Principios de Programación

Primeros Pasos en C - Semana 2

Mariano Zunino

Tecnólogo en Informática

Agenda de Hoy

Estructura Básica:

- Estructura de un programa C
- Función main()
- Directivas del preprocesador
- Comentarios en código

Entrada y Salida:

- Función printf()
- Función scanf()
- Formato de salida
- De pseudocódigo a código C
- Primer programa completo

¿Cuándo y dónde surgió C?

Contexto histórico:

- Finales de los años 60 y comienzos de los 70 (1969–1972)
- Laboratorios Bell (Bell Labs), en Estados Unidos
- Principal autor: Dennis Ritchie
- Objetivo: crear un lenguaje para desarrollar sistemas operativos (como Unix)

Conexión con Unix:

- Unix nació como un sistema operativo experimental
- Al principio estaba escrito en ensamblador
- Reescribirlo en C permitió que fuera más fácil de mantener y portar a otras máquinas

¿Por qué se creó C?

Problema inicial:

- El ensamblador es muy difícil de leer y mantener
- Cada arquitectura de hardware tiene su propio ensamblador
- Portar un sistema operativo a otra máquina requería reescribir mucho código

Respuesta de C:

- Ofrecer un lenguaje más legible que el ensamblador
- Mantener un control fino sobre el hardware y la memoria
- Permitir escribir sistemas operativos y software de bajo nivel
- Lograr portabilidad: el mismo código C puede compilarse en distintas arquitecturas

C: ¿alto nivel o bajo nivel?

Como lenguaje de alto nivel:

- Usa estructuras, funciones y tipos de datos más cercanos a la forma de pensar humana
- El código es mucho más legible que el ensamblador
- Permite expresar algoritmos de forma clara

Como lenguaje de bajo nivel:

- Permite controlar cómo se usan la memoria y el hardware
- Pocas cosas suceden «por detrás» del programa
- Es posible escribir código muy eficiente (si se escribe con cuidado)

Conclusión:

- A menudo se lo llama **lenguaje de sistemas**:
  - Suficientemente cercano al hardware
  - Pero con abstracciones suficientes para ser legible

¿En qué se usa C?

- Kernels de sistemas operativos (Linux, partes de Windows, BSD, etc.)
- Controladores de dispositivos (drivers)
- Software embebido (routers, microcontroladores, etc.)
- Bibliotecas de rendimiento crítico (compresión, bases de datos, motores de juegos)

Idea central (inspirada en «Effective C»):

- C fue diseñado para escribir código **eficiente**, **portátil** y **predecible**
- Obliga a pensar en cómo se representan los datos en memoria
- Bien usado, permite construir bloques sólidos sobre los que otros lenguajes se apoyan

¿Por qué aprender C hoy?

Pregunta típica del estudiante:

- «Yo quiero usar TypeScript, Python, Ruby… para qué me sirve C?»

C como base de otros lenguajes:

- Muchos intérpretes y máquinas virtuales (Python, Ruby, JavaScript) están escritos en C
- Estructuras como arreglos, strings y diccionarios se implementan (muy a menudo) en C
- Entender C ayuda a entender qué pasa «por debajo» cuando usamos lenguajes modernos

Ventajas para la formación:

- Desarrollar una intuición fuerte sobre memoria, eficiencia y coste de las operaciones
- Lenguaje natural para cursos de Estructuras de Datos y Algoritmos (EDA)
- Facilita aprender otros lenguajes: reconoces patrones y entiendes mejor sus límites

Mensaje importante:

- No se busca que sean programadores profesionales de C
- Usamos C como **laboratorio** para aprender a programar y comprender los fundamentos

Recordando: De Pseudocódigo a C

En la semana anterior:

- Se trabajó con el diseño de algoritmos en pseudocódigo
- Se revisó cómo estructurar soluciones paso a paso
- Ejemplo: algoritmo para sumar dos números

Hoy aprenderemos:

- Cómo traducir ese pseudocódigo a código C
- La estructura básica de un programa C
- Cómo hacer que la computadora ejecute las instrucciones

Ejemplo de conexión:

Pseudocódigo (Semana 1):

Sumar dos números

```
1 Alg: Sumar dos números
2 Vars: entero numero1, numero2, resultado
3 inicio
4     print «Ingrese primer número:»
5     read numero1
6     print «Ingrese segundo número:»
7     read numero2
8     resultado = numero1 + numero2
9     print «La suma es:», resultado
```

Hoy veremos cómo convertir esto en código C ejecutable.

«Todo programa C tiene una estructura básica que debemos seguir»

Componentes principales:

- **Directivas del preprocesador**: `#include`
- **Función main()**: punto de entrada
- **Instrucciones**: código que se ejecuta
- **Comentarios**: explicaciones en el código

Ejemplo mínimo:

```c
#include <stdio.h>

int main() {
    int numero;  // Declaramos una variable entera
    return 0;
}
```

¿Qué es una variable? (intuición):

- Cuando se declara `int numero;` el programa reserva una pequeña región de memoria
- Podemos imaginarla como una **cajita** en memoria
- Esa cajita tiene:
  - Un **nombre**: `numero`
  - Un **tipo**: `int` (entero)
  - Un **valor** que podremos guardar allí
- En el estándar de C, a esa región de memoria se le suele llamar un **object** (objeto)

Bloques `{}` como grupos de instrucciones

Recordemos el ejemplo:

```c
int main() {
    int numero;
    return 0;
}
```

¿Qué es un bloque?

- Todo lo que está entre `{` y `}` forma un **bloque**
- En este caso, el bloque es el **cuerpo** de la función `main`
- Dentro del bloque se escriben las instrucciones que se ejecutan

Más adelante veremos:

- Bloques en `if`, `while`, `for`, etc.
- Las variables que declaramos **viven** dentro de un bloque
- Por ahora, es útil quedarse con la idea de que `{` y `}` agrupan instrucciones que van juntas

Desglose del ejemplo:

- `#include <stdio.h>`
  - **Qué es**: Incluye código de la biblioteca estándar de entrada/salida
  - **Por qué lo necesitamos**: Para usar funciones como `printf()` y `scanf()`
  - **Dónde va**: Siempre al inicio del archivo, antes de cualquier código
  - **Cómo funciona**: El compilador «copia» el contenido de ese archivo aquí
- `int main()`
  - **Qué es**: La función principal del programa
  - **Por qué se llama main**: Es el punto de entrada estándar en C
  - **Qué significa int**: Indica que la función retorna un número entero
  - **Cuántas puede haber**: Exactamente una por programa
- `return 0;`
  - **Qué hace**: Indica que el programa terminó correctamente
  - **Por qué 0**: Es la convención del sistema operativo (0 = éxito)
  - **Dónde va**: Siempre al final de `main()`
  - **Qué ocurre si no se incluye**: El programa puede funcionar, pero es buena práctica incluirlo

¿Qué es main()?

- Es la función que se ejecuta primero
- Todo programa C debe tener exactamente una función main
- Es el punto de entrada del programa

Sintaxis:

```c
int main() {
    // Código aquí
    return 0;
}
```

Partes importantes:

- `int`: tipo de retorno (número entero)
- `main`: nombre de la función
- `()`: lista de parámetros (vacía en este caso)
- `{ }`: cuerpo de la función
- `return 0;`: valor de retorno

Variantes comunes:

```c
// Forma estándar (RECOMENDADA)
int main() {
    return 0;
}

// Con void (NO RECOMENDADA)
void main() {
    // No retorna valor
    // Puede causar problemas en algunos sistemas
}
```

Recomendación: Usar siempre `int main()` y retornar 0 al final. Esto indica al sistema operativo que el programa terminó correctamente. Es la forma estándar y más portable.

«Los include permiten usar código que alguien ya escribió»

La idea: No reinventar la rueda

- Muchas funciones son muy comunes (mostrar texto, leer números, etc.)
- Alguien ya escribió ese código y lo compartió
- Simplemente se incluye y se usa
- Esto ahorra tiempo y evita errores

Analogía: Es como usar una herramienta que alguien ya construyó. No necesitamos construir un martillo desde cero cada vez que queremos clavar un clavo.

Ejemplo:

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

Esto incluye código que contiene funciones como:

- `printf()`: para mostrar texto
- `scanf()`: para leer datos
- Y muchas otras funciones útiles

¿Cómo Funciona #include?

¿Cómo funciona?

- `#include <stdio.h>` le dice al compilador: «Incluye aquí todo el código del archivo stdio.h»
- Ese archivo contiene las definiciones de funciones como `printf`
- Sin el `#include`, no podríamos usar `printf`
- Con el `#include`, podemos usar todas las funciones de esa biblioteca

Directiva más común:

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

- `#include`: comando para incluir código
- `<stdio.h>`: archivo con funciones de entrada/salida
- `< >`: indica que es una biblioteca estándar del sistema

Otras bibliotecas útiles:

```
#include <math.h>   // Funciones matemáticas (sqrt, pow, etc.)
#include <stdlib.h> // Utilidades generales
```

¿Qué son las directivas?

- Instrucciones que se procesan antes de compilar
- Empiezan con el símbolo #
- Se colocan siempre al inicio del archivo
- El preprocesador las procesa primero

Ejemplo práctico:

```c
#include <stdio.h>

int main() {
    printf("Hola\n");  // printf viene de stdio.h
    return 0;
}
```

Sin el `#include`, el compilador no sabría qué es `printf`.

Reglas importantes:

- Los archivos entre < > son bibliotecas estándar
- Se buscan en directorios del sistema
- Siempre van al inicio del archivo (antes de cualquier código)
- Una directiva por línea
- No llevan punto y coma al final

Bibliotecas comunes:

```
#include <stdio.h>  // Entrada/salida (printf, scanf)
#include <math.h>    // Matemáticas (sqrt, pow, sin, etc.)
#include <stdlib.h> // Utilidades (malloc, rand, etc.)
```

Nota: Por ahora, usaremos principalmente `stdio.h`. Las otras las veremos más adelante cuando las necesitemos.

Tipos de comentarios:

- **Comentario de línea**: `//`
  - Todo lo que sigue en la línea es comentario
  - Se ignora al compilar
- **Comentario de bloque**: `/* */`
  - Puede abarcar varias líneas
  - Todo entre `/*` y `*/` es comentario

Ejemplos:

```
// Este es un comentario de una línea

/* Este es un comentario
   que puede tener
   varias líneas */

int x = 5;  // Comentario al final de línea
```

Cuándo y Cómo Usar Comentarios

¿Cuándo usar comentarios?

- Explicar qué hace el código
- Documentar decisiones importantes
- Aclarar lógica compleja
- Desactivar código temporalmente

Buenas prácticas:

- No comentar lo obvio
- Explicar el «por qué», no el «qué»
- Mantener comentarios actualizados
- Usar comentarios para dividir secciones

Ejemplo:

```
// Calcular el promedio de tres números
int promedio = (a + b + c) / 3;
```

¿Qué es printf()?

- Función para mostrar texto en pantalla
- Equivalente a `print` en pseudocódigo
- Permite formatear la salida
- Requiere `#include <stdio.h>`

Sintaxis básica:

```c
printf("texto a mostrar");
```

Ejemplo simple:

```c
#include <stdio.h>

int main() {
    // printf() muestra texto en la pantalla
    // El texto va entre comillas dobles
    printf("Hola mundo!");
    return 0;
}
```

Antes de ejecutar, es útil pensar: ¿Qué se espera que muestre este programa?

Salida:

```
Hola mundo!
```

Conexión con pseudocódigo:

- Pseudocódigo: `print "Hola mundo"`
- C: `printf("Hola mundo");`

Observa cómo la traducción es casi directa: solo cambia la sintaxis.

Caracteres especiales (secuencias de escape):

- `\n`: nueva línea (salto de línea)
- `\t`: tabulación (tabulador)
- `\\`: muestra el carácter `\`
- `\"`: muestra comillas dobles

Ejemplos:

```
printf("Primera línea\n");
printf("Segunda línea\n");
printf("Tabulado\ttexto\n");
```

Salida:

```
Primera línea
Segunda línea
Tabulado    texto
```

Importante: Sin `\n`, todo se imprime en la misma línea. Usar `\n` al final para saltar de línea.

Ejercicio: Piensa antes de ejecutar ¿Qué mostraría este código?

```
printf("Primera");
printf("Segunda");
printf("Tercera");
```

Sin `\n`, todo se imprime seguido: `PrimeraSegundaTercera`

Imprimir variables: Para mostrar el valor de una variable, usamos especificadores de formato:

Especificadores comunes:

- `%d`: números enteros (int)
- `%f`: números decimales (float)
- `%c`: un carácter (char)
- `%s`: texto (string)

Ejemplo:

```
int edad = 20;
printf("Tengo %d años\n", edad);
```

Salida:

```
Tengo 20 años
```

Múltiples variables:

```c
int a = 5;
int b = 3;
int suma = a + b;

printf("La suma de %d y %d es %d\n", a, b, suma);
```

Salida:

```
La suma de 5 y 3 es 8
```

Importante:

- El orden de los especificadores debe coincidir con el orden de las variables
- Cada `%d` corresponde a una variable en orden

¿Qué es scanf()?

- Función para leer datos desde el teclado
- Equivalente a `read` en pseudocódigo
- Permite recibir valores del usuario
- Requiere `#include <stdio.h>`

Sintaxis básica:

```c
scanf("%d", &variable);
```

Ejemplo:

```c
int numero;
printf("Ingrese un número: ");
// scanf() lee un valor del teclado
// "%d" indica que esperamos un número entero
// &numero es necesario: el & le dice a scanf dónde guardar el valor
scanf("%d", &numero);
printf("Usted ingresó: %d\n", numero);
```

Importante - El símbolo &

- El `&` es necesario para `scanf()` (por ahora, acéptalo como regla)
- Sin el `&`, el programa puede fallar o comportarse de forma inesperada
- Más adelante entenderemos por qué (tiene que ver con direcciones de memoria)
- El especificador debe coincidir con el tipo de variable

Conexión con pseudocódigo:

- Pseudocódigo: `read numero`
- C: `scanf("%d", &numero);`

Especificadores para scanf()

Especificadores para scanf:

- `%d`: leer entero
- `%f`: leer decimal (float)
- `%c`: leer un carácter
- `%s`: leer texto (sin espacios)

Ejemplo completo:

```c
int edad;
float altura;

printf("Ingrese su edad: ");
// Leemos un entero: usamos %d
scanf("%d", &edad);

printf("Ingrese su altura: ");
// Leemos un decimal: usamos %f
scanf("%f", &altura);

// Mostramos ambos valores
// %.2f significa: mostrar con 2 decimales
printf("Edad: %d, Altura: %.2f\n", edad, altura);
```

Nota importante:

- `%d` para enteros (int)
- `%f` para decimales (float)
- `%.2f` muestra el número con exactamente 2 decimales
- El especificador debe coincidir con el tipo de variable

«Traduciendo el algoritmo de pseudocódigo a código C»

Recordemos el pseudocódigo:

Sumar dos números

```
1 Alg: Sumar dos números
2 Vars: entero numero1, numero2, resultado
3 inicio
4     print «Ingrese primer número:»
5     read numero1
6     print «Ingrese segundo número:»
7     read numero2
8     resultado = numero1 + numero2
9     print «La suma es:», resultado
```

Código C equivalente:

```c
// Incluimos la biblioteca necesaria para printf y scanf
#include <stdio.h>

int main() {
    // Declaramos las variables que necesitamos
    // En C debemos declarar las variables antes de usarlas
    int numero1, numero2, suma;

    // Pedimos y leemos el primer número
    printf("Ingrese primer número: ");
    scanf("%d", &numero1);

    // Pedimos y leemos el segundo número
    printf("Ingrese segundo número: ");
    scanf("%d", &numero2);

    // Calculamos la suma
    suma = numero1 + numero2;

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

    return 0;
}
```

Correspondencias pseudocódigo ↔ C:

- `print "texto"` ⟶ `printf("texto");`
- `read variable` ⟶ `scanf("%d", &variable);`
- `entero` ⟶ `int`
- `real` ⟶ `float`

Observación: La estructura es casi idéntica. El pseudocódigo ayudó a diseñar la solución antes de escribir código.

Explicación paso a paso:

1. Incluir biblioteca:

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

Incluye código con funciones de entrada/salida.

2. Función principal:

```
int main() {
```

Punto de entrada del programa.

3. Declarar variables:

```
int numero1, numero2, suma;
```

Reserva espacio para tres números enteros.

4. Pedir primer número:

```
printf("Ingrese primer número: ");
scanf("%d", &numero1);
```

Muestra mensaje y lee el valor ingresado.

5. Pedir segundo número:

```
printf("Ingrese segundo número: ");
scanf("%d", &numero2);
```

Repite el proceso para el segundo número.

6. Calcular suma:

```
suma = numero1 + numero2;
```

Realiza la operación aritmética.

7. Mostrar resultado:

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

Imprime el resultado en pantalla.

8. Terminar programa:

```
return 0;
```

Indica que el programa terminó correctamente.

Ejecución del programa:

```
Ingrese primer número: 5
Ingrese segundo número: 3
La suma es: 8
```

Flujo de ejecución:

1. El programa inicia en `main()`
2. Se ejecutan las instrucciones en orden secuencial
3. Cuando encuentra `scanf()`, el programa espera entrada del usuario
4. El usuario ingresa los valores
5. Se realizan los cálculos
6. Se muestra el resultado con `printf()`
7. El programa termina con `return 0`

Observación importante: La estructura del código C sigue exactamente la misma lógica que el pseudocódigo que diseñamos en la semana anterior.

Puntos clave:

- Cada instrucción se ejecuta en orden
- El programa espera cuando necesita datos del usuario
- Las variables almacenan los valores temporalmente
- El resultado se calcula y muestra automáticamente

Traza de ejecución: técnica que consiste en seguir el programa línea por línea, anotando los valores de las variables y la salida en cada paso.

Ejemplo: programa de suma, con entrada 5 y 3

Línea	numero1	numero2	suma	Salida
—	—	—	—	—
scanf	5	—	—	Ingrese primer número:
scanf	5	3	—	Ingrese segundo número:
suma = ...	5	3	8	—
printf	5	3	8	La suma es: 8

Uso: permite predecir el resultado sin ejecutar y ayuda a encontrar errores de lógica.

Otro Ejemplo: Calcular Promedio

Problema: Calcular el promedio de dos calificaciones

Recordemos el pseudocódigo (Semana 1):

Promedio de calificaciones

```
1 Alg: Promedio de calificaciones
2 Vars: real calificacion1, calificacion2, suma, promedio
3 inicio
4     read calificacion1
5     read calificacion2
6     suma = calificacion1 + calificacion2
7     promedio = suma / 2
8     print promedio
```

Código C:

```c
#include <stdio.h>

int main() {
    // Declaramos variables de tipo float (números decimales)
    float calif1, calif2, promedio;

    // Leemos la primera calificación
    printf("Ingrese primera calificación: ");
    scanf("%f", &calif1);

    // Leemos la segunda calificación
    printf("Ingrese segunda calificación: ");
    scanf("%f", &calif2);

    // Calculamos el promedio
    // Dividimos entre 2.0 (no 2) para obtener resultado decimal
    promedio = (calif1 + calif2) / 2.0;

    // Mostramos el resultado con 2 decimales
    printf("El promedio es: %.2f\n", promedio);

    return 0;
}
```

Puntos importantes:

- `real` en pseudocódigo $\longrightarrow$ `float` en C
- `%f` para leer y mostrar decimales
- `%.2f` muestra exactamente 2 decimales

¿Por qué dividir entre 2.0?

¿Por qué dividir entre 2.0 y no 2?

- Si dividimos entre `2` (entero), el resultado puede truncarse
- Dividir entre `2.0` (decimal) asegura que el resultado sea decimal
- Ejemplo: `5 / 2` = 2, pero `5 / 2.0` = 2.5

Ejecución:

```
Ingrese primera calificación: 85.5
Ingrese segunda calificación: 92.0
El promedio es: 88.75
```

Proceso paso a paso:

1. Escribir el código en un archivo `.c`
2. Compilar con GCC
3. Ejecutar el programa

Comandos básicos:

```
# Compilar
gcc programa.c -o programa

# Ejecutar
./programa
```

Explicación:

- `gcc`: compilador de C
- `programa.c`: archivo fuente
- `-o programa`: nombre del ejecutable
- `./programa`: ejecuta el programa

Flujo de trabajo:

1. Crear o abrir un archivo con extensión `.c` (p. ej. `suma.c`) en una carpeta conocida
2. Escribir el código en el editor y guardar
3. En la terminal, ubicarse en esa carpeta y ejecutar: `gcc suma.c -o suma`
4. Ejecutar el programa: `./suma` (en Linux/Mac) o `suma.exe` (en Windows)

Si se usa ZinjaI: El menú o botones «Compilar» y «Ejecutar» realizan estos pasos; conviene saber qué comando usa por debajo (`gcc`).

Ejemplo completo:

```
$ gcc suma.c -o suma
$ ./suma
Ingrese primer número: 5
Ingrese segundo número: 3
La suma es: 8
```

Si hay errores:

```
$ gcc programa.c -o programa
programa.c:5:5: error: expected ';' before 'printf'
```

El compilador indica el archivo, línea y tipo de error. Corregir el error y volver a compilar.

Si el programa no compila:

- Leer el **primer** mensaje del compilador (a veces los demás se deben al primero)
- Ir a la **línea** indicada en el mensaje
- Revisar: punto y coma, llaves, `&` en `scanf`, `#include <stdio.h>`, etc.

Si compila pero no hace lo esperado:

- Releer el enunciado del problema
- Hacer **traza a mano** (tabla de variables y salida)
- **Agregar printf** en puntos clave para ver el valor de las variables

Importante: Los errores son parte del proceso. La primera herramienta es leer con atención el mensaje y la línea indicada.

Error 1: Falta include

```c
int main() {
    printf("Hola");  // Error! printf no está definido
    return 0;
}
```

¿Por qué falla? Sin `#include <stdio.h>`, el compilador no sabe qué es `printf`. **Solución:** Agregar `#include <stdio.h>` al inicio

Error 2: Falta & en scanf

```c
int x;
scanf("%d", x);  // Error! Falta el &
```

¿Por qué falla? `scanf()` necesita saber dónde guardar el valor (dirección de memoria). **Solución:** Usar `scanf("%d", &x);` (con el símbolo &)

Error 3: Falta punto y coma

```c
int x = 5  // Error! Falta punto y coma
printf("%d", x);
```

¿Por qué falla? En C, cada instrucción debe terminar con `;` **Solución:** Agregar `;` al final: `int x = 5;`

Error 4: Especificador incorrecto

```c
int x = 5;
printf("%f", x);  // Error! x es int, pero usamos %f
```

¿Por qué falla? El especificador `%f` es para decimales, pero `x` es entero. **Solución:** Usar `%d` para enteros: `printf("%d", x);`

Error 5: Olvidar return

```c
int main() {
    printf("Hola");
    // Falta return 0;
}
```

¿Por qué es problema? `main()` debe retornar un `int`, pero no lo hace. **Solución:** Agregar `return 0;` al final

Consejo importante:

- Leer cuidadosamente los mensajes de error del compilador
- Indican el archivo, línea y tipo de error
- Los errores suelen aparecer en cascada: corregir el primero puede resolver varios

Problema: El programa compila pero el resultado es incorrecto (ej.: promedio mal calculado).

Código con error de lógica:

```
promedio = (num1 + num2) / 3;  // Error: son 2 números, debe ser / 2
```

Estrategia: Agregar `printf` para inspeccionar valores antes y después del cálculo:

```
printf("num1=%d num2=%d\n", num1, num2);
promedio = (num1 + num2) / 2.0;
printf("promedio=%.2f\n", promedio);
```

Así se verifica que los datos leídos son correctos y que el resultado coincide con lo esperado. Luego se pueden quitar los `printf` de depuración.

Organización del código:

- Siempre incluir la directiva `#include <stdio.h>` al inicio
- Usar función `main()` con `int` y `return 0`
- Declarar variables al inicio de la función
- Un espacio después de cada coma
- Indentar el código dentro de `main()`

Nombres de variables:

- Usar nombres descriptivos: `edad`, `suma`, `promedio`
- Evitar nombres muy cortos: `x`, `a`, `b` (excepto en ejemplos simples)
- Usar minúsculas para variables
- No usar espacios ni caracteres especiales

Legibilidad:

- Dejar líneas en blanco para separar secciones
- Comentar cuando sea necesario
- Alinear el código correctamente (indentación)
- Usar mensajes claros en `printf()`

Ejemplo de código bien organizado:

```c
#include <stdio.h>

int main() {
    int numero1, numero2, resultado;

    // Sección: Entrada de datos
    printf("Ingrese primer número: ");
    scanf("%d", &numero1);
    printf("Ingrese segundo número: ");
    scanf("%d", &numero2);

    // Sección: Procesamiento
    resultado = numero1 + numero2;

    // Sección: Salida de datos
    printf("El resultado es: %d\n", resultado);

    return 0;
}
```

Observación: Los comentarios ayudan a dividir el código en secciones lógicas:

- Entrada de datos
- Procesamiento
- Salida de datos

Esto hace el código más fácil de leer y entender.

Resumen de Conceptos Clave

Estructura básica:

- Todo programa C tiene `#include` y `main()`
- La función `main()` es el punto de entrada
- `return 0;` indica éxito
- Los comentarios ayudan a entender el código

Entrada y salida:

- `printf()`: mostrar texto y variables
- `scanf()`: leer valores del usuario
- Especificadores: `%d` (entero), `%f` (decimal)
- Usar `&` con variables en `scanf()`

Proceso de trabajo:

- Escribir código en archivo `.c`
- Compilar: `gcc programa.c -o programa`
- Ejecutar: `./programa`
- Corregir errores si los hay
- Probar con diferentes valores

Próxima clase:

- Variables y tipos de datos
- Identificadores y constantes
- Operadores aritméticos
- Más ejemplos prácticos

Problema: Escribir un programa que:

1. Pida al usuario su nombre
2. Pida su edad
3. Muestre un mensaje personalizado

Antes de programar, es útil pensar:

- ¿Qué variables se necesitan?
- ¿Qué tipo de datos son? (texto, número entero)
- ¿En qué orden se deben pedir los datos?
- ¿Cómo se mostrará el mensaje final?

Metodología recomendada:

1. Primero diseñar el algoritmo en pseudocódigo
2. Luego traducir a código C
3. Compilar y probar

Pista - Estructura básica:

```c
#include <stdio.h>

int main() {
    char nombre[50];  // Para almacenar texto
    int edad;         // Para almacenar un número entero

    // Escribir el código aquí

    return 0;
}
```

Nota sobre leer texto: Para leer texto, usar `scanf("%s", nombre);` (sin & porque es un arreglo de caracteres) Esto lo veremos en detalle más adelante.

Ejemplo de ejecución esperada:

```
Ingrese su nombre: Juan
Ingrese su edad: 20
Hola Juan, tiene 20 años
```

Pasos sugeridos:

1. Escribir el pseudocódigo primero
2. Identificar las variables necesarias
3. Traducir cada paso del pseudocódigo a C
4. Compilar: `gcc programa.c -o programa`
5. Ejecutar: `./programa`
6. Probar con diferentes valores

Es importante recordar:

- Planificar antes de programar
- Compilar antes de ejecutar
- Leer mensajes de error si aparecen
- Probar con diferentes valores de entrada

Tareas para esta semana:

- Completar el ejercicio propuesto
- Escribir y ejecutar al menos 3 programas diferentes
- Familiarizarse con el compilador GCC
- Practicar con printf y scanf
- Revisar errores comunes y sus soluciones

Herramientas:

- Editor de texto: ZinjaI (recomendado), VS Code, Vim, o similar
- Compilador: GCC (verificar que esté instalado)
- Terminal/Consola para ejecutar comandos

Bibliografía:

- «El lenguaje de programación C» - Kernighan & Ritchie
- «Cómo programar en C/C++» - Deitel & Deitel
- Documentación online de C

Próxima clase:

- Tipos de datos: int, float, char
- Declaración e inicialización de variables
- Operadores aritméticos básicos
- Más ejemplos y ejercicios prácticos