genérica para intercambiar dos bloques de memoria de cualquier tipo
void swap_generico(void *a, void *b, size_t size) {
    // 1. Asignar un buffer temporal en el stack
    void *temp = malloc(size);
    if (temp == NULL) return;

    // 2. Copiar el contenido de 'a' a 'temp'
    memcpy(temp, a, size);

    // 3. Copiar el contenido de 'b' a 'a'
    memcpy(a, b, size);

    // 4. Copiar el contenido de 'temp' a 'b'
    memcpy(b, temp, size);

    free(temp);
}

int main() {
    int x = 10, y = 20;
    float f1 = 3.14, f2 = 2.71;

    printf("Antes (int): x=%d, y=%d\n", x, y);
    swap_generico(&x, &y, sizeof(int));
    printf("Después (int): x=%d, y=%d\n", x, y);

    printf("Antes (float): f1=%.2f, f2=%.2f\n", f1, f2);
    swap_generico(&f1, &f2, sizeof(float));
    printf("Después (float): f1=%.2f, f2=%.2f\n", f1, f2);

    return 0;
}
```

Ejercicio Práctico: Explicá por qué la función `memcpy` recibe punteros de tipo `void *` en lugar de un tipo específico como `char *` o `int *`.

Espacio de Práctica:

> *Respuesta:*

Solución Esperada:

> *La función `memcpy` recibe punteros de tipo `void *` porque su propósito es manipular bloques de memoria de forma **agnóstica al tipo de dato**. `memcpy` solo necesita saber dónde comienza el bloque de memoria (`origen` y `destino`) y cuántos bytes debe copiar (`size_t n`). Al usar `void *`, la función se vuelve genérica y puede copiar cualquier tipo de dato (enteros, flotantes, estructuras, arreglos, etc.) sin necesidad de sobrecargar la función o requerir conversiones de tipo específicas por parte del usuario.*

Módulo 14: Optimización y Debugging

14.1. Técnicas de Optimización

Teoría: La optimización en C se divide en dos áreas: la que realiza el **programador** (elección de algoritmos y estructuras de datos) y la que realiza el **compilador**.

Optimización del Compilador (GCC): El compilador GCC ofrece diferentes niveles de optimización que se activan con la bandera `-O` (mayúscula O):

- `-O0`: Sin optimización (por defecto). Ideal para *debugging*.
- `-O1`: Optimización básica (reduce el tamaño del código y el tiempo de ejecución).
- `-O2`: Optimización más agresiva (la más recomendada para la mayoría de las aplicaciones de producción).
- `-O3`: Optimización muy agresiva (puede aumentar el tiempo de compilación).
- `-Os`: Optimiza para el tamaño del código (útil para sistemas embebidos).

Optimización del Código (Programador):

- **Algoritmos:** Elegir un algoritmo eficiente (ej: *quicksort* en lugar de *bubblesort*).
- **Uso de `const`:** Usar `const` para variables que no cambian permite al compilador realizar optimizaciones de caché.
- **Evitar I/O dentro de bucles:** Las operaciones de entrada/salida (`printf`, `scanf`) son lentas.

- **Evitar cálculos redundantes:** Calcular una expresión una sola vez fuera de un bucle si su valor no cambia.

Ejercicio Práctico (Conceptual): Si estás desarrollando un *firmware* para un microcontrolador con memoria muy limitada, ¿qué nivel de optimización de GCC (`-O0`, `-O2`, `-Os`) sería el más adecuado y por qué?

Espacio de Práctica:

> *Respuesta:*

Solución Esperada:

> *El nivel de optimización más adecuado sería* ***`-Os`*** *. Este nivel está específicamente diseñado para optimizar el* ***tamaño del código*** *binario, lo cual es crucial en sistemas embebidos y microcontroladores donde la memoria de programa (Flash) es un recurso muy limitado.*

14.2. Debugging con GDB

Teoría: GDB (GNU Debugger) es la herramienta estándar para depurar programas C/C++ en entornos Unix/Linux. Permite ejecutar el programa paso a paso, inspeccionar variables y modificar el flujo de ejecución.

Pasos para usar GDB:

1. **Compilar con información de *debugging*:** Debés compilar tu código con la bandera `-g` (ej: `gcc -g mi_programa.c -o mi_programa`).
2. **Iniciar GDB:** `gdb ./mi_programa`
3. **Comandos Clave:**
   - `break [función/línea]`: Establece un punto de interrupción.
   - `run`: Ejecuta el programa hasta el primer *breakpoint*.
   - `next`: Ejecuta la línea actual y avanza a la siguiente (salta la ejecución de funciones).
   - `step`: Ejecuta la línea actual y entra en la función si es una llamada.
   - `print [variable]`: Muestra el valor de una variable.
   - `continue`: Continúa la ejecución hasta el siguiente *breakpoint* o el final.

- `quit`: Sale de GDB.

Ejercicio Práctico (Conceptual): Estás depurando un bucle que se ejecuta 1000 veces. Querés detener la ejecución solo cuando la variable `i` del bucle alcance el valor 500. ¿Qué comando de GDB usarías?

Espacio de Práctica:

> *Respuesta:*

Solución Esperada:

> *Usarías un breakpoint condicional:*
>
> `break [nombre_archivo]:[número_línea] if i == 500`
>
> *O, si ya estás en la función:*
>
> `break if i == 500`

14.3. Manejo de Errores Avanzado

Teoría: El manejo de errores en C a nivel experto se centra en la robustez y la capacidad de recuperación del sistema.

- **Códigos de Retorno:** Las funciones deben devolver un código de error (generalmente un entero distinto de 0) para indicar un fallo.
- **`errno` y `perror`:** Ya vistos en el Nivel Avanzado, son la forma estándar de manejar errores del sistema operativo (ej: fallos de I/O, memoria).
- **Aserciones (`assert.h`):** La macro `assert(expresión)` se usa para verificar suposiciones del programador. Si la expresión es falsa, el programa termina inmediatamente con un mensaje de error que indica el archivo y la línea. **Solo se usa para *debugging***, ya que se deshabilita en compilaciones de producción (si se define `NDEBUG`).

Código de Ejemplo Explicado (`assert`):

```c
#include <stdio.h>
#include <assert.h> // Necesario para assert

// Función que no debería recibir un puntero NULL
int procesar_puntero(int *p) {
    // Si 'p' es NULL, el programa termina aquí con un mensaje de error
    assert(p != NULL);

    // Si la aserción pasa, el código continúa
    return *p * 2;
}

int main() {
    int valor = 10;
    int *p1 = &valor;
    int *p2 = NULL;

    printf("Resultado 1: %d\n", procesar_puntero(p1));

    // Esto causará que el programa aborte con un mensaje de error
    printf("Resultado 2: %d\n", procesar_puntero(p2));

    return 0;
}
```

Ejercicio Práctico: Explicá la diferencia de uso entre `assert()` y una verificación manual de error con `if (p == NULL) { return ERROR; }`.

Espacio de Práctica:

> *Respuesta:*

Solución Esperada:

> *La diferencia principal es su **propósito y ciclo de vida**:*
>
> - `assert()` *se usa para verificar **condiciones internas** que el programador asume que siempre serán verdaderas (ej: la entrada de una función nunca será* `NULL`*). Está diseñado para **detectar errores de programación** durante el desarrollo y debugging. Se **deshabilita** automáticamente en el código de producción (al compilar con* `-DNDEBUG`*), por lo que no debe usarse para validar entradas de usuario o errores esperados.*

> - *La verificación manual (`if (p == NULL)`) se usa para manejar **errores esperados** (ej: fallo de `malloc`, archivo no encontrado, entrada de usuario inválida). Este código **permanece** en la versión de producción para garantizar la robustez y la capacidad de recuperación del programa.*

Resumen del Módulo Experto

Concepto Clave	Descripción
Listas Enlazadas	Estructura de datos dinámica con nodos que contienen dato y puntero al siguiente.
Pila/Cola	LIFO (`push` / `pop`) y FIFO (`enqueue` / `dequeue`). Implementadas con listas o arreglos.
Bits	Operadores `&`, `
Memoria	`memcpy` (copia bloques), `memset` (rellena bloques). Usan `void *`.
`void *`	Puntero genérico. Debe ser convertido antes de desreferenciar o usar aritmética.
Optimización	Niveles de compilador (`-O2`, `-Os`) y buenas prácticas de código.
Debugging	GDB. Comandos clave: `break`, `run`, `next`, `step`, `print`.
Errores	`assert()` para *debugging* (se deshabilita en producción). Códigos de retorno para errores de *runtime*.

Evaluación Parcial: Nivel Experto

Instrucciones: Resolvé los siguientes ejercicios en un solo programa.

1. **Manipulación de Bits:** Escribí una función que reciba un entero y un número de bit (0 a 31). La función debe devolver el entero con ese bit **activado** (puesto a 1), sin afectar los demás bits.

2. **Liberación de Lista:** Escribí una función llamada `liberar_lista` que recorra una lista enlazada simple (usando la estructura `struct Nodo` del 12.1) y libere la memoria de cada nodo.
3. **Inicialización de Estructura:** Definí una estructura `Config` con 5 miembros `int`. Asigná memoria dinámicamente para una variable `Config` y usá `memset` para inicializar todos sus miembros a 0. Imprimí el valor del primer miembro para verificar.

Solución de la Evaluación Parcial:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Estructura para Ejercicio 2
struct Nodo {
    int dato;
    struct Nodo *siguiente;
};

// Estructura para Ejercicio 3
struct Config {
    int a, b, c, d, e;
};

// Función para Ejercicio 1: Activar Bit
int activar_bit(int num, int bit_pos) {
    // 1 << bit_pos crea una máscara con un 1 en la posición deseada
    // El operador OR (|) activa el bit sin afectar los demás
    return num | (1 << bit_pos);
}

// Función para Ejercicio 2: Liberar Lista
void liberar_lista(struct Nodo *cabeza) {
    struct Nodo *actual = cabeza;
    struct Nodo *siguiente;

    while (actual != NULL) {
        siguiente = actual->siguiente; // Guardar el puntero al siguiente
        free(actual); // Liberar el nodo actual
        actual = siguiente; // Mover al siguiente
    }
    printf("Lista liberada.\n");
}

int main() {
    // --- Ejercicio 1: Manipulación de Bits ---
    printf("--- Ejercicio 1 ---\n");
    int num = 8; // Binario: 1000 (Bit 3 activo)
    int pos = 1; // Queremos activar el Bit 1
    int nuevo_num = activar_bit(num, pos); // Resultado: 1010 (10)
    printf("Original: %d (Bit 3 activo)\n", num);
    printf("Nuevo (Bit %d activado): %d\n", pos, nuevo_num);

    // --- Ejercicio 2: Liberación de Lista ---
```

```
    printf("\n--- Ejercicio 2 ---\n");
    struct Nodo *cabeza = (struct Nodo *)malloc(sizeof(struct Nodo));
    if (cabeza != NULL) {
        cabeza->dato = 1;
        cabeza->siguiente = (struct Nodo *)malloc(sizeof(struct Nodo));
        if (cabeza->siguiente != NULL) {
            cabeza->siguiente->dato = 2;
            cabeza->siguiente->siguiente = NULL;
        }
    }
    liberar_lista(cabeza);

    // --- Ejercicio 3: Inicialización de Estructura ---
    printf("\n--- Ejercicio 3 ---\n");
    struct Config *p_config = (struct Config *)malloc(sizeof(struct
Config));

    if (p_config == NULL) {
        printf("Error de memoria.\n");
        return 1;
    }

    // Inicializar todos los bytes de la estructura a 0
    memset(p_config, 0, sizeof(struct Config));

    printf("Tamaño de struct Config: %zu bytes\n", sizeof(struct Config));
    printf("Primer miembro (a) inicializado a: %d\n", p_config->a);

    free(p_config);

    return 0;
}
```

Proyectos Integradores Finales

Estos proyectos están diseñados para que apliques y combines los conocimientos adquiridos en los cuatro niveles del manual. Cada proyecto simula una aplicación real y requiere la integración de múltiples conceptos, desde la sintaxis básica hasta el manejo de memoria dinámica y archivos.

Proyecto 1: Gestor de Tareas (To-Do List)

Objetivo: Crear un gestor de tareas simple que permita agregar, listar y eliminar tareas.

Conceptos Integrados:

- **Experto:** Listas Enlazadas Simples (Estructuras, Punteros, Memoria Dinámica `malloc` / `free`).
- **Avanzado:** Manejo de Archivos de Texto (persistencia de datos).
- **Intermedio:** Cadenas de Caracteres (`strcpy` , `strlen`).
- **Básico:** Funciones, Control de Flujo (`switch` , `while`).

Código Comentado:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Definición de la estructura del nodo de la lista enlazada
struct Tarea {
    int id;
    char descripcion[100];
    struct Tarea *siguiente;
};

// Puntero global a la cabeza de la lista
struct Tarea *cabeza = NULL;
int id_contador = 1; // Contador para asignar IDs únicos

// --- Funciones de Utilidad ---

// 1. Cargar tareas desde el archivo
void cargar_tareas() {
    FILE *archivo = fopen("tareas.txt", "r");
    if (archivo == NULL) {
        // El archivo no existe o no se puede abrir, se asume lista vacía
        return;
    }

    int id;
    char descripcion[100];

    // Leer el archivo línea por línea
    while (fscanf(archivo, "%d %[^\n]", &id, descripcion) == 2) {
        // Crear un nuevo nodo para la tarea
        struct Tarea *nueva_tarea = (struct Tarea *)malloc(sizeof(struct
Tarea));
        if (nueva_tarea == NULL) {
            perror("Error de memoria al cargar tarea");
            break;
        }

        nueva_tarea->id = id;
        strcpy(nueva_tarea->descripcion, descripcion);
        nueva_tarea->siguiente = NULL;

        // Insertar al final de la lista (para mantener el orden)
        if (cabeza == NULL) {
            cabeza = nueva_tarea;
```

```
        } else {
            struct Tarea *actual = cabeza;
            while (actual->siguiente != NULL) {
                actual = actual->siguiente;
            }
            actual->siguiente = nueva_tarea;
        }

        // Actualizar el contador de ID
        if (id >= id_contador) {
            id_contador = id + 1;
        }
    }

    fclose(archivo);
}

// 2. Guardar tareas en el archivo
void guardar_tareas() {
    FILE *archivo = fopen("tareas.txt", "w");
    if (archivo == NULL) {
        perror("Error al guardar el archivo");
        return;
    }

    struct Tarea *actual = cabeza;
    while (actual != NULL) {
        // Escribir ID y descripción en una línea
        fprintf(archivo, "%d %s\n", actual->id, actual->descripcion);
        actual = actual->siguiente;
    }

    fclose(archivo);
}

// 3. Liberar toda la memoria de la lista
void liberar_lista() {
    struct Tarea *actual = cabeza;
    struct Tarea *siguiente;
    while (actual != NULL) {
        siguiente = actual->siguiente;
        free(actual);
        actual = siguiente;
    }
    cabeza = NULL;
}
```

```c
// --- Funciones del Gestor ---

// 4. Agregar una nueva tarea
void agregar_tarea() {
    char desc[100];
    printf("Descripción de la nueva tarea: ");
    // Leer la línea completa, incluyendo espacios
    scanf(" %[^\n]", desc);

    struct Tarea *nueva_tarea = (struct Tarea *)malloc(sizeof(struct
Tarea));
    if (nueva_tarea == NULL) {
        perror("Error de memoria");
        return;
    }

    nueva_tarea->id = id_contador++;
    strcpy(nueva_tarea->descripcion, desc);
    nueva_tarea->siguiente = NULL;

    // Insertar al final
    if (cabeza == NULL) {
        cabeza = nueva_tarea;
    } else {
        struct Tarea *actual = cabeza;
        while (actual->siguiente != NULL) {
            actual = actual->siguiente;
        }
        actual->siguiente = nueva_tarea;
    }

    printf("Tarea %d agregada.\n", nueva_tarea->id);
}

// 5. Listar todas las tareas
void listar_tareas() {
    if (cabeza == NULL) {
        printf("No hay tareas pendientes.\n");
        return;
    }

    struct Tarea *actual = cabeza;
    printf("\n--- Lista de Tareas ---\n");
    while (actual != NULL) {
        printf("[%d] %s\n", actual->id, actual->descripcion);
```

```
        actual = actual->siguiente;
    }
    printf("-----------------------\n");
}

// 6. Eliminar una tarea por ID
void eliminar_tarea() {
    int id_eliminar;
    printf("Ingresá el ID de la tarea a eliminar: ");
    scanf("%d", &id_eliminar);

    struct Tarea *actual = cabeza;
    struct Tarea *anterior = NULL;

    // Caso 1: Eliminar la cabeza
    if (actual != NULL && actual->id == id_eliminar) {
        cabeza = actual->siguiente;
        free(actual);
        printf("Tarea %d eliminada.\n", id_eliminar);
        return;
    }

    // Caso 2: Buscar y eliminar en el medio/final
    while (actual != NULL && actual->id != id_eliminar) {
        anterior = actual;
        actual = actual->siguiente;
    }

    // Si no se encontró la tarea
    if (actual == NULL) {
        printf("Error: Tarea con ID %d no encontrada.\n", id_eliminar);
        return;
    }

    // Desvincular el nodo de la lista
    anterior->siguiente = actual->siguiente;
    free(actual);
    printf("Tarea %d eliminada.\n", id_eliminar);
}

// --- Función Principal ---

int main() {
    int opcion;

    // Cargar tareas al inicio del programa
```

```
    cargar_tareas();

    do {
        printf("\n--- Gestor de Tareas ---\n");
        printf("1. Agregar Tarea\n");
        printf("2. Listar Tareas\n");
        printf("3. Eliminar Tarea\n");
        printf("4. Salir y Guardar\n");
        printf("Seleccioná una opción: ");
        scanf("%d", &opcion);

        switch (opcion) {
            case 1:
                agregar_tarea();
                break;
            case 2:
                listar_tareas();
                break;
            case 3:
                eliminar_tarea();
                break;
            case 4:
                printf("Guardando tareas y saliendo...\n");
                guardar_tareas();
                break;
            default:
                printf("Opción no válida. Intentá de nuevo.\n");
        }
    } while (opcion != 4);

    // Liberar la memoria antes de terminar
    liberar_lista();

    return 0;
}
```

Proyecto 2: Sistema de Registro de Estudiantes

Objetivo: Crear un sistema que permita registrar estudiantes (nombre, edad, nota) y guardar/cargar la información en un archivo binario.

Conceptos Integrados:

- **Avanzado:** Manejo de Archivos Binarios (`fwrite` , `fread` , `fseek`).
- **Intermedio:** Estructuras (`struct`), Arreglos de Estructuras.
- **Básico:** Funciones, Control de Flujo.

Código Comentado:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Definición de la estructura del estudiante
struct Estudiante {
    int id;
    char nombre[50];
    int edad;
    float nota_final;
};

// Nombre del archivo binario
#define NOMBRE_ARCHIVO "estudiantes.bin"

// --- Funciones de Archivo ---

// 1. Agregar un estudiante al final del archivo
void agregar_estudiante() {
    struct Estudiante nuevo_estudiante;
    FILE *archivo;

    printf("\n--- Nuevo Estudiante ---\n");
    printf("ID: ");
    scanf("%d", &nuevo_estudiante.id);
    printf("Nombre: ");
    scanf(" %49[^\n]", nuevo_estudiante.nombre); // Leer hasta 49 caracteres
o salto de línea
    printf("Edad: ");
    scanf("%d", &nuevo_estudiante.edad);
    printf("Nota Final: ");
    scanf("%f", &nuevo_estudiante.nota_final);

    // Abrir en modo anexar binario ("ab")
    archivo = fopen(NOMBRE_ARCHIVO, "ab");
    if (archivo == NULL) {
        perror("Error al abrir el archivo para escritura");
        return;
    }

    // Escribir la estructura completa en el archivo
    fwrite(&nuevo_estudiante, sizeof(struct Estudiante), 1, archivo);

    fclose(archivo);
    printf("Estudiante %s registrado exitosamente.\n",
```

```
nuevo_estudiante.nombre);
}

// 2. Listar todos los estudiantes
void listar_estudiantes() {
    struct Estudiante estudiante_leido;
    FILE *archivo;

    // Abrir en modo lectura binaria ("rb")
    archivo = fopen(NOMBRE_ARCHIVO, "rb");
    if (archivo == NULL) {
        printf("No hay estudiantes registrados.\n");
        return;
    }

    printf("\n--- Lista de Estudiantes ---\n");
    printf("ID | Nombre | Edad | Nota\n");
    printf("---|--------|------|------\n");

    // Leer estructuras una por una hasta el final del archivo
    while (fread(&estudiante_leido, sizeof(struct Estudiante), 1, archivo)
== 1) {
        printf("%-2d | %-10s | %-4d | %.2f\n",
               estudiante_leido.id,
               estudiante_leido.nombre,
               estudiante_leido.edad,
               estudiante_leido.nota_final);
    }

    fclose(archivo);
}

// 3. Buscar un estudiante por ID
void buscar_estudiante() {
    int id_buscado;
    printf("Ingresá el ID del estudiante a buscar: ");
    scanf("%d", &id_buscado);

    struct Estudiante estudiante_leido;
    FILE *archivo = fopen(NOMBRE_ARCHIVO, "rb");
    if (archivo == NULL) {
        printf("No hay estudiantes registrados.\n");
        return;
    }

    int encontrado = 0;
```

```
    // Recorrer el archivo
    while (fread(&estudiante_leido, sizeof(struct Estudiante), 1, archivo)
== 1) {
        if (estudiante_leido.id == id_buscado) {
            printf("\n--- Estudiante Encontrado ---\n");
            printf("Nombre: %s\n", estudiante_leido.nombre);
            printf("Edad: %d\n", estudiante_leido.edad);
            printf("Nota Final: %.2f\n", estudiante_leido.nota_final);
            encontrado = 1;
            break; // Salir del bucle una vez encontrado
        }
    }

    if (!encontrado) {
        printf("Estudiante con ID %d no encontrado.\n", id_buscado);
    }

    fclose(archivo);
}

// --- Función Principal ---

int main() {
    int opcion;

    do {
        printf("\n--- Sistema de Registro ---\n");
        printf("1. Agregar Estudiante\n");
        printf("2. Listar Estudiantes\n");
        printf("3. Buscar Estudiante por ID\n");
        printf("4. Salir\n");
        printf("Seleccioná una opción: ");
        scanf("%d", &opcion);

        switch (opcion) {
            case 1:
                agregar_estudiante();
                break;
            case 2:
                listar_estudiantes();
                break;
            case 3:
                buscar_estudiante();
                break;
            case 4:
                printf("Saliendo del sistema...\n");
```

```
                break;
            default:
                printf("Opción no válida. Intentá de nuevo.\n");
        }
    } while (opcion != 4);

    return 0;
}
```

Proyecto 3: Juego de Adivinanza con Números

Objetivo: Crear un juego simple donde el usuario debe adivinar un número aleatorio en la menor cantidad de intentos posible.

Conceptos Integrados:

- **Básico/Intermedio:** Control de Flujo (`while` , `if`), Funciones, Entrada/Salida.
- **Avanzado:** Librería `stdlib.h` (`rand` , `srand`) y `time.h` (para la semilla aleatoria).

Código Comentado:

```c
#include <stdio.h>
#include <stdlib.h> // Para rand() y srand()
#include <time.h>   // Para time()

// Definición de constantes
#define MIN_NUMERO 1
#define MAX_NUMERO 100

// Función para generar el número secreto
int generar_numero_secreto() {
    // Usar la hora actual como semilla para el generador de números
aleatorios
    srand(time(NULL));

    // Generar un número entre MIN_NUMERO y MAX_NUMERO
    return (rand() % (MAX_NUMERO - MIN_NUMERO + 1)) + MIN_NUMERO;
}

// Función principal del juego
void jugar_adivinanza() {
    int numero_secreto = generar_numero_secreto();
    int intento;
    int contador_intentos = 0;

    printf("\n--- ¡Adiviná el Número! ---\n");
    printf("Estoy pensando en un número entre %d y %d.\n", MIN_NUMERO,
MAX_NUMERO);

    do {
        printf("Ingresá tu intento: ");
        // Usar scanf para leer el intento del usuario
        if (scanf("%d", &intento) != 1) {
            printf("Entrada inválida. Por favor, ingresá un número.\n");
            // Limpiar el buffer de entrada para evitar bucles infinitos
            while (getchar() != '\n');
            continue;
        }

        contador_intentos++;

        // Estructuras condicionales para dar pistas
        if (intento < numero_secreto) {
            printf("¡Demasiado bajo! Intentá con un número mayor.\n");
        } else if (intento > numero_secreto) {
            printf("¡Demasiado alto! Intentá con un número menor.\n");
```

```
        } else {
            // Condición de salida del bucle
            printf("\n¡FELICITACIONES! Adivinaste el número %d.\n",
numero_secreto);
            printf("Te tomó %d intentos.\n", contador_intentos);
        }

    } while (intento != numero_secreto);
}

int main() {
    jugar_adivinanza();

    return 0;
}
```

Proyecto 4: Calculadora de Matrices

Objetivo: Implementar la suma y resta de dos matrices de 3x3.

Conceptos Integrados:

- **Intermedio:** Arreglos Multidimensionales (Matrices).
- **Básico:** Funciones, Bucles anidados (`for`), Control de Flujo.

Código Comentado:

```c
#include <stdio.h>

#define FILAS 3
#define COLUMNAS 3

// 1. Función para leer los elementos de una matriz
void leer_matriz(int matriz[FILAS][COLUMNAS], const char *nombre) {
    printf("\nIngresá los elementos de la matriz %s (%dx%d):\n", nombre,
FILAS, COLUMNAS);
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            printf("[%d][%d]: ", i, j);
            scanf("%d", &matriz[i][j]);
        }
    }
}

// 2. Función para imprimir una matriz
void imprimir_matriz(int matriz[FILAS][COLUMNAS]) {
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            printf("%4d", matriz[i][j]); // %4d para alinear
        }
        printf("\n");
    }
}

// 3. Función para sumar dos matrices
void sumar_matrices(int A[FILAS][COLUMNAS], int B[FILAS][COLUMNAS], int
C[FILAS][COLUMNAS]) {
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            C[i][j] = A[i][j] + B[i][j];
        }
    }
}

// 4. Función para restar dos matrices
void restar_matrices(int A[FILAS][COLUMNAS], int B[FILAS][COLUMNAS], int
C[FILAS][COLUMNAS]) {
    for (int i = 0; i < FILAS; i++) {
        for (int j = 0; j < COLUMNAS; j++) {
            C[i][j] = A[i][j] - B[i][j];
        }
    }
```

```c
}

int main() {
    int matrizA[FILAS][COLUMNAS];
    int matrizB[FILAS][COLUMNAS];
    int matrizSuma[FILAS][COLUMNAS];
    int matrizResta[FILAS][COLUMNAS];

    // Lectura de las matrices
    leer_matriz(matrizA, "A");
    leer_matriz(matrizB, "B");

    // Suma
    sumar_matrices(matrizA, matrizB, matrizSuma);

    // Resta
    restar_matrices(matrizA, matrizB, matrizResta);

    // Resultados
    printf("\n--- Matriz A ---\n");
    imprimir_matriz(matrizA);

    printf("\n--- Matriz B ---\n");
    imprimir_matriz(matrizB);

    printf("\n--- Resultado de la Suma (A + B) ---\n");
    imprimir_matriz(matrizSuma);

    printf("\n--- Resultado de la Resta (A - B) ---\n");
    imprimir_matriz(matrizResta);

    return 0;
}
```