Principios de Programación

Introducción a la Programación - Semana 1

Mariano Zunino

Tecnólogo en Informática

Agenda de Hoy

Conceptos Fundamentales:

- ¿Qué es programar?
- Lenguajes de programación
- Algoritmos y su importancia
- Diagramas de flujo
- Pseudocódigo

Proceso de Desarrollo:

- De problema a solución
- De algoritmo a programa
- Proceso de compilación
- Primer acercamiento a C

«Programar es dar instrucciones precisas a una computadora para resolver un problema»

Analogía: Receta de Cocina

- Ingredientes (datos)
- Pasos ordenados (instrucciones)
- Resultado esperado (salida)
- Instrucciones claras y precisas

Ejemplo: Hacer un café

- Calentar agua
- Agregar café al filtro
- Verter agua caliente
- Esperar que gotee
- Servir en taza

En Programación:

- **Datos**: información que maneja el programa
- **Instrucciones**: pasos que ejecuta la computadora
- Deben ser claras y precisas
- Deben estar en orden correcto
- La computadora sigue las instrucciones exactamente

Características importantes:

- Precisión: cada paso debe ser exacto
- Secuencia: el orden importa
- Completitud: no pueden faltar pasos

Resolver problemas reales:

- Automatizar tareas repetitivas
- Procesar grandes cantidades de datos
- Crear herramientas útiles
- Desarrollar aplicaciones

Ejemplos cotidianos:

- Calculadora
- Agenda de contactos
- Sistema de calificaciones
- Juegos simples

Desarrollar habilidades:

- Pensamiento lógico y estructurado
- Resolución de problemas
- Atención al detalle
- Abstracción y modelado

Aplicaciones prácticas:

- Análisis de datos
- Automatización
- Desarrollo de software
- Base para cursos avanzados

¿Qué es un lenguaje de programación?

- Conjunto de reglas y símbolos que permiten comunicarse con la computadora
- Traduce ideas a instrucciones que la computadora puede ejecutar
- Es el «idioma» que se usa para dar órdenes a la máquina

¿Por qué necesitamos lenguajes?

- Las computadoras solo entienden 0s y 1s (binario)
- Escribir directamente en binario es muy difícil para los humanos
- Los lenguajes de programación son un «puente» entre nosotros y la máquina

Lenguaje de Máquina (Binario):

Ejemplo: 10110000 01100001

- Solo 0s y 1s
- Difícil de leer y escribir para humanos
- Directamente ejecutable por la computadora
- Cada instrucción es una secuencia específica de bits

Analogía: Es como si tuviéramos que comunicarnos con alguien usando solo señales de luz (encendido/apagado). Funciona, pero es muy tedioso.

Lenguajes de Alto Nivel:

- Más cercanos al lenguaje humano
- Más fáciles de leer y escribir
- Requieren traducción (compilación) para que la computadora los entienda
- Ejemplos: C, Python, Java, JavaScript

Ventajas:

- Código más legible y comprensible
- Desarrollo más rápido
- Menos propenso a errores
- Portabilidad entre diferentes sistemas

En este curso: Se utilizará C, un lenguaje de alto nivel que permite aprender los fundamentos de la programación de forma clara y estructurada.

Comparación práctica:

Sumar dos números:

```
// Alto nivel (C)
// Esto es fácil de leer y entender
int suma = a + b;
```

```
; Bajo nivel (Ensamblador)
; Esto es más difícil de leer
mov eax, a    ; Mover 'a' al registro eax
add eax, b    ; Sumar 'b' a eax
mov suma, eax ; Guardar resultado en 'suma'
```

Observación: Ambos hacen lo mismo, pero el código en C es mucho más fácil de entender. En este curso trabajaremos con C, que es un buen balance entre claridad y control.

«Un algoritmo es una secuencia ordenada y finita de pasos que resuelve un problema específico»

Características de un algoritmo:

- **Preciso**: cada paso está claramente definido
- **Determinista**: siempre produce el mismo resultado
- **Finito**: tiene un número limitado de pasos
- **Efectivo**: resuelve el problema planteado
- **Entrada**: recibe datos de entrada
- **Salida**: produce un resultado

Ejemplo: Sumar dos números

1. Leer primer número (a)
2. Leer segundo número (b)
3. Calcular suma = a + b
4. Mostrar suma

Ejemplo: Encontrar el mayor de dos números

1. Leer primer número (a)
2. Leer segundo número (b)
3. Si a > b entonces
   - resultado = a
4. Si no entonces
   - resultado = b
5. Mostrar resultado

Ejemplo: Calcular Promedio

Problema: Calcular el promedio de tres calificaciones

Algoritmo paso a paso:

1. Leer calificación 1
2. Leer calificación 2
3. Leer calificación 3
4. Calcular suma = calificación1 + calificación2 + calificación3
5. Calcular promedio = suma / 3
6. Mostrar promedio

Ejemplo con valores:

```
Entrada:
- Calificación 1: 85
- Calificación 2: 92
- Calificación 3: 78

Proceso:
- Suma = 85 + 92 + 78 = 255
- Promedio = 255 / 3 = 85

Salida:
- Promedio: 85
```

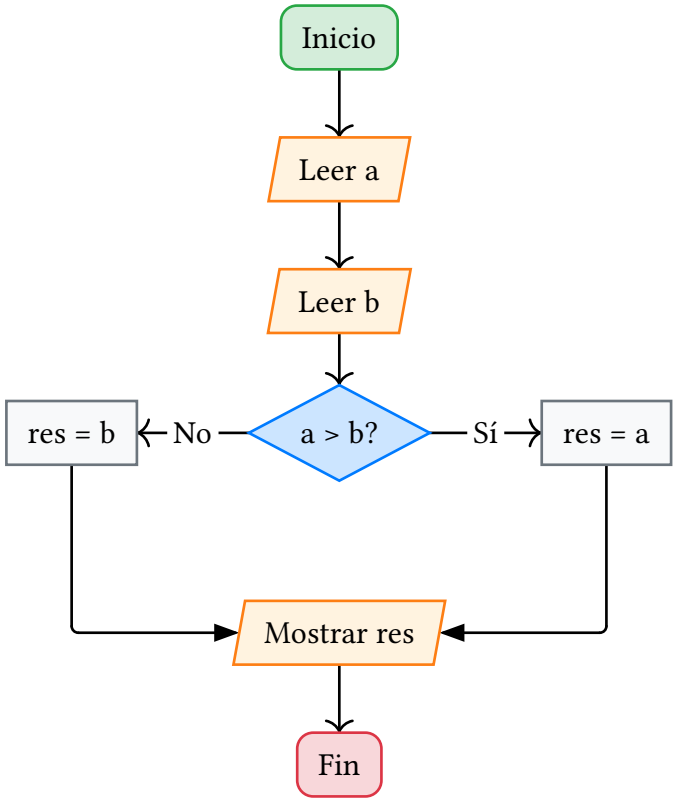
Un algoritmo bien definido puede ser implementado en cualquier lenguaje de programación

¿Qué es un diagrama de flujo?

- Representación gráfica de un algoritmo
- Muestra pasos y decisiones (ramas Sí/No)
- Ayuda a entender el **flujo** antes de escribir pseudocódigo o código

Símbolos comunes:

- Inicio / Fin (óvalo)
- Proceso (rectángulo)
- Decisión (rombo)
- Entrada / Salida (paralelogramo)
- Flechas (dirección del flujo)



¿Qué es el pseudocódigo?

- Lenguaje intermedio entre lenguaje natural y código
- No tiene sintaxis estricta
- Facilita el diseño de algoritmos
- Se puede traducir fácilmente a código real

Ventajas:

- No requiere conocer un lenguaje específico
- Enfocado en la lógica, no en la sintaxis
- Fácil de entender y modificar
- Útil para planificar antes de programar

Estructura típica:

Nombre del algoritmo

```
1 Alg: Nombre del algoritmo
2 Vars: tipo nombre_variable
3 inicio
4     read variable1
5     variable2 = variable1 * 2
6     print variable2
```

Palabras clave comunes:

- `read`, `print`
- `si ... entonces ... sino`
- `mientras ... hacer`
- `para ... hacer`

Ejemplo: Máximo de dos números

Problema: Encontrar el mayor de dos números

Pseudocódigo:

Máximo de dos números

```
1  Alg: Máximo de dos números
2  Input: dos números (a, b)
3  Output: el número mayor
4  inicio
5      read a
6      read b
7      si a > b entonces
8          resultado = a
9      sino
10         resultado = b
11     print resultado
```

Antes de continuar, conviene pensar: ¿Qué pasaría si a = 5 y b = 8? ¿Cuál sería el resultado?

Explicación paso a paso:

- Primero leemos los dos números (a y b)
- Luego comparamos: si a > b, el resultado es a
- Si no (es decir, si b >= a), el resultado es b
- Finalmente mostramos el resultado

Ejemplo de ejecución:

```
Entrada: a = 5, b = 8
Proceso:
  - ¿5 > 8? No
  - Entonces: resultado = 8
Salida: resultado = 8
```

Otro ejemplo:

```
Entrada: a = 10, b = 7
Proceso:
  - ¿10 > 7? Sí
  - Entonces: resultado = 10
Salida: resultado = 10
```

Ejemplo: Promedio en Pseudocódigo

Problema: Calcular el promedio de tres calificaciones

Pseudocódigo:

Promedio de calificaciones

```
1 Alg: Promedio de calificaciones
2 Vars: real calificacion1, calificacion2, calificacion3, suma, promedio
3 inicio
4     read calificacion1
5     read calificacion2
6     read calificacion3
7     suma = calificacion1 + calificacion2 + calificacion3
8     promedio = suma / 3
9     print promedio
```

Ejemplo con valores:

```
Entrada:
- Calificación 1: 85
- Calificación 2: 92
- Calificación 3: 78

Proceso:
- Suma = 85 + 92 + 78 = 255
- Promedio = 255 / 3 = 85

Salida:
- Promedio: 85
```

Observación: Este es el mismo algoritmo que se presentó antes, pero ahora expresado en pseudocódigo de forma más estructurada.

Ejemplo: Pseudocódigo para Suma

Problema: Sumar dos números y mostrar el resultado

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
```

Análisis del algoritmo paso a paso:

- **Línea 1-2**: Pedimos y leemos el primer número
- **Línea 3-4**: Pedimos y leemos el segundo número
- **Línea 5**: Calculamos la suma y la guardamos en `resultado`
- **Línea 6**: Mostramos el resultado

Ejemplo de ejecución:

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

Observaciones importantes:

- El algoritmo es claro y fácil de seguir
- Cada paso tiene un propósito específico
- **El orden de las instrucciones es importante**: no podemos calcular la suma antes de leer los números
- Este pseudocódigo puede implementarse en cualquier lenguaje de programación

Ejemplo: Par o Impar

Problema: Determinar si un número es par o impar

Pseudocódigo:

Par o Impar

```
1 Alg: Par o Impar
2 Vars: entero numero
3 inicio
4     print «Ingrese un número:»
5     read numero
6     si (numero % 2 == 0) entonces
7         print «El número es par»
8     sino
9         print «El número es impar»
```

- Si el usuario ingresa 8, ¿qué se mostrará?
- Si el usuario ingresa 7, ¿qué se mostrará?
- ¿Qué hace el operador %?

Análisis del algoritmo:

- Leemos el número del usuario
- Verificamos si el resto de dividir entre 2 es 0
- Si es 0, el número es par (porque es divisible entre 2)
- Si no es 0, el número es impar

Ejemplo de ejecución:

```
Ingrese un número: 8
El número es par
```

(Porque 8 % 2 = 0)

```
Ingrese un número: 7
El número es impar
```

(Porque 7 % 2 = 1, que no es 0)

Nota importante: El operador `%` calcula el **resto** de la división.

- Si `numero % 2 == 0`, el número es par
- Si `numero % 2 != 0`, el número es impar

Este algoritmo usa una estructura condicional (`si...entonces...sino`) para tomar decisiones.

Cómo leer un enunciado

Antes de diseñar el algoritmo, conviene extraer del enunciado:

- **Entrada**: datos que el programa recibe (lo que el usuario ingresa o se lee).
- **Salida**: resultado que el programa debe producir (lo que se muestra o se devuelve).

Plantilla útil:

- ¿Qué datos entran?
- ¿Qué se debe mostrar o devolver?

Ejemplo (enunciado claro): «Dados dos números enteros, calcular y mostrar su suma.»

- Entrada: dos números enteros
- Salida: la suma (un número)

Enunciado ambiguo vs. enunciado claro

Enunciado ambiguo: «Calcular el promedio.»

- No se especifica: ¿de cuántos valores? ¿enteros o decimales? ¿qué hacer si no hay datos?

Enunciado más claro: «Dadas tres calificaciones (números reales), calcular y mostrar el promedio.»

- Entrada: tres números reales
- Salida: el promedio (un número real)
- No hay ambigüedad sobre la cantidad ni el tipo de datos

Requerimientos: qué debe hacer el programa (la funcionalidad esperada).

- Ejemplo: «El programa debe calcular el promedio de las calificaciones.»

Restricciones: límites o condiciones sobre los datos o el problema.

- Ejemplo: «Se tienen exactamente 3 calificaciones.»
- Otro ejemplo: «Los números son enteros positivos.»

En el ejemplo del promedio de tres calificaciones:

- Requerimiento: calcular y mostrar el promedio
- Restricción: siempre son 3 valores (no N valores)

Ciclo que se aplica al resolver problemas:

1. **Entender**: comprender el problema, entradas y salidas
2. **Planificar**: diseñar el algoritmo (p. ej. en pseudocódigo)
3. **Ejecutar**: implementar en código y compilar
4. **Revisar**: probar, depurar y verificar

En esta asignatura se hace especial hincapié en las dos primeras fases (entender y planificar). Un buen análisis y diseño reducen errores y facilitan la implementación.

Proceso paso a paso:

1. **Problema**: entender qué necesitamos resolver
2. **Algoritmo**: diseñar la solución en pseudocódigo
3. **Código fuente**: escribir el programa en C
4. **Compilación**: traducir a lenguaje de máquina
5. **Ejecución**: correr el programa

Flujo:

```
Problema
  ↓
Algoritmo (Pseudocódigo)
  ↓
Código Fuente (.c)
  ↓
Compilación
  ↓
Programa Ejecutable
  ↓
Ejecución
```

Herramientas necesarias:

- **Editor de texto**: para escribir código (recomendado: https://zinjai.sourceforge.net/)
- **Compilador**: traduce código a ejecutable
  - En este curso: GCC (GNU Compiler Collection)
- **Terminal/Consola**: para ejecutar comandos

Comando básico:

```
gcc programa.c -o programa
./programa
```

Archivos generados:

- `programa.c` → código fuente
- `programa` → ejecutable (sin extensión en Linux/Mac)

¿Qué hace el compilador? El compilador traduce el código C (que es legible) a un programa ejecutable (que la computadora puede ejecutar).

Proceso simplificado:

1. **Preprocesador**
   - Procesa directivas como `#include`
   - Prepara el código para compilación
2. **Compilador**
   - Traduce código C a lenguaje de máquina
   - Verifica que no haya errores de sintaxis
3. **Linker (Enlazador)**
   - Combina todo en un programa ejecutable
   - Conecta con funciones estándar (como `printf`)

Lo importante: Se escribe código C → El compilador lo traduce → Se obtiene un programa ejecutable

Visualización del proceso:

```
programa.c
  (código fuente)
  ↓
[Compilador]
  (traduce automáticamente)
  ↓
programa (ejecutable)
  (programa que podemos ejecutar)
```

Comando que usaremos:

```
gcc programa.c -o programa
```

- `gcc`: el compilador
- `programa.c`: el código fuente
- `-o programa`: el nombre del ejecutable
- El compilador hace todo el trabajo automáticamente

Nota: No es necesario entender todos los detalles internos. Lo importante es saber que el compilador convierte el código en un programa ejecutable.

Flujo completo simplificado:

1. Se escribe código
   `programa.c`

2. Se compila
   `gcc programa.c -o programa`

3. Se ejecuta
   `./programa`

Lo que necesitamos saber:

- Se escribe código en un archivo `.c`
- Se usa `gcc` para compilar
- Se obtiene un ejecutable que se puede ejecutar
- El compilador se encarga de todos los detalles técnicos

Analogía: Es como escribir una carta en español y tener un traductor que la convierte a otro idioma. Se escribe en C, el compilador traduce a lenguaje de máquina.

Ejemplo práctico:

```
// 1. Se escribe programa.c
#include <stdio.h>
int main() {
    printf("Hola mundo!\n");
    return 0;
}

// 2. Compilamos
$ gcc programa.c -o programa

// 3. Ejecutamos
$ ./programa
Hola mundo!
```

Errores comunes:

- Si hay errores de sintaxis, el compilador avisará
- Se deben corregir los errores antes de poder ejecutar
- Los mensajes de error ayudan a encontrar el problema

En la próxima clase: Se verá cómo escribir el primer programa completo en C.

Tipos de errores:

- **Errores de sintaxis**
  - El código no sigue las reglas del lenguaje
  - Detectados durante la compilación
  - Ejemplo: falta punto y coma
- **Errores de lógica**
  - El programa compila pero no hace lo esperado
  - Detectados durante la ejecución
  - Ejemplo: usar + en lugar de -

Ejemplo: Error de Sintaxis

Ejemplo de error de sintaxis:

```c
#include <stdio.h>

int main() {
    int x = 5  // Error: falta punto y coma
    printf("%d\n", x);
    return 0;
}
```

Mensaje del compilador:

```
error: expected ';' before 'printf'
```

Solución:

```c
int x = 5;  // Agregar punto y coma
```

Ejemplo: Error de Lógica

Problema: Calcular el promedio de dos números

Código con error de lógica:

```c
#include <stdio.h>

int main() {
    int num1, num2;
    float promedio;

    printf("Ingrese primer número: ");
    scanf("%d", &num1);

    printf("Ingrese segundo número: ");
    scanf("%d", &num2);

    // Error: debería dividir entre 2
    promedio = (num1 + num2) / 3;

    printf("El promedio es: %.2f\n", promedio);
    return 0;
}
```

El programa compila correctamente, pero el resultado es incorrecto.

Análisis del error:

- El código tiene sintaxis correcta
- Compila sin errores
- Pero la lógica está mal: divide entre 3 en lugar de 2

Solución correcta:

```
promedio = (num1 + num2) / 2.0;
```

Nota importante:

- Los errores de lógica son más difíciles de detectar
- Requieren probar el programa con diferentes valores
- Es fundamental entender el problema antes de programar

Proceso recomendado:

1. **Entender el problema**
   - Leer cuidadosamente el enunciado
   - Identificar qué se necesita resolver
   - Identificar entradas y salidas
2. **Diseñar el algoritmo**
   - Escribir en pseudocódigo
   - Probar mentalmente con ejemplos
   - Verificar que resuelve el problema
3. **Probar mentalmente o con traza**
   - Anotar en una tabla los valores de las variables y la salida paso a paso
   - Confirmar que el algoritmo se comporta como se espera antes de codificar
4. **Implementar en código**
   - Traducir pseudocódigo a C
   - Seguir la estructura diseñada
   - Comentar el código cuando sea necesario
5. **Probar y depurar**
   - Ejecutar con diferentes valores
   - Verificar resultados esperados
   - Corregir errores encontrados

Ventajas de esta metodología:

- Reduce errores de lógica
- Facilita la depuración
- Mejora la comprensión del problema
- Permite reutilizar algoritmos
- Facilita el trabajo en equipo

Ejemplo práctico:

```
Problema: Calcular área de rectángulo

Paso 1: Entender
- Entrada: base, altura
- Salida: área

Paso 2: Algoritmo
1. Leer base
2. Leer altura
3. área = base * altura
4. Mostrar área

Paso 3: Implementar en C
Paso 4: Probar con base=5, altura=3
```

Resumen de Conceptos Clave

Conceptos aprendidos hoy:

- **Programar**: dar instrucciones precisas a una computadora
- **Lenguajes de programación**: herramientas para comunicarnos
  - Alto nivel: más legible (C, Python)
  - Bajo nivel: más cercano al hardware
- **Algoritmos**: secuencia ordenada de pasos para resolver un problema

Herramientas y procesos:

- **Pseudocódigo**: diseño de algoritmos antes de programar
- **Compilación**: proceso de traducir código a ejecutable
- **Flujo completo**: Problema $\longrightarrow$ Algoritmo $\longrightarrow$ Código $\longrightarrow$ Ejecutable

Próxima clase:

- Estructura básica de un programa en C
- Primer programa
- Compilación y ejecución práctica

Ejercicio para Pensar

Problema: Diseñar un algoritmo en pseudocódigo para calcular el área de un rectángulo.

Pasos a seguir:

1. Identificar las entradas necesarias
   - ¿Qué datos necesitamos?
2. Identificar la salida esperada
   - ¿Qué queremos obtener?
3. Escribir los pasos en pseudocódigo
   - ¿Cuál es la fórmula del área?
4. Probar mentalmente con valores de ejemplo
   - ¿Funciona con base=5 y altura=3?

Pista: El área de un rectángulo se calcula como: área = base × altura

Solución sugerida:

Calcular área de rectángulo

```
1 Alg: Calcular área de rectángulo
2 Vars: real base, altura, area
3 inicio
4     print «Ingrese la base:»
5     read base
6     print «Ingrese la altura:»
7     read altura
8     area = base * altura
9     print «El área es:», area
```

Prueba con valores:

```
base = 5, altura = 3
área = 5 × 3 = 15
```

Observación: Este algoritmo es claro, tiene todos los pasos necesarios y puede implementarse fácilmente en C.

Antes de la próxima clase

Checklist para estar listo:

- Tener instalado un **editor** (ZinjaI u otro indicado en el curso)
- Tener **GCC** instalado; verificar con: `gcc --version` en la terminal
- Haber escrito, compilado y ejecutado al menos un programa «Hola mundo» (aunque sea copiado)

Objetivo: Llegar a la próxima clase con el entorno funcionando para practicar directamente en C.

Tareas para esta semana:

- Completar el ejercicio de pseudocódigo
- Leer sobre la estructura básica de C
- Preparar el entorno de desarrollo
- Familiarizarse con un editor de texto

Herramientas recomendadas:

- Editor: **ZinjaI** (recomendado), VS Code, Vim, o cualquier editor de texto
- Compilador: GCC (instalar si no está instalado)
- Terminal/Consola para ejecutar comandos

Bibliografía sugerida:

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

Próxima clase:

- Estructura de un programa C
- Función `main()`
- `printf` y entrada/salida básica
- Primer programa completo