

Apunte de C para principiantes

El lenguaje C es uno de los más importantes en el mundo de la programación. A pesar de su antigüedad, sigue siendo ampliamente utilizado y enseñado en universidades e instituciones de tecnología. A continuación, te explico las razones clave por las que alguien debería estudiar C.

1 – Introducción al Lenguaje C

Estructura de un programa en C

Un programa en C está compuesto por varias partes:

- Directivas de preprocesador
- Declaración de funciones
- Función principal (**main**)
- Definición de variables
- Lógica del programa (instrucciones y sentencias)
- Funciones auxiliares (opcionales)

```
//❶ Directivas del preprocesador
#include <stdio.h> // Librería estándar para entrada/salida
#include <math.h> // Librería para funciones matemáticas
//❷ Declaración de funciones (prototipos)
float calcularAreaCirculo(float radio);
//❸ Función principal (punto de entrada del programa)
int main() {
    //❹ Definición de variables
    float radio, area;
    //❺ Lógica del programa
    printf("Ingrese el radio del círculo: ");
    scanf("%f", &radio);
    // Llamada a una función auxiliar
    area = calcularAreaCirculo(radio);
    printf("El área del círculo con radio %.2f es: %.2f\n", radio, area);
    return 0; //❻ Fin del programa (código de salida)
}
//❼ Definición de funciones auxiliares
float calcularAreaCirculo(float radio) {
    return M_PI * radio * radio; // Fórmula del área de un círculo
}
```

Directivas del Preprocesador

Se usan para incluir **librerías** necesarias antes de compilar el código.

```
#include <stdio.h>    // Para funciones como printf() y scanf()
#include <math.h>     // Para usar funciones matemáticas como M_PI
```

Declaración de Funciones (Prototipos)

Se escriben antes de `main()` para indicar que existen funciones auxiliares en el programa. Permiten que `main()` pueda llamarlas sin errores.

```
float calcularAreaCirculo(float radio);
```

Función Principal (`main`)

Es el punto de entrada del programa, donde inicia la ejecución y debe existir en todos los programas en C.

```
int main() {
    // Código del programa
    return 0; // Indica que el programa finalizó correctamente
}
```

Definición de Variables

Se deben declarar antes de usarlas. C permite **diferentes tipos de datos**.

```
int edad;    // Entero
float precio; // Decimal
char letra;  // Caracter
```

Lógica del Programa (Instrucciones y Sentencias): Incluye la **entrada de datos**, **procesamiento** y **salida de datos**.

```
printf("Ingrese un número: ");
scanf("%d", &numero);
printf("El número ingresado es: %d", numero);
```

Funciones Auxiliares (Opcionales): Son fragmentos de código reutilizables que **separan la lógica del programa**. Mejoran la organización y evitan repetir código.

```
float calcularAreaCirculo(float radio) {
    return M_PI * radio * radio;
}
```

Composición Secuencial de Acciones

La **composición secuencial** significa que las instrucciones se ejecutan en el orden en que aparecen en el código. Cada instrucción se ejecuta una tras otra, sin saltos ni repeticiones automáticas.

```
#include <stdio.h>
int main() {
    printf("Bienvenido al programa\n");
    printf("Por favor, ingrese su nombre: ");
    char nombre[20];
    scanf("%s", nombre);
    printf("Hola, %s!\n", nombre);
    return 0;
}
```

- Imprime un mensaje de bienvenida.
- Pide al usuario su nombre.
- Lee el nombre ingresado y lo muestra en pantalla.

Procesos

En el contexto de C y los sistemas operativos, un **proceso** es un **programa en ejecución**. Cuando un programa escrito en C es ejecutado, el sistema operativo lo convierte en un proceso que tiene su propia **memoria, registros, pila y recursos**.

Características de un Proceso

- **Código ejecutable:** Instrucciones del programa en C.
- **Memoria asignada:** Datos y variables del programa.
- **Contexto de ejecución:** Estado de los registros del procesador.
- **Recursos del sistema:** Archivos abiertos, conexiones de red, etc.

Ejemplo básico de un Proceso en C

Cuando ejecutamos un programa en C, el sistema operativo crea un proceso para ejecutarlo.

```
#include <stdio.h>
int main() {
    printf("¡Hola, soy un proceso en ejecución!\n");
    return 0;
}
```

```
}
```

- **Compilamos el código:** `gcc programa.c -o programa`
- **Ejecutamos:** `./programa`
- **El sistema operativo crea un proceso** para ejecutar el programa.
- **El proceso imprime un mensaje** en la consola.
- **El proceso finaliza y libera sus recursos.**

Un proceso dentro de otro

El programa imprimirá mensajes indicando que el proceso está iniciando, ejecutando y terminando.

```
#include <stdio.h>
#include <unistd.h> // Para usar sleep()
// Función que representa un proceso en ejecución
void procesoEjemplo() {
    printf("🟦 Proceso iniciado...\n");
    sleep(2); // Simula que el proceso está haciendo algo durante 2 segundos
    printf("🟢 Proceso ejecutándose...\n");
    sleep(2);
    printf("✅ Proceso finalizado.\n");
}
int main() {
    printf("💻 Iniciando programa...\n");
    // Llamamos a la función que simula un proceso
    procesoEjemplo();
    printf("🏁 Programa terminado.\n");
    return 0;
}
```

- **El programa inicia** e imprime "💻 Iniciando programa...".
- **Llamamos a `procesoEjemplo()`**, que representa un proceso en ejecución.
- **Dentro de `procesoEjemplo()`:**
 - Se imprime "🟦 Proceso iniciado...".
 - Se usa `sleep(2)`, lo que hace que el programa espere **2 segundos**.
 - Luego imprime "🟢 Proceso ejecutándose..." y espera otros **2 segundos**.
 - Finalmente, imprime "✅ Proceso finalizado."
- **El control vuelve a `main()`** y se imprime "🏁 Programa terminado."

Acciones y Estados

Acciones:

Son las tareas que realiza un proceso (leer datos, ejecutar cálculos, imprimir en pantalla, esperar recursos, etc.).

Estados:

Son las fases en las que se encuentra un proceso en su ciclo de vida.

Un proceso en C pasa por distintos estados en su ejecución:

Estado	Descripción
Nuevo	Se crea el proceso pero aún no se ejecuta.
Listo	Está esperando que el sistema le asigne CPU.
Ejecución	Está siendo ejecutado en el CPU.
Espera	Está pausado esperando un recurso (ejemplo: entrada del usuario).
Terminado	Ha finalizado su ejecución.

Este código **simula un proceso** que pasa por los diferentes estados y realiza acciones en cada uno.

```
#include <stdio.h>
#include <unistd.h> // Para sleep()

void ejecutarProceso() {
    printf("🟦 Estado: NUEVO - Creando proceso...\n");
    sleep(1);

    printf("🟡 Estado: LISTO - Esperando asignación de CPU...\n");
    sleep(1);

    printf("🟢 Estado: EJECUCIÓN - El proceso está corriendo...\n");
    sleep(2); // Simula el tiempo de ejecución

    printf("⌚ Estado: ESPERA - El proceso está esperando entrada del usuario...\n");
    char input;
    printf("Presiona cualquier tecla y Enter para continuar: ");
```

```

scanf(" %c", &input); // Espera entrada del usuario

printf("✅ Estado: TERMINADO - El proceso ha finalizado.\n");
}

int main() {
printf("🚀 Iniciando el programa...\n");
ejecutarProceso();
printf("🏁 Programa finalizado.\n");
return 0;
}

```

- El programa inicia en `main()` y llama a `ejecutarProceso()`.
- Estados del proceso en `ejecutarProceso()`:
 - **NUEVO**: Se indica que el proceso se está creando.
 - **LISTO**: El proceso espera ser ejecutado por el CPU.
 - **EJECUCIÓN**: El proceso está activo (simulado con `sleep(2)`).
 - **ESPERA**: Se pausa esperando entrada del usuario (`scanf`).
 - **TERMINADO**: El proceso finaliza.
- El programa finaliza y vuelve a `main()`, mostrando "🏁 Programa finalizado."

Variables y Constantes

Variables

Una **variable** es un espacio en memoria que guarda un valor que **puede cambiar** durante la ejecución del programa.

Cada variable tiene un **tipo de dato** que define qué tipo de información puede almacenar.

Para declarar una variable en C, usamos la sintaxis:

```

tipo_de_dato nombre_variable;

```

Variables en un Programa en C

```

#include <stdio.h>

int main() {
    // Declaración de variables

```

```

int edad = 25;           // Variable de tipo entero
float precio = 15.99;    // Variable de tipo flotante
char letra = 'A';        // Variable de tipo carácter

// Mostramos los valores de las variables
printf("Edad: %d años\n", edad);
printf("Precio: $%.2f\n", precio);
printf("Letra: %c\n", letra);

// Modificamos la variable edad
edad = 30;
printf("Nueva edad: %d años\n", edad);

return 0;
}

```

La variable `edad` cambió su valor de `25` a `30`, lo que demuestra que **una variable puede modificarse** en la ejecución.

Constantes

Una **constante** es un valor que **no cambia** durante la ejecución del programa. Se usa la palabra clave `const` o la directiva `#define` para definir las.

```

const float PI = 3.1416;
#define PI 3.1416

```

```

#include <stdio.h>
#define PI 3.1416 // Definimos una constante con #define
int main() {
    const float GRAVEDAD = 9.81; // Definimos una constante con const
    float radio = 5.0;
    // Calculamos el área de un círculo con PI
    float area = PI * radio * radio;
    printf("Radio: %.2f\n", radio);
    printf("Área del círculo: %.2f\n", area);
    printf("Valor de la gravedad: %.2f m/s^2\n", GRAVEDAD);

    // GRAVEDAD = 10; ❌ Esto daría error porque es una constante
    return 0;
}

```


Diferencias entre Variables y Constantes

Característica	Variable	Constante
Valor	Puede cambiar en la ejecución	No cambia una vez definida
Declaración	<code>int edad = 25;</code>	<code>const float PI = 3.1416; o #define PI 3.1416</code>
Modificable	✓ Sí	✗ No
Ejemplo	<code>edad = 30;</code>	<code>PI = 3.5; // ✗ ERROR</code>

Operaciones básicas de salida

En **C**, las operaciones de salida se utilizan para mostrar información en la pantalla. La función principal para esto es `printf()`, que nos permite imprimir texto y valores de variables.

Funciones Básicas de Salida

`printf()` pertenece a la biblioteca estándar `stdio.h`, por lo que siempre debemos incluir esta línea al inicio del programa:

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

Su sintaxis básica es:

```
printf("Texto a mostrar");
```

También podemos imprimir valores de **variables** usando **especificadores de formato**.

Especificadores de Formato en `printf()`

Sí, aquí tienes la **sintaxis de `printf()`** para mostrar diferentes tipos de datos en C, incluyendo **enteros, caracteres, flotantes y más**, con ejemplos detallados.

Mostrar un número entero (int)

```
#include <stdio.h>

int main() {
    int edad = 25;
    printf("Tengo %d años\n", edad);
    return 0;
}
```

Salida:

```
Tengo 25 años
```

Mostrar un número decimal (**float**)

```
#include <stdio.h>

int main() {
    float pi = 3.14159;
    printf("El valor de Pi es %f\n", pi);
    return 0;
}
```

Salida:

```
El valor de Pi es 3.141590
```

Nota: **printf()** imprime 6 decimales por defecto.

Controlar la cantidad de decimales

Si quieres **limitar la cantidad de decimales**, usa **%.xf**:

```
#include <stdio.h>

int main() {
    float pi = 3.14159;
    printf("Pi con 2 decimales: %.2f\n", pi);
    printf("Pi con 4 decimales: %.4f\n", pi);
    return 0;
}
```

Salida:

```
Pi con 2 decimales: 3.14
Pi con 4 decimales: 3.1416
```

Nota: Se redondea automáticamente.

Mostrar un número de doble precisión (**double**)

```
#include <stdio.h>

int main() {
    double numero = 123.456789;
    printf("Número double: %lf\n", numero);
    printf("Número double con 3 decimales: %.3lf\n", numero);
    return 0;
}
```

Salida:

```
Número double: 123.456789
Número double con 3 decimales: 123.457
```

Mostrar un carácter (**char**)

```
#include <stdio.h>

int main() {
    char letra = 'A';
    printf("La letra es %c\n", letra);
    return 0;
}
```

Salida:

```
La letra es A
```

Mostrar una cadena de caracteres (**char[]**)

```
#include <stdio.h>

int main() {
    char nombre[] = "Juan";
    printf("Mi nombre es %s\n", nombre);
}
```

```
    return 0;
}
```

Salida:

```
Mi nombre es Juan
```

Mostrar múltiples variables en un solo `printf()`

```
#include <stdio.h>

int main() {
    int edad = 30;
    float altura = 1.75;
    char inicial = 'J';

    printf("Tengo %d años, mido %.2f metros y mi inicial es %c\n", edad,
altura, inicial);
    return 0;
}
```

Salida:

```
Tengo 30 años, mido 1.75 metros y mi inicial es J
```

Formatos más usados en `printf()`

Formato	Tipo de dato	Ejemplo
%d	Entero (<code>int</code>)	<code>printf("%d", 100);</code>
%f	Flotante (<code>float</code>)	<code>printf("%.2f", 3.14);</code>
%lf	Doble (<code>double</code>)	<code>printf("%.3lf", 2.71828);</code>
%c	Carácter (<code>char</code>)	<code>printf("%c", 'A');</code>

<code>%s</code>	Cadena (<code>char[]</code>)	<code>printf("%s", "Hola");</code>
-----------------	--------------------------------	------------------------------------

Salto de Línea y Caracteres Especiales

Dentro de las cadenas de `printf()`, podemos usar algunos caracteres especiales:

Carácter Especial	Función
<code>\n</code>	Salto de línea
<code>\t</code>	Tabulación (espacio grande)
<code>\\</code>	Imprime una barra invertida (<code>\</code>)
<code>\"</code>	Imprime comillas dobles (<code>"</code>)

```
#include <stdio.h>

int main() {
    printf("Hola\nMundo\n");
    printf("Este es un\t ejemplo con tabulación.\n");
    printf("Usamos comillas dobles: \"Hola\"\n");

    return 0;
}
```

Salida esperada:

```
Hola
Mundo
Este es un    ejemplo con tabulación.
Usamos comillas dobles: "Hola"
```

Funciones Básicas de Entrada

Las **funciones básicas de entrada** permiten al usuario ingresar datos desde el teclado. Las funciones más usadas son:

scanf() → Para leer datos primitivos como enteros, flotantes, caracteres y cadenas de texto.

fgets() → Alternativa segura para leer cadenas de texto.

scanf()

scanf() se usa para leer datos de diferentes tipos. Su sintaxis es:

```
scanf("formato", &variable);
```

- Se usa **&** (ampersand) antes de la **variable** cuando se trata de números (**int**, **float**, etc.).
- Para cadenas (**char[]**), no se usa **&** porque los arrays ya representan direcciones de memoria.

Leer un número entero

```
#include <stdio.h>

int main() {
    int edad;
    printf("Ingrese su edad: ");
    scanf("%d", &edad);
    printf("Su edad es %d años.\n", edad);
    return 0;
}
```

Leer varios datos (entero y flotante)

```
#include <stdio.h>

int main() {
    int edad;
    float altura;

    printf("Ingrese su edad y altura separados por espacio: ");
    scanf("%d %f", &edad, &altura);

    printf("Edad: %d años, Altura: %.2f metros\n", edad, altura);
    return 0;
}
```

Leer caracteres

Para leer un **carácter** (**char**), usamos **%c**, pero hay un problema:

scanf("%c", &variable) puede capturar el ENTER de una entrada anterior.

Leer un solo carácter

```
#include <stdio.h>

int main() {
    char letra;
    printf("Ingrese una letra: ");
    scanf(" %c", &letra); // Espacio antes de %c evita capturar el ENTER
    printf("La letra ingresada es: %c\n", letra);
    return 0;
}
```

El **espacio antes de %c** evita capturar el ENTER que queda en el buffer.

Leer una cadena de texto

Leer una palabra

```
#include <stdio.h>

int main() {
    char nombre[20];

    printf("Ingrese su nombre: ");
    scanf("%s", nombre);

    printf("Hola, %s!\n", nombre);
    return 0;
}
```

scanf("%s", nombre); solo captura una palabra. No lee espacios.

Leer una línea completa con **fgets()** (Alternativa Segura)

Como **scanf("%s", variable)** no lee espacios, para leer una línea usamos **fgets()**.

Leer una frase con espacios

```
#include <stdio.h>

int main() {
    char nombre_completo[50];

    printf("Ingrese su nombre completo: ");
    fgets(nombre_completo, sizeof(nombre_completo), stdin);

    printf("Hola, %s", nombre_completo);
    return 0;
}
```

Formatos en `scanf()`

Formato	Tipo de Dato	Ejemplo de Código
<code>%d</code>	Entero (<code>int</code>)	<code>scanf("%d", &num);</code>
<code>%f</code>	Flotante (<code>float</code>)	<code>scanf("%f", &num);</code>
<code>%lf</code>	Doble precisión (<code>double</code>)	<code>scanf("%lf", &num);</code>
<code>%c</code>	Carácter (<code>char</code>)	<code>scanf(" %c", &letra);</code>
<code>%s</code>	Cadena (<code>char[]</code>)	<code>scanf("%s", nombre);</code>
<code>fgets()</code>	Cadena con espacios	<code>fgets(nombre, 50, stdin);</code>

2 - Estructura de Datos

Unidad 2 - Estructura de Datos en C

Tipos Primitivos de Datos

Son los tipos básicos proporcionados por el lenguaje C:

Tipo	Descripción	Ejemplo	Tamaño en Memoria
<code>int</code>	Números enteros	<code>int edad = 25;</code>	4 bytes (en la mayoría de sistemas)
<code>float</code>	Números decimales de precisión simple	<code>float precio = 10.5;</code>	4 bytes
<code>double</code>	Números decimales de precisión doble	<code>double pi = 3.141592;</code>	8 bytes
<code>char</code>	Caracteres individuales	<code>char letra = 'A';</code>	1 byte

```
#include <stdio.h>

int main() {
    int numero = 10;
    float decimal = 3.14;
    char letra = 'A';

    printf("Número entero: %d\n", numero);
    printf("Número decimal: %.2f\n", decimal);
    printf("Letra: %c\n", letra);

    return 0;
}
```

Salida esperada:

```
Número entero: 10
Número decimal: 3.14
```

Dominio de cada Tipo de Dato y Operaciones Válidas

Cada tipo de dato tiene un rango de valores permitidos:

Tipo	Dominio (Rango de Valores)
int	-2,147,483,648 a 2,147,483,647 (en sistemas de 4 bytes)
float	Aproximadamente $\pm 3.4E-38$ a $\pm 3.4E+38$
double	Aproximadamente $\pm 1.7E-308$ a $\pm 1.7E+308$
char	-128 a 127 (valores ASCII)

Los tipos de datos soportan distintas **operaciones matemáticas, relacionales y lógicas**:

Operaciones Matemáticas

Operador	Descripción	Ejemplo
+	Suma	$a + b$
-	Resta	$a - b$
*	Multiplicación	$a * b$
/	División	a / b
%	Módulo (resto de la división)	$a \% b$

Operaciones Relacionales

Operador	Descripción
==	Igualdad
!=	Diferente
>	Mayor que
<	Menor que

>=	Mayor o igual que
<=	Menor o igual que

Operaciones Lógicas

Operador	Descripción
&&	AND (Y lógico)
,	
!	NOT (Negación)

```
#include <stdio.h>

int main() {
    int a = 10, b = 5;
    printf("a > b: %d\n", a > b);
    printf("a == 10 && b == 5: %d\n", a == 10 && b == 5);
    return 0;
}
```

Precedencia entre Operadores

La precedencia define qué operación se ejecuta primero:

1. (): Paréntesis
2. * / %: Multiplicación, división, módulo
3. + -: Suma y resta
4. > < >= <=: Comparaciones
5. == !=: Igualdades
6. &&: AND lógico
7. ||: OR lógico
8. =: Asignación

```
#include <stdio.h>

int main() {
```

```
int resultado = 5 + 3 * 2;
printf("Resultado: %d\n", resultado);
return 0;
}
```

Estructuras de Datos

Las estructuras de datos pueden ser **estáticas** o **dinámicas**:

- **Estáticas:** Su tamaño se define en tiempo de compilación (ej: arrays).
- **Dinámicas:** Se asignan y liberan en tiempo de ejecución (ej: listas enlazadas).

💡 **Ejemplo de array estático:**

```
#include <stdio.h>

int main() {
    int numeros[5] = {1, 2, 3, 4, 5};
    printf("Primer elemento: %d\n", numeros[0]);
    return 0;
}
```

3 - Sentencias de Control e Iteración

La Estructura de Decisión

Las estructuras de decisión permiten que un programa tome diferentes caminos según ciertas condiciones.

Sentencia if

La sentencia `if` permite ejecutar un bloque de código **sólo si una condición es verdadera**. Si la condición es falsa, el código dentro del `if` no se ejecuta.

```
if (condición) {  
    // Código que se ejecuta si la condición es verdadera  
}
```

Evaluar si un número es positivo

```
#include <stdio.h>  
  
int main() {  
    int numero;  
  
    printf("Ingrese un número: ");  
    scanf("%d", &numero);  
  
    if (numero > 0) {  
        printf("El número es positivo.\n");  
    }  
  
    printf("Fin del programa.\n");  
  
    return 0;  
}
```

1. **Pide al usuario** que ingrese un número.
2. **Verifica si el número es mayor que 0** con la condición `numero > 0`.

3. Si la condición es verdadera, imprime "El número es positivo."
4. Si la condición es falsa, no imprime nada y sigue con el resto del código.
5. Siempre imprime "Fin del programa." al final.

Sentencia if-else

La sentencia `if-else` permite ejecutar un bloque de código si la condición es verdadera y otro bloque si la condición es falsa.

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

Verificar si un número es positivo o negativo

```
#include <stdio.h>  
  
int main() {  
    int numero;  
  
    printf("Ingrese un número: ");  
    scanf("%d", &numero);  
  
    if (numero > 0) {  
        printf("El número es positivo.\n");  
    } else {  
        printf("El número es negativo o cero.\n");  
    }  
  
    return 0;  
}
```

1. Pide al usuario un número.
2. Evalúa si el número es positivo (`numero > 0`).
 - Si la condición es verdadera, imprime "El número es positivo."
 - Si la condición es falsa, ejecuta el bloque `else`, que imprime "El número es negativo o cero."

Diferencia clave entre if y if-else

- **if** solo ejecuta código cuando la condición es verdadera, si es falsa, no hace nada.
- **if-else** tiene dos caminos posibles, uno cuando la condición es verdadera y otro cuando es falsa.

Sentencias if-else anidados

Los **if-else** anidados permiten tomar **decisiones más complejas** dentro de un programa. Se usan cuando se necesita evaluar **múltiples condiciones de manera jerárquica**.

Un **if-else** anidado ocurre cuando **dentro de un if o un else hay otro if-else**. Se usa para evaluar **varias condiciones** en un orden específico.

```
if (condición1) {  
    // Código si condición1 es verdadera  
    if (condición2) {  
        // Código si condición2 también es verdadera  
    } else {  
        // Código si condición2 es falsa  
    }  
} else {  
    // Código si condición1 es falsa  
}
```

Verificar si un número es positivo, negativo o cero

```
#include <stdio.h>  
  
int main() {  
    int numero;  
  
    printf("Ingrese un número: ");  
    scanf("%d", &numero);  
  
    if (numero > 0) { // Primer if  
        printf("El número es positivo.\n");  
    } else {  
        if (numero < 0) { // Segundo if dentro del else  
            printf("El número es negativo.\n");  
        } else {  
            printf("El número es cero.\n"); // Se ejecuta si numero no es
```

```

    ni mayor ni menor a 0
    }
}

return 0;
}

```

1. Se solicita un número al usuario.
2. Se verifica si es mayor que 0 (positivo).
3. Si no es positivo (**else**), se evalúa si es negativo.
4. Si tampoco es negativo, entonces es 0.

Clasificación de notas

```

#include <stdio.h>

int main() {
    int nota;

    printf("Ingrese la nota del estudiante (0-100): ");
    scanf("%d", &nota);

    if (nota >= 90) {
        printf("Calificación: A\n");
    } else {
        if (nota >= 80) {
            printf("Calificación: B\n");
        } else {
            if (nota >= 70) {
                printf("Calificación: C\n");
            } else {
                if (nota >= 60) {
                    printf("Calificación: D\n");
                } else {
                    printf("Calificación: F\n");
                }
            }
        }
    }

    return 0;
}

```

1. Si la nota es **90 o más**, se muestra "A".

2. Si no, pero es **80 o más**, se muestra "B".
3. Si no, pero es **70 o más**, se muestra "C".
4. Si no, pero es **60 o más**, se muestra "D".
5. Si ninguna de las anteriores se cumple, se muestra "F".

Menú interactivo con subopciones

```
#include <stdio.h>

int main() {
    int opcion;

    printf("Menú de opciones:\n");
    printf("1. Operaciones matemáticas\n");
    printf("2. Juegos\n");
    printf("Seleccione una opción: ");
    scanf("%d", &opcion);

    if (opcion == 1) { // Primer nivel de decisión
        int subopcion;
        printf("\nSeleccione la operación:\n");
        printf("1. Suma\n");
        printf("2. Resta\n");
        printf("Seleccione una subopción: ");
        scanf("%d", &subopcion);

        if (subopcion == 1) {
            printf("Seleccionaste Suma.\n");
        } else if (subopcion == 2) {
            printf("Seleccionaste Resta.\n");
        } else {
            printf("Subopción no válida.\n");
        }
    }

    } else if (opcion == 2) { // Otra opción del primer nivel
        printf("Accediendo a juegos...\n");
    } else {
        printf("Opción no válida.\n");
    }

    return 0;
}
```

1. Se muestra un menú con opciones.

2. Si el usuario elige "1", se despliega otro submenú.
3. Si elige "2", se le muestra un mensaje relacionado con juegos.
4. Si ingresa otra opción, se le indica que no es válida.

Sentencia switch

Cuando hay múltiples opciones, `switch` simplifica el código en lugar de usar varios `if-else`.

```
#include <stdio.h>

int main() {
    int opcion;
    printf("Seleccione una opción (1-3): ");
    scanf("%d", &opcion);

    switch (opcion) {
        case 1:
            printf("Opción 1 seleccionada.\n");
            break;
        case 2:
            printf("Opción 2 seleccionada.\n");
            break;
        case 3:
            printf("Opción 3 seleccionada.\n");
            break;
        default:
            printf("Opción inválida.\n");
    }
    return 0;
}
```

La Estructura de Iteración

Permiten ejecutar repetidamente un bloque de código mientras se cumpla una condición.

Sentencia while

La sentencia `while` permite repetir un bloque de código **mientras una condición sea verdadera**. Se usa cuando no sabemos exactamente cuántas veces se repetirá el ciclo, ya que depende de la evaluación de la condición.

```
while (condición) {
    // Código que se ejecuta mientras la condición sea verdadera
}
```

- Antes de cada iteración, **se evalúa la condición**.
- Si la condición es **true** (verdadera), se ejecuta el bloque de código dentro del **while**.
- Si la condición es **false** (falsa), el ciclo termina y el programa continúa con la siguiente instrucción después del **while**.

Contador del 1 al 5

```
#include <stdio.h>

int main() {
    int contador = 1; // Inicialización de la variable

    while (contador <= 5) { // Condición: Mientras contador sea menor o
        igual a 5
        printf("Número: %d\n", contador);
        contador++; // Incrementa el contador en 1
    }

    return 0;
}
```

1. Se inicializa **contador** en 1.
2. El **while** evalúa si **contador <= 5**. Si es cierto, ejecuta el bloque dentro del ciclo.
3. Se imprime el valor de **contador**.
4. Se incrementa **contador** en 1 (**contador++**).
5. Se vuelve a evaluar la condición. Si sigue siendo verdadera, repite el ciclo.
6. Cuando **contador** llega a 6, la condición es falsa y el **while** termina.

Salida del programa:

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

Solicitar contraseña hasta que sea correcta

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

```

int main() {
    int clave;

    printf("Ingrese la clave numérica (1234): ");
    scanf("%d", &clave);

    while (clave != 1234) { // Mientras la clave no sea 1234
        printf("Clave incorrecta. Intente nuevamente: ");
        scanf("%d", &clave);
    }

    printf("¡Acceso concedido!\n");

    return 0;
}

```

1. Se le pide al usuario que ingrese una clave numérica.
2. **Si la clave ingresada no es 1234**, el programa muestra "Clave incorrecta" y vuelve a pedir la clave.
3. **Cuando el usuario ingresa 1234**, el ciclo termina y se muestra "¡Acceso concedido!".

Ejemplo de salida si el usuario ingresa claves incorrectas:

```

Ingrese la clave numérica (1234): 5678
Clave incorrecta. Intente nuevamente: 9876
Clave incorrecta. Intente nuevamente: 1234
¡Acceso concedido!

```

Errores comunes al usar while

Ciclo infinito

Si la variable dentro del **while** **no cambia dentro del ciclo**, la condición **nunca será falsa** y el programa quedará atrapado en un bucle infinito.

```


int x = 1;
while (x <= 5) {
    printf("Número: %d\n", x); // Falta incrementar x, el ciclo será
    infinito
}

```

Solución: Asegurarse de que la variable `x` cambie dentro del ciclo.

Condición incorrecta

```
int x = 10;
while (x < 5) { // x nunca será menor que 5
    printf("Esto nunca se imprimirá\n");
}
```

 **Solución:** Asegurarse de que la condición permita al `while` ejecutarse al menos una vez si es necesario.

Sentencia for

La sentencia `for` se usa para ejecutar un bloque de código **un número determinado de veces**. A diferencia de `while`, donde el control de la condición es más manual, `for` tiene una estructura más organizada con **tres partes**:

1. **Inicialización** → Se ejecuta una sola vez antes del primer ciclo.
2. **Condición** → Se evalúa antes de cada iteración; si es `true`, el ciclo continúa.
3. **Actualización** → Se ejecuta después de cada iteración para modificar la variable de control.

```
for (inicialización; condición; actualización) {
    // Código que se ejecuta en cada iteración mientras la condición sea verdadera
}
```

Contar del 1 al 5

```
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) { // Se inicializa i en 1, el ciclo
        // sigue mientras i <= 5, se incrementa i en cada iteración
        printf("Número: %d\n", i);
    }

    return 0;
}
```

1. **Inicialización:** `int i = 1;` → Se declara e inicializa `i` en 1.
2. **Condición:** `i <= 5;` → Mientras `i` sea menor o igual a 5, el ciclo continúa.

3. **Actualización:** `i++` → Se incrementa `i` en 1 después de cada iteración.
4. **Salida esperada:**

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

Sumar los primeros 10 números enteros

```
#include <stdio.h>

int main() {
    int suma = 0;

    for (int i = 1; i <= 10; i++) {
        suma += i; // Se acumula la suma de los números
    }

    printf("La suma de los primeros 10 números es: %d\n", suma);

    return 0;
}
```

1. Se inicializa `suma` en 0.
2. El ciclo `for` recorre los valores del 1 al 10.
3. En cada iteración, `suma += i;` acumula la suma de los números.
4. Al finalizar el ciclo, se muestra el resultado.

Mostrar una tabla de multiplicar (ingresada por el usuario)

```
#include <stdio.h>

int main() {
    int numero;

    printf("Ingrese un número para mostrar su tabla de multiplicar: ");
    scanf("%d", &numero);

    for (int i = 1; i <= 10; i++) {
        printf("%d x %d = %d\n", numero, i, numero * i);
    }
}
```

```
    return 0;
}
```

1. Se solicita un número al usuario.
2. El `for` recorre los valores del 1 al 10.
3. En cada iteración, se muestra la multiplicación `numero * i`.
4. **Ejemplo de salida para `numero = 3`:**

```
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
...
3 x 10 = 30
```

Errores comunes con for

No modificar la variable de control (Bucle infinito)

```
for (int i = 1; i <= 5; ) { // Falta el incremento
    printf("%d\n", i);
}
```

Solución: Agregar `i++` en la actualización.

Condición incorrecta (No entra al ciclo)

```
for (int i = 10; i < 5; i++) { // i empieza en 10, pero nunca es menor
    que 5
    printf("Esto nunca se imprimirá.\n");
}
```

Solución: Corregir la condición `i > 5` o ajustar el valor inicial de `i`.

Sentencia do-while

La sentencia `do-while` es una estructura de control de flujo que **ejecuta un bloque de código al menos una vez** y luego sigue ejecutándose mientras **una condición sea verdadera**.

A diferencia del `while`, donde primero se evalúa la condición antes de ejecutar el código, en `do-while` el código **se ejecuta al menos una vez** antes de comprobar la condición.

```
do {  
    // Código a ejecutar al menos una vez  
} while (condición);
```

- Primero se ejecuta el bloque de código dentro de **do**
- Luego se evalúa la condición
- Si la condición es **true**, se repite el proceso; si es **false**, el bucle termina

Pedir un número hasta que sea mayor que 10

```
#include <stdio.h>  
  
int main() {  
    int numero;  
  
    do {  
        printf("Ingrese un número mayor que 10: ");  
        scanf("%d", &numero);  
    } while (numero <= 10); // Si el número es 10 o menor, vuelve a pedirlo  
  
    printf("Número aceptado: %d\n", numero);  
  
    return 0;  
}
```

1. Se declara la variable **numero**.
2. Dentro del **do**, se muestra un mensaje y se pide al usuario un número.
3. **Si el número es menor o igual a 10, se repite el proceso.**
4. **Si el número es mayor a 10, el bucle termina** y se muestra el mensaje final.

```
Ingrese un número mayor que 10: 5  
Ingrese un número mayor que 10: 8  
Ingrese un número mayor que 10: 12  
Número aceptado: 12
```

- Se ejecutó al menos una vez.
- El ciclo se repitió hasta que la condición fue falsa (**numero > 10**)

Menú interactivo

Un menú que sigue mostrando opciones hasta que el usuario elija salir.

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



```

int main() {
    int opcion;

    do {
        printf("\n--- Menú ---\n");
        printf("1. Saludar\n");
        printf("2. Despedirse\n");
        printf("3. Salir\n");
        printf("Seleccione una opción: ");
        scanf("%d", &opcion);

        switch (opcion) {
            case 1:
                printf("¡Hola!\n");
                break;
            case 2:
                printf("¡Adiós!\n");
                break;
            case 3:
                printf("Saliendo del programa...\n");
                break;
            default:
                printf("Opción no válida. Intente nuevamente.\n");
        }
    } while (opcion != 3); // Repite mientras la opción no sea 3

    return 0;
}

```

1. El menú siempre se muestra al menos una vez.
2. El usuario elige una opción y se usa **switch** para ejecutar la acción.
3. Si el usuario elige 3 (Salir), el ciclo termina.
4. Si ingresa otra opción, el menú se repite.

Contar del 1 al 5 con do-while

```

#include <stdio.h>

int main() {
    int i = 1;

    do {
        printf("Número: %d\n", i);
        i++; // Incrementamos la variable i
    } while (i < 6);
}

```

```

    } while (i <= 5); // Repite hasta que i sea mayor que 5

    return 0;
}

```

1. Se inicializa **i** en 1.
2. El bloque dentro de **do** se ejecuta una vez antes de evaluar la condición.
3. Se imprime el valor de **i** y luego se incrementa.
4. Si **i <= 5**, el ciclo continúa; si **i > 5**, se detiene.

Diferencia entre while y do-while

Característica	while	do-while
¿Cuándo se evalúa la condición?	Antes de entrar al bucle	Después de ejecutar el bloque
¿Se ejecuta al menos una vez?	No, si la condición es false desde el inicio, nunca entra al ciclo	Sí, siempre se ejecuta al menos una vez
Uso recomendado	Cuando puede ser que el bloque nunca se ejecute	Cuando se necesita que el bloque se ejecute al menos una vez

Ejemplo comparativo

```

// Usando while
int x = 5;
while (x > 10) {
    printf("Esto no se imprimirá\n");
}

```

```

// Usando do-while
int y = 5;
do {
    printf("Esto se imprimirá al menos una vez\n");
} while (y > 10);

```

Errores comunes con do-while

Olvidar la actualización de la variable de control

```
int i = 1;
do {
    printf("%d\n", i);
} while (i <= 5); // ¡Ciclo infinito porque i nunca cambia!
```

Solución: `i++` dentro del bucle.

Usar `;` después de `while` erróneamente

```
do {
    printf("Hola\n");
} while (0); // Aquí es correcto, se ejecutará una vez
```

4 - Arreglos, Cadenas y Estructuras dinámicas

Arreglos

Un **arreglo** (o array) es una **colección de variables del mismo tipo** almacenadas en **posiciones contiguas de memoria**. Sirve para **guardar muchos datos similares** bajo un mismo nombre.

¿Qué son los Arreglos en C?

Un **arreglo** (o array) es una **colección de variables del mismo tipo** almacenadas en **posiciones contiguas de memoria**. Sirve para **guardar muchos datos similares** bajo un mismo nombre.

¿Para qué sirven?

Supongamos que querés guardar las notas de 5 estudiantes. Podés hacerlo así:

```
int nota1, nota2, nota3, nota4, nota5;
```

Pero es poco práctico. Con un arreglo, lo hacés en una sola línea:

```
int notas[5];
```

Ahora podés usar un índice para acceder a cada posición:

- `notas[0]` → primera nota
- `notas[1]` → segunda nota
- ...
- `notas[4]` → quinta nota (los arreglos **empiezan en 0**)

¿Cómo se declara un arreglo?

```
tipo nombre[tamaño];
```

¿Cómo se asignan valores?

Manualmente:

```
int numeros[3];  
numeros[0] = 10;  
numeros[1] = 20;  
numeros[2] = 30;
```

Directamente al declarar:

```
int numeros[3] = {10, 20, 30};
```

Arreglos Unidimensionales (Vectores)

Un arreglo unidimensional es como una fila de casilleros. Cada casillero guarda un valor.

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

```

int main() {
    int numeros[5] = {10, 20, 30, 40, 50};

    // Recorrido: mostrar cada número
    for (int i = 0; i < 5; i++) {
        printf("Elemento %d: %d\n", i, numeros[i]);
    }

    return 0;
}

```

Arreglos Bidimensionales (Matrices)

Son como una tabla con filas y columnas.

```

#include <stdio.h>

int main() {
    int matriz[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    // Recorrer la matriz
    for (int fila = 0; fila < 2; fila++) {
        for (int col = 0; col < 3; col++) {
            printf("%d ", matriz[fila][col]);
        }
        printf("\n");
    }

    return 0;
}

```

Operaciones con Arreglos

Recorrido:

Usar un `for` para mostrar o procesar cada elemento.

```

#include <stdio.h>

```

```

int main() {
    int numeros[5] = {10, 20, 30, 40, 50};

    // Recorrido: mostrar cada número
    for (int i = 0; i < 5; i++) {
        printf("Elemento %d: %d\n", i, numeros[i]);
    }

    return 0;
}

```

Búsqueda:

Buscar si un valor está en el arreglo.

```

#include <stdio.h>

int main() {
    int numeros[5] = {3, 7, 9, 1, 4};
    int buscado = 9;
    int encontrado = 0;

    for (int i = 0; i < 5; i++) {
        if (numeros[i] == buscado) {
            printf("Encontrado en la posición %d\n", i);
            encontrado = 1;
            break;
        }
    }

    if (!encontrado) {
        printf("No encontrado.\n");
    }

    return 0;
}

```

Inserción:

En C los arreglos tienen tamaño fijo, así que podemos reemplazar valores, pero no agregar nuevos dinámicamente sin usar estructuras más avanzadas.

```
numeros[2] = 100; // Reemplaza el valor en la posición 2
```

Cadenas de Caracteres

En C, una **cadena** es un **arreglo de caracteres** (**char**) que termina con el carácter nulo `'\0'`. Por ejemplo:

```
char nombre[] = "Juan"; // contiene: 'J', 'u', 'a', 'n', '\0'
```

Para manipular cadenas, se usan funciones definidas en la biblioteca:

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

Funciones de <string.h> más usadas

strcpy(destino, origen) – Copiar cadenas

Copia el contenido de una cadena **origen** en otra **destino**.

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

int main() {
    char origen[] = "Hola";
    char destino[20];

    strcpy(destino, origen); // Copiamos "Hola" a destino

    printf("Destino: %s\n", destino);
    return 0;
}
```

strlen(cadena) – Longitud de la cadena

Devuelve la **cantidad de caracteres** (sin contar el `\0`).

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

```
int main() {
    char texto[] = "Programar";
    int longitud = strlen(texto);

    printf("Longitud de '%s': %d\n", texto, longitud);
    return 0;
}
```

strcmp(cad1, cad2) – Comparar cadenas

Compara dos cadenas **carácter por carácter**.

- Devuelve 0 si son iguales
- Un número negativo si `cad1 < cad2`
- Un número positivo si `cad1 > cad2`

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

int main() {
    char a[] = "hola";
    char b[] = "hola";
    char c[] = "mundo";

    if (strcmp(a, b) == 0) {
        printf("a y b son iguales\n");
    }

    if (strcmp(a, c) != 0) {
        printf("a y c son diferentes\n");
    }

    return 0;
}
```

strcat(destino, origen) – Concatenar cadenas

Agrega el contenido de **origen** al final de **destino**.


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

int main() {
    char saludo[20] = "Hola ";
    char nombre[] = "Juan";

    strcat(saludo, nombre);

    printf("Saludo completo: %s\n", saludo);
    return 0;
}
```

strchr(cadena, caracter) – Buscar un carácter

Busca la **primera aparición** de un carácter en la cadena.

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

int main() {
    char palabra[] = "computadora";
    char *ptr = strchr(palabra, 'u');

    if (ptr != NULL) {
        printf("Encontrado: %c en la posición %ld\n", *ptr, ptr -
palabra);
    } else {
        printf("No se encontró el carácter\n");
    }

    return 0;
}
```

strstr(cadena, subcadena) – Buscar una subcadena

Busca la **primera aparición** de una subcadena dentro de otra.

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

int main() {
```

```
char frase[] = "Me gusta programar en C";
char *ptr = strstr(frase, "programar");

if (ptr != NULL) {
    printf("Subcadena encontrada: %s\n", ptr);
} else {
    printf("Subcadena no encontrada\n");
}

return 0;
}
```