

Fundamentos de Programación

1. Conceptos Básicos

a. Variables

Definición: Una variable es un espacio en la memoria (ram) que se utiliza para almacenar un valor que puede cambiar durante la ejecución de un programa.

Declaración: Para declarar una variable, debes especificar el **tipo de dato** seguido del nombre de la variable. Ejemplo:

```
int edad;
double salario;
char inicial;
```

Asignación: Después de declarar una variable, puedes asignarle un valor usando el operador =

```
edad = 25;
salario = 1500.75;
inicial = 'A';
```

Declaración y asignación simultánea:

```
int edad = 25;
double salario = 1500.75;
char inicial = 'A';
```

Hay 6 formas básicas de inicializar variables en C++:(int)

```
int a;           // Sin inicializador (inicialización predeterminada)
int b = 5;       // valor inicial después del signo igual (inicialización de copia)
int c( 6 );      // valor inicial entre paréntesis (inicialización directa)

                // Métodos de inicialización de listas (C++11) (preferido)

int d { 7 };     // valor inicial entre llaves (inicialización de lista directa)
int e = { 8 };   // Valor inicial entre llaves después del signo igual (inicialización de lista de copia)
int f {};        // El inicializador son llaves vacías (inicialización de valor)
```

1. ¿Qué son las variables locales?

Las variables locales son aquellas que se **declaran dentro de un bloque de código, como una función, un bucle o una estructura de control**. Solo son accesibles dentro de ese bloque y no pueden ser utilizadas fuera de él.

Características de las variables locales:

Alcance (scope): Limitado al bloque donde se declara.

Ciclo de vida: Existe solo mientras el control del programa está dentro de ese bloque. Cuando el bloque termina, la variable es destruida.

Visibilidad: Solo puede ser accedida dentro del bloque donde fue declarada.

Ejemplo de variable local:

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

void miFuncion() {
    int numero = 10; // Variable local de miFuncion
    cout << "El número es: " << numero << endl;
}

int main() {
    miFuncion();
    // cout << numero; // Error: 'numero' no está definido en main
    return 0;
}
```

En este ejemplo, la variable `numero` es local a `miFuncion` y no puede ser accedida desde `main`.

2. ¿Qué son las variables globales?

Las variables globales son aquellas que se **declaran fuera de todas las funciones** y están disponibles para todo el programa. Es decir, cualquier función puede acceder a una variable global.

Características de las variables globales:

Alcance (scope): Todo el archivo o programa, desde el punto donde se declara hasta el final del archivo.

Ciclo de vida: Existe durante toda la ejecución del programa.

Visibilidad: Accesible desde cualquier función del programa (salvo que haya un conflicto de nombres con una variable local).

Ejemplo de variable global:

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

int numero = 20; // Variable global

void miFuncion() {
    cout << "Variable global dentro de miFuncion: " << numero << endl;
}

int main() {
    cout << "Variable global en main: " << numero << endl;
    miFuncion();
    return 0;
}
```

En este ejemplo, la variable `numero` es global y puede ser utilizada tanto en `main` como en `miFuncion`.

3. Diferencias clave entre variables globales y locales

Característica	Variables Locales	Variables Globales
Alcance	Dentro del bloque donde se declaran.	En todo el programa.
Ciclo de vida	Se crea al entrar en el bloque y se destruye al salir.	Existe durante toda la ejecución del programa.
Visibilidad	Visible solo en el bloque donde se declara.	Visible para todas las funciones del programa.
Memoria	Se almacenan en el stack (pila).	Se almacenan en la sección de datos globales.

4. Implementación y uso adecuado

Variables locales: Cuándo y cómo usarlas

Las variables locales son ideales cuando necesitas un valor que solo será usado dentro de una función o bloque específico. Son más seguras porque no afectan otras partes del programa.

Ejemplo: Usar variables locales para cálculos internos:

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

void sumar() {
    int a = 5, b = 10; // Variables locales
    int resultado = a + b;
    cout << "El resultado de la suma es: " << resultado << endl;
}

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

Variables globales: Cuándo y cómo usarlas Las variables globales se usan cuando necesitas compartir un dato entre varias funciones.

Sin embargo, *su uso debe ser limitado porque:*

Pueden ser modificadas por cualquier función, lo que puede llevar a errores difíciles de detectar. Reducen la modularidad del programa.

Ejemplo: Usar una variable global para contar acciones:

```
#include <iostream>
using namespace std;
int contador = 0; // Variable global
void incrementar() { contador++; // Modifica la variable global }
int main() {
    incrementar();
    incrementar();
    cout << "El contador es: " << contador << endl; // Imprime: 2
    return 0;
}
```

5. Buenas prácticas

Usa variables locales siempre que sea posible:

Reduce la posibilidad de errores porque las variables están encapsuladas en su bloque.

Minimiza el uso de variables globales:

Si necesitas compartir datos entre funciones, considera usar parámetros de función o estructuras como clases y objetos.

Evita nombres conflictivos:

Una variable local con el mismo nombre que una global oculta la variable global dentro de ese bloque. Esto puede causar confusión.

Ejemplo de conflicto:

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

int numero = 100; // Variable global

void miFuncion() {
    int numero = 200; // Variable local que oculta la global
    cout << "Variable local: " << numero << endl;
}

int main() {
    cout << "Variable global: " << numero << endl;
    miFuncion();
    return 0;
}

Salida:
```

Variable global: 100

Variable local: 200

6. Combinación de variables globales y locales

A veces necesitas usar tanto variables globales como locales. En estos casos, puedes usar el operador de alcance `::` para referirte explícitamente a la variable global.

Ejemplo:

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

int numero = 50; // Variable global

void miFuncion() {
    int numero = 20; // Variable local
    cout << "Variable local: " << numero << endl;
    cout << "Variable global usando '::': " << ::numero << endl;
}

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

Salida:

```
Variable local: 20
Variable global usando '::': 50
```

Conversión de Variables a Constante:

En C++, puedes convertir una variable en una constante de diferentes maneras dependiendo de tu propósito y del momento en que quieras garantizar que su valor no pueda cambiar. Aquí están las formas más comunes de hacerlo:

1. Usar la palabra clave **const**

La forma más común de declarar una constante es usar **const** al declarar una variable. Una vez asignado un valor, no puede modificarse.

```
const int valor = 10; // Una constante de tipo entero
```

- **Características:**

- El valor debe asignarse al momento de la declaración.
- Intentar modificarlo provocará un error en tiempo de compilación.

Ejemplo:

```
#include <iostream>
int main() {
    const double pi = 3.14159; // Constante de tipo double
    // pi = 3.14; // Error: no se puede modificar una constante
    std::cout << "El valor de pi es: " << pi << std::endl;
    return 0;
}
```

2. Usar **constexpr** para constantes en tiempo de compilación

constexpr es una forma más estricta de definir constantes. Garantiza que el valor de la constante sea evaluado en tiempo de compilación.

```
constexpr int valor = 100;
```

- **Características:**

- Ideal para constantes que necesitas evaluar durante la compilación.
- Solo se pueden asignar valores que también sean constantes en tiempo de compilación.
- Es más eficiente, especialmente cuando se usa en expresiones complejas.

Ejemplo:

```
#include <iostream>
constexpr int cuadrado(int x) { return x * x; }

int main() {
    constexpr int valor = cuadrado(5); // Evaluado en tiempo de compilación
    std::cout << "El cuadrado de 5 es: " << valor << std::endl;
    return 0;
}
```

3. Usar **#define** para constantes preprocesadas

Otra manera de definir constantes es usar directivas del preprocesador con **#define**. Aunque es una técnica válida, es menos recomendada en comparación con **const** y **constexpr**.

```
#define VALOR 42
```

- **Características:**

- No tiene tipo, por lo que no se verifica en tiempo de compilación.
- Puede causar problemas de depuración y errores difíciles de rastrear.
- No respeta el alcance (**scope**) como las constantes definidas con **const** o **constexpr**.

Ejemplo:

```
#include <iostream>
#define PI 3.14159

int main() {
    std::cout << "El valor de PI es: " << PI << std::endl;
    return 0;
}
```

```
}
```

4. Usar `enum` para constantes enteras

`enum` se puede usar para definir constantes enteras relacionadas. Aunque no es su propósito principal, es una técnica válida.

```
enum { CONSTANTE = 10 };
```

- **Características:**

- Útil para valores relacionados (como días de la semana, estados, etc.).
- Todas las constantes declaradas en el `enum` son de tipo `int`.

Ejemplo:

```
#include <iostream>
int main() {
    enum { MAX_LIMITE = 100 };
    std::cout << "El límite máximo es: " << MAX_LIMITE << std::endl;
    return 0;
}
```

5. Declarar constantes en clases o estructuras

Puedes definir constantes como miembros de una clase o estructura. Para esto, puedes usar `static const` o `static constexpr`.

Ejemplo con `static const`:

```
#include <iostream>
class MiClase {
public:
    static const int constante = 42; // Constante estática
};

int main() {
    std::cout << "Constante en clase: " << MiClase::constante << std::endl;
    return 0;
}
```

Ejemplo con `static constexpr`:

```
#include <iostream>
class MiClase {
public:
    static constexpr double pi = 3.14159;
    // Constante estática evaluada en tiempo de compilación
};

int main() {
    std::cout << "El valor de pi es: " << MiClase::pi << std::endl;
    return 0;
}
```

Resumen Comparativo

Método	Tiempo de Evaluación	Tipo	Uso Recomendado
<code>const</code>	En tiempo de ejecución	Tipado	Para constantes típicas.
<code>constexpr</code>	En tiempo de compilación	Tipado	Para constantes evaluadas al compilar.
<code>#define</code>	En tiempo de preprocesado	No tipado	No recomendado (uso legado).
<code>enum</code>	En tiempo de ejecución	Entero	Constantes relacionadas.
<code>static const</code>	En tiempo de ejecución	Tipado	Constantes en clases o structs.
<code>static constexpr</code>	En tiempo de compilación	Tipado	Constantes de clases evaluadas en tiempo de compilación.

El uso (o NO) de Variables:

En C++, el atributo `[[maybe_unused]]` se introdujo en C++17 para indicar que una variable, función, parámetro, o cualquier otra entidad del programa, puede no ser utilizada en algunas circunstancias, y esto es intencional. Esto es útil porque, sin este atributo, los compiladores suelen emitir advertencias sobre variables o funciones declaradas pero no utilizadas, lo que a veces es necesario o deseado en ciertos contextos.

Sintaxis

`[[maybe_unused]] tipo nombre_variable;`

O también:

`[[maybe_unused]] tipo funcion(...);`

El atributo `[[maybe_unused]]` puede aplicarse a *variables locales, parámetros de funciones, funciones enteras y otros elementos*.

¿Cuándo usar `[[maybe_unused]]`?

- Variables locales:** Si declaras una variable que a veces es útil pero no siempre, puedes marcarla como `[[maybe_unused]]` para evitar advertencias del compilador.
- Parámetros de funciones:** A veces, un parámetro de una función puede no ser usado en todas las implementaciones de la función, especialmente en implementaciones de plantillas.
- Funciones:** Una función que puede no ser utilizada en ciertos contextos, pero que debe ser declarada para cumplir con un estándar o interfaz.

Ejemplo 1: Variables locales

```
#include <iostream>

int main() {
    [[maybe_unused]] int x = 42; // 'x' no se utiliza, pero no genera advertencia

    std::cout << "Hola, mundo" << std::endl;

    return 0;
}
```

En este caso, `x` no se utiliza dentro del programa. Si no se hubiera marcado con `[[maybe_unused]]`, el compilador podría emitir una advertencia. Pero con el atributo, le indicamos al compilador que no emita advertencias porque es intencional.

Ejemplo 2: Parámetros de función

```
#include <iostream>

void funcion([[maybe_unused]] int valor) {
    // No se usa 'valor' en la implementación, pero no queremos una advertencia
}
```

```

    std::cout << "Llamando a la función" << std::endl;
}
int main() {
    funcion(5);
    return 0;
}

```

Aquí, el parámetro valor de la función **función** no se usa dentro del cuerpo de la función, pero el atributo `[[maybe_unused]]` evita que el compilador emita una advertencia.

Ejemplo 3: Uso en funciones

```

[[maybe_unused]] void funcionNoUsada() {
    // Esta función no se utiliza, pero puede estar aquí por compatibilidad futura.
    std::cout << "Esta función no se usa" << std::endl;
}

int main() {
    std::cout << "Programa principal" << std::endl;
    return 0;
}

```

Aquí, la función **funcionNoUsada** no se utiliza en ninguna parte del programa, pero el atributo `[[maybe_unused]]` evita que el compilador genere una advertencia.

¿Por qué es útil?

Evita advertencias innecesarias: A veces, las advertencias de variables no utilizadas son solo ruido, especialmente en versiones de código que están en desarrollo o cuando una variable o función está ahí por razones específicas.

Mejora la legibilidad: El uso de `[[maybe_unused]]` indica claramente que no utilizar una variable es intencional y que el desarrollador ya es consciente de esto.

Compatibilidad con código heredado: En proyectos grandes, a veces el código tiene que cumplir con ciertas convenciones o contratos (interfaces, por ejemplo) que pueden implicar declarar variables o parámetros no utilizados.

La memoria RAM

La memoria RAM no necesariamente separa físicamente los datos según su tipo (como **int**, **double**, **float**, etc.), pero sí tiene un orden lógico que los programas siguen para organizarlos. A continuación, te lo explico paso a paso:

Organización de la memoria RAM

En un programa, la memoria RAM generalmente se divide en varias secciones lógicas para organizar los datos y el código. Estas son las principales:

Sección de código (Text segment):

Contiene las instrucciones del programa.
Es donde se almacena el código ejecutable.

Sección de datos estáticos (Data segment):

Aquí se guardan variables globales y estáticas, y se dividen a su vez en:

Datos inicializados: Variables globales o estáticas con un valor inicial definido, como `int x = 10;`.

Datos no inicializados (BSS): Variables globales o estáticas declaradas pero no inicializadas, como `int y;`.

Heap (Montículo):

Es usado para datos dinámicos, es decir, memoria asignada manualmente con **new** en C++ o funciones similares.

Ejemplo: `int* ptr = new int[10];`.

Stack (Pila):

Es usado para variables locales y datos temporales creados en las funciones.

Ejemplo: Cuando declaras `int numero = 20;` dentro de una función, se almacena aquí.

Se organiza de manera LIFO (Last In, First Out).

¿Se separan por tipo de dato?

No se separan físicamente por tipo de dato (`int`, `float`, etc.), pero el sistema sigue ciertas reglas:

Tamaño del dato y alineación: Cada tipo de dato ocupa un número específico de bytes y se almacena de manera alineada. Por ejemplo:

`int` (generalmente 4 bytes) puede empezar en una dirección divisible por 4.

`double` (8 bytes) puede necesitar alineación a 8 bytes.

Zona común: Variables locales (independientemente de su tipo) se almacenan en el **stack**, mientras que variables dinámicas van al **heap**.

Por ejemplo:

```
void ejemplo() {
    int numero = 20;        // Stack
    float pi = 3.14;        // Stack
    bool valor = true;      // Stack
    int* arreglo = new int[5]; // Heap
}
```

En este caso:

`numero`, `pi` y `valor` estarán en el **stack**, organizados según el orden de declaración y reglas de alineación. El `arreglo` estará en el **heap**.

La reserva de memoria

depende del tipo de variable que declares y del alcance de la misma. Vamos a analizarlo según el caso:

1. Variables globales y estáticas:

¿Cuándo se reserva la memoria?

Para estas variables, la memoria se reserva **durante la compilación** y se asigna cuando el programa comienza a ejecutarse (*en tiempo de carga*).

Ejemplo:

```
int globalVar; // Memoria reservada en la sección BSS(es una parte de un archivo de
objeto, ejecutable o código de lenguaje ensamblador que contiene variables asignadas
estáticamente, pero a las que aún no se les ha asignado un valor).
static float pi = 3.14; // Memoria reservada en la sección de datos inicializados.
```

Estas variables existen durante toda la ejecución del programa, ya que se colocan en la sección de datos estáticos de la memoria.

2. Variables locales (en funciones):

¿Cuándo se reserva la memoria?

Para variables locales (***declaradas dentro de una función***), la memoria se reserva en tiempo de ejecución cuando se llama a la función y se libera automáticamente al salir de ella.

Ejemplo:

```
void ejemplo() {
    int valor1; // Se reserva espacio en el stack al entrar a la función.
    float valor2; // También en el stack.
} // La memoria se libera al salir de la función.
```

Estas variables **no existen antes ni después de la ejecución de la función**.

3. Variables dinámicas (en el heap):

¿Cuándo se reserva la memoria?

En este caso, la memoria se asigna en **tiempo de ejecución** mediante el uso de operadores como `new` (en `C++`) o funciones como `malloc` (en `C`).

Ejemplo:

```
int* ptr = new int; // Se reserva espacio dinámicamente en el heap.
```

La memoria asignada dinámicamente permanece hasta que la liberes usando `delete` o `free`.

Respuesta:

Caso 1: Declaración global o estática

Si declaras:

```
int valor1;
float valor2;
... como variables globales o estáticas, la memoria se reserva con la compilación, porque estas variables se guardan en la sección de datos del programa.
```

Caso 2: Declaración local

Si declaras las mismas variables dentro de una función:

```
void ejemplo() {
    int valor1;
    float valor2;
}
... la memoria no se reserva con la compilación, sino en tiempo de ejecución cuando la función es llamada. Esto ocurre porque las variables locales viven en el stack y solo existen durante la ejecución de la función.
```

Nota:

Cuando declaras un **int** valor (*que ocupa 32 bits o 4 bytes*) y le asignas un valor como 3, efectivamente no estás utilizando todos los bits disponibles para ese tipo de dato. Sin embargo, esto no se considera necesariamente un desperdicio, y a continuación te explico por qué:

1. ¿Por qué **int** ocupa siempre 32 bits (4 bytes)?

El tamaño de un tipo de dato, como **int**, está definido por el compilador y la arquitectura del sistema (*32 o 64 bits, por ejemplo*).

Esto permite un acceso eficiente a la memoria, ya que las operaciones con datos de tamaño fijo son más rápidas y fáciles de manejar para el procesador.

Aunque el valor 3 necesita solo unos pocos bits (*en binario: 00000011*), la reserva de 32 bits se hace por razones de alineación y compatibilidad.

2. ¿Qué pasa con los bits no utilizados?

Cuando asignas **int valor = 3;**:

Se reservan **32 bits** para la variable, porque ese es el tamaño fijo de **int**. El número 3 se almacena en binario como:

```
00000000 00000000 00000000 00000011
```

Los bits restantes (*los ceros de más a la izquierda*) no se "desperdician"; simplemente forman parte del espacio asignado al **int**. Es decir, los 32 bits son tratados como un bloque indivisible.

3. ¿Es esto un desperdicio de memoria?

No necesariamente, porque:

El uso de tipos de tamaño fijo como **int** permite que las operaciones aritméticas, comparaciones y transferencias de datos se realicen de manera uniforme y eficiente. El costo de usar un poco más de memoria para cada variable es insignificante en comparación con los beneficios de rendimiento en sistemas modernos con suficiente RAM. Sin embargo, si trabajas en un entorno donde la memoria es limitada (*como sistemas embebidos*), puedes optar por tipos más pequeños, como:

```
int8_t (1 byte, 8 bits): Valores de -128 a 127.
int16_t (2 bytes, 16 bits): Valores de -32,768 a 32,767.
```

Ejemplo:

```
#include <stdint>
int8_t pequeño = 3; // Solo usa 8 bits.
```

NOTA: Dentro de **#include <stdint>**, encontrarás tipos de enteros con tamaños fijos como:

`int8_t` y `uint8_t` para enteros de **8 bits** con y sin signo, respectivamente.

`int16_t` y `uint16_t` para enteros de **16 bits**.

`int32_t` y `uint32_t` para enteros de **32 bits**.

`int64_t` y `uint64_t` para enteros de **64 bits**.

4. ¿Qué puedes hacer para evitar el "desperdicio"?

En sistemas donde optimizar la memoria es clave:

Usa tipos de datos más pequeños (`int8_t`, `uint8_t`, etc.) en lugar de `int` si sabes que no necesitas valores grandes.

Usa estructuras compactas para agrupar datos pequeños:

```
struct Compacto {
    int8_t a; // 1 byte
    int8_t b; // 1 byte
}; // Total: 2 bytes en lugar de 8 si fueran ints normales.
```

Usa **bitfields** en estructuras para usar solo los bits necesarios:

```
struct Flags {
    unsigned int flag1 : 1; // Solo 1 bit
    unsigned int flag2 : 1; // Solo 1 bit
}; // Total: 2 bits en lugar de 2 bytes.
```

5. Caso práctico

Supongamos que necesitas almacenar solo números pequeños (*como en un contador de 0 a 255*):

Usar `int` desperdicia memoria, porque ocupa 4 bytes.

Usar `uint8_t` (*8 bits sin signo*) es más eficiente:

```
uint8_t contador = 3; // Solo usa 1 byte.
```

Tiempo de Ejecución

El *tiempo de ejecución*, o "runtime" en inglés, se refiere al período durante el cual un programa de computadora está siendo ejecutado. En otras palabras, es el tiempo desde que el programa comienza a ejecutarse hasta que termina. Aquí hay algunos puntos clave sobre el tiempo de ejecución:

Conceptos Clave:

Inicio del Programa: El tiempo de ejecución comienza cuando el programa es lanzado. Esto puede ser al hacer clic en un icono, ejecutar un comando en la terminal, etc.

Ejecución de Instrucciones: Durante el tiempo de ejecución, el programa lleva a cabo todas las instrucciones escritas en su código. Esto incluye la lectura de entradas del usuario, procesamiento de datos, y generación de salidas.

Gestión de Recursos: El sistema operativo asigna recursos como memoria, tiempo de CPU y acceso a dispositivos de entrada/salida al programa durante su tiempo de ejecución.

Errores en Tiempo de Ejecución: Son problemas que ocurren mientras el programa está en funcionamiento, como intentos de dividir por cero, acceso a memoria no válida, etc. Estos errores suelen detener la ejecución del programa y pueden requerir depuración para solucionarlos.

Diferencias con Tiempo de Compilación: A diferencia del tiempo de ejecución, el tiempo de compilación es el período durante el cual el código fuente del programa es traducido a código máquina por un compilador. Los errores en tiempo de compilación son detectados y corregidos antes de que el programa se ejecute.

Ejemplo:

Imagina que tienes un programa en C++ que calcula la suma de dos números. El tiempo de

ejecución es el período en que el programa está recibiendo los números, sumándolos y mostrando el resultado.

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

int main() {
    int a, b;
    cout << "Ingrese dos números: ";
    cin >> a >> b;
    int suma = a + b;
    cout << "La suma es: " << suma << endl;
    return 0;
}
```

En este ejemplo:

Inicio del tiempo de ejecución: Cuando ejecutas el programa.

Ejecución de instrucciones: Cuando el programa recibe los números, los suma y muestra el resultado.

Fin del tiempo de ejecución: Cuando el programa termina y devuelve el control al sistema operativo.

Tiempo de Compilación

El **tiempo de compilación**, o "*compilation time*" en inglés, es el periodo durante el cual el código fuente de un programa se traduce a código máquina o código ejecutable por un compilador. Este proceso ocurre antes de que el programa pueda ser ejecutado en una computadora. Aquí tienes algunos puntos clave sobre el tiempo de compilación:

Conceptos Clave:

Traducción del código: Durante el tiempo de compilación, el compilador toma el código fuente escrito en un lenguaje de alto nivel (como **C++**, **Java**, **etc.**) y lo convierte en código máquina que puede ser entendido y ejecutado por la CPU.

Detección de errores: El compilador verifica el código fuente en busca de errores sintácticos y semánticos. Estos errores deben corregirse antes de que el programa pueda ser ejecutado. Los errores de compilación incluyen problemas como nombres de variables incorrectos, errores de tipo, declaraciones faltantes, etc.

Generación de ejecutable: Si el código fuente es correcto y no tiene errores, el compilador genera un archivo ejecutable o **bytecode**, dependiendo del lenguaje de programación. Este archivo es lo que finalmente se ejecuta en la computadora.

Optimización: Algunos compiladores realizan optimizaciones durante el tiempo de compilación para mejorar el rendimiento del código ejecutable resultante. Esto puede incluir la eliminación de código innecesario, la reorganización de instrucciones, etc.

Diferencias con Tiempo de Ejecución:

Tiempo de compilación: Ocurre antes de que el programa se ejecute y se enfoca en la traducción del código fuente y la detección de errores.

Tiempo de ejecución: Es cuando el programa se está ejecutando en la computadora, realizando las tareas para las que fue diseñado.

Ejemplo en C++:

Supongamos que tienes el siguiente código fuente en C++:

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

int main() {
    cout << "Hola, mundo!" << endl;
    return 0;
}
```

Proceso de Compilación:

Escribiendo el código fuente: Escribes el código en un archivo, por ejemplo, `holamundo.cpp`.

Compilación: Usas un compilador como `g++` para compilar el código:
`g++ holamundo.cpp -o holamundo`

Durante este proceso, el compilador traducirá `holamundo.cpp` a un archivo ejecutable llamado `holamundo`.

Generación de ejecutable: Si no hay errores, se generará el archivo ejecutable `holamundo`.

Ejecución del Programa:

Una vez que tienes el archivo ejecutable, puedes ejecutarlo en la terminal:

```
./holamundo
```

Esto inicia el tiempo de ejecución, durante el cual el programa imprime "**Hola, mundo!**" en la pantalla.

¿Qué es el operador ternario?

El operador ternario tiene la forma:

condición ? expresión1 : expresión2;

Componentes:

condición: Es una expresión booleana (que se evalúa como **true** o **false**).

?: Indica el inicio del operador ternario.

expresión1: Valor que se devuelve o ejecuta si la condición es **true**.

: → Separa las dos expresiones.

expresión2: Valor que se devuelve o ejecuta si la condición es **false**.

¿Cómo funciona?

El operador ternario evalúa la condición:

Si la condición es **true**, ejecuta expresión1.

Si la condición es **false**, ejecuta expresión2.

Ejemplo básico

Sin operador ternario:

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

int main() {
    int a = 10, b = 20;
    int mayor;

    if (a > b) {
        mayor = a;
    } else {
        mayor = b;
    }

    cout << "El número mayor es: " << mayor << endl;

    return 0;
}
```

Con operador ternario:

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

int main() {
    int a = 10, b = 20;

    int mayor = (a > b) ? a : b;

    cout << "El número mayor es: " << mayor << endl;

    return 0;
}
```

Salida en ambos casos:

El número mayor es: 20

Ventajas del operador ternario

Menor cantidad de código: Es compacto y fácil de leer para condiciones simples.

Ideal para asignaciones: Permite decidir el valor de una variable en una sola línea.

Ejemplo con strings

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

int main() {
    int edad;
    cout << "Ingresa tu edad: ";
```

```
    cin >> edad;

    string categoria = (edad >= 18) ? "Adulto" : "Menor";

    cout << "Eres: " << categoria << endl;

    return 0;
}
```

Salida 1 (si ingresas 20):

Eres: Adulto

Salida 2 (si ingresas 15):

Eres: Menor

Ejemplo con funciones

El operador ternario también se puede usar para invocar funciones.

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

void mensajePositivo() {
    cout << "El número es positivo." << endl;
}

void mensajeNegativo() {
    cout << "El número es negativo o cero." << endl;
}

int main() {
    int num;
    cout << "Ingresa un número: ";
    cin >> num;

    (num > 0) ? mensajePositivo() : mensajeNegativo();

    return 0;
}
```

Salida 1 (si ingresas 5):

El número es positivo.

Salida 2 (si ingresas -3):

El número es negativo o cero.

Limitaciones del operador ternario

No reemplaza estructuras complejas: Para condiciones con múltiples ramas, usa **if-else**.

Menor legibilidad en casos complejos: Evita usar el operador ternario en expresiones largas o anidadas.

Ejemplo complicado (difícil de leer):

```
int resultado = (a > b) ? ((b > c) ? b : c) : a;
```

Equivalente con **if-else** (más claro):

```
if (a > b) {
    if (b > c) {
        resultado = b;
    } else {
        resultado = c;
    }
} else {
    resultado = a;
}
```

Resumen

El operador ternario es una forma abreviada de escribir condiciones simples.

Sintaxis: *condición ? expresión1 : expresión2.*

Útil para asignaciones rápidas o llamadas simples a funciones.

Evítalo en condiciones complejas para no afectar la legibilidad.

b. Tipos de Datos:

Datos primitivos:	Datos no primitivos
<ul style="list-style-type: none">• char: Almacena un solo carácter (Ej. char letra = 'A';).Tamaño 1 byte;• wchar_t 2 o 4 bytes Representa un carácter ancho (wide character)• int: Almacena números enteros (Ej. int x = 5;).Tamaño 4 byte;• long: Para números más grandes. Tamaño 4 bytes (32-bit) o 8 bytes (64-bit).• long long 8 bytes Entero muy largo.• float: Almacena números en coma flotante de precisión simple (Ej. float x = 5.75f;).Tamaño 4 byte;• double: Almacena números en coma flotante de doble precisión (Ej. double x = 19.99;). Tamaño 1 byte.• long double 8, 12 o 16 bytes Número de punto flotante de precisión extendida (dependiendo del compilador).• unsigned int: Solo números positivos• bool: Almacena valores booleanos (true o false). Tamaño 1 byte;• <i>char16_t y char32_t son tipos de datos primitivos en C++. Fueron introducidos en el estándar C++11 para proporcionar soporte para caracteres de 16 y 32 bits, respectivamente.</i>• char16_t se utiliza para representar caracteres en codificación UTF-16.• char32_t se utiliza para representar caracteres en codificación UTF-32.	<ul style="list-style-type: none">• Arreglos (Arrays): Colección de elementos del mismo tipo.• Estructuras (Structs): Agrupación de variables bajo un mismo nombre.• Clases (Classes): Blueprint para crear objetos con atributos y métodos.• Cadenas de caracteres (Strings): En C++ se puede usar std::string.• Contenedores de la STL (Standard Template Library): Incluyen vectores, listas, mapas, etc.• Apuntadores (Pointers): Variables que almacenan direcciones de memoria.• Enumeraciones (Enums): Conjunto de constantes con nombre.

Operadores de Incremento y Decremento

Los operadores de incremento (++) y decremento (--) se utilizan para modificar el valor de una variable numérica, incrementándola o decrementándola en 1.

1. Incremento (++)

Este operador suma 1 al valor de la variable. Puede usarse de dos formas:

Prefijo (++x): Incrementa la variable **antes** de usarla.
Postfijo (x++): Incrementa la variable **después** de usarla.

Ejemplo: Incremento

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

int main() {
    int x = 5;

    cout << "Valor inicial de x: " << x << endl;

    // Incremento en prefijo
    cout << "Prefijo (++x): " << ++x << endl; // Incrementa y luego muestra

    // Incremento en postfijo
    cout << "Postfijo (x++): " << x++ << endl; // Muestra y luego incrementa

    cout << "Valor de x después del postfijo: " << x << endl;

    return 0;
}
```

Salida:

Valor inicial de x: 5
Prefijo (++x): 6
Postfijo (x++): 6
Valor de x después del postfijo: 7

2. Decremento (--)

Este operador resta 1 al valor de la variable. También tiene dos formas:

- Prefijo (--x):** Decrementa la variable antes de usarla.
- Postfijo (x--):** Decrementa la variable después de usarla.

Ejemplo: Decremento

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

int main() {
    int y = 10;

    cout << "Valor inicial de y: " << y << endl;

    // Decremento en prefijo
    cout << "Prefijo (--y): " << --y << endl; // Decrementa y luego muestra

    // Decremento en postfijo
    cout << "Postfijo (y--): " << y-- << endl; // Muestra y luego decrementa

    cout << "Valor de y después del postfijo: " << y << endl;

    return 0;
}
```

Salida:

Valor inicial de y: 10
Prefijo (--y): 9
Postfijo (y--): 9
Valor de y después del postfijo: 8

Diferencias entre Prefijo y Postfijo

Aspecto	Prefijo (++x / --x)	Postfijo (x++ / x--)
Orden de operación	Modifica la variable antes de usarla.	Usa la variable y luego la modifica.
Uso en expresiones	Cambia el valor inmediatamente.	El valor original se utiliza primero.

Ejemplo con expresiones:

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

int main() {
    int a = 3, b = 3;

    int resultadoPrefijo = ++a * 2; // Incrementa primero, luego multiplica
    int resultadoPostfijo = b++ * 2; // Multiplica primero, luego incrementa

    cout << "Resultado Prefijo (++a * 2): " << resultadoPrefijo << endl; // 8
    cout << "Resultado Postfijo (b++ * 2): " << resultadoPostfijo << endl; // 6

    return 0;
}
```

Operadores Lógicos

Los operadores lógicos son usados para evaluar expresiones lógicas y combinarlas. Devuelven valores booleanos (true o false).

Tipos de Operadores Lógicos.

Operador	Nombre	Ejemplo	Descripción
&&	AND lógico	a && b	Devuelve true si ambas condiciones son true .
			OR lógico
!	NOT lógico	!a	Invierte el valor lógico (true → false).

1. AND Lógico (&&)

Este operador devuelve **true** solo si ambas expresiones son verdaderas.

Ejemplo:

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

int main() {
    int x = 5, y = 10;

    if (x > 0 && y > 0) {
        cout << "Ambos números son positivos" << endl;
    } else {
        cout << "Al menos uno de los números no es positivo" << endl;
    }

    return 0;
}
```

Salida:

Ambos números son positivos

2. OR Lógico (||)

Este operador devuelve **true** si al menos una de las expresiones es verdadera.

Ejemplo:

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

int main() {
    int x = -5, y = 10;

    if (x > 0 || y > 0) {
        cout << "Al menos uno de los números es positivo" << endl;
    } else {
        cout << "Ningún número es positivo" << endl;
    }

    return 0;
}
```

Salida:

Al menos uno de los números es positivo

3. NOT Lógico (!)

Este operador invierte el valor lógico de la expresión. Si es true, devuelve false y viceversa.

Ejemplo:

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

int main() {
    bool esVerdad = true;

    cout << "Valor original: " << esVerdad << endl;
    cout << "Valor invertido: " << !esVerdad << endl;

    return 0;
}
```

Salida:

Valor original: 1
Valor invertido: 0

Combinar Operadores Lógicos

Puedes combinar múltiples operadores para evaluar expresiones más complejas.

Ejemplo:

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

```
int main() {
    int x = 5, y = 10, z = -3;

    if ((x > 0 && y > 0) || z > 0) {
        cout << "Se cumple al menos una condición" << endl;
    } else {
        cout << "Ninguna condición se cumple" << endl;
    }

    return 0;
}
```

Precedencia de los Operadores Lógicos

- **! (NOT lógico) tiene la mayor precedencia.**
- **&& (AND lógico) tiene precedencia media.**
- **|| (OR lógico) tiene la menor precedencia.**

Usa paréntesis para controlar el orden de evaluación y mejorar la legibilidad.

Resumen

- *Incremento y Decremento (++ , --):*
- *Incrementan o decrementan en 1.*
- *Prefijo modifica antes, postfijo después.*
- *Operadores Lógicos (&&, ||, !):*
- *Usados para evaluar y combinar expresiones lógicas.*
- *Devuelven valores booleanos (true o false).*

Operadores de asignación en C++

Los operadores de asignación se utilizan para asignar valores a las variables. El operador más básico es **= (asignación simple)**, pero hay otros operadores compuestos que combinan una operación aritmética o lógica con una asignación.

1. Asignación simple (=)

Este operador asigna el valor del lado derecho a la variable del lado izquierdo.

```
#include <iostream>

int main() {
    int x = 10; // Asigna 10 a x
    std::cout << "x = " << x << std::endl;
    return 0;
}
```

2. Asignación con suma (+=)

Suma el valor de la derecha a la variable de la izquierda y asigna el resultado.

```
#include <iostream>

int main() {
    int x = 10;
    x += 5; // Equivalente a: x = x + 5
    std::cout << "x = " << x << std::endl; // Resultado: 15
    return 0;
}
```

3. Asignación con resta (--=)

Resta el valor de la derecha de la variable de la izquierda y asigna el resultado.

```
#include <iostream>

int main() {
    int x = 20;
    x -= 5; // Equivalente a: x = x - 5
    std::cout << "x = " << x << std::endl; // Resultado: 15
    return 0;
}
```

4. Asignación con multiplicación (*=)

Multiplica la variable de la izquierda por el valor de la derecha y asigna el resultado.

```
#include <iostream>

int main() {
    int x = 4;
```

```

    x *= 3; // Equivalente a: x = x * 3
    std::cout << "x = " << x << std::endl; // Resultado: 12
    return 0;
}

```

5. Asignación con división (/=)

Divide la variable de la izquierda por el valor de la derecha y asigna el resultado.

```

#include <iostream>

int main() {
    int x = 20;
    x /= 4; // Equivalente a: x = x / 4
    std::cout << "x = " << x << std::endl; // Resultado: 5
    return 0;
}

```

6. Asignación con módulo (%=)

Calcula el resto de la división de la izquierda entre la derecha y asigna el resultado.

```

#include <iostream>

int main() {
    int x = 23;
    x %= 5; // Equivalente a: x = x % 5
    std::cout << "x = " << x << std::endl; // Resultado: 3
    return 0;
}

```

7. Asignación con desplazamiento a la izquierda (<<=)

Desplaza los bits de la variable de la izquierda hacia la izquierda por un número de posiciones especificado por el valor de la derecha.

```

#include <iostream>

int main() {
    int x = 5; // Binario: 0101
    x <<= 1; // Equivalente a: x = x << 1 (Desplaza un bit a la izquierda)
    std::cout << "x = " << x << std::endl; // Resultado: 10 (Binario: 1010)
    return 0;
}

```

8. Asignación con desplazamiento a la derecha (>>=)

Desplaza los bits de la variable de la izquierda hacia la derecha por un número de posiciones especificado por el valor de la derecha.

```

#include <iostream>

int main() {
    int x = 20; // Binario: 10100
    x >>= 2; // Equivalente a: x = x >> 2 (Desplaza dos bits a la derecha)
    std::cout << "x = " << x << std::endl; // Resultado: 5 (Binario: 0101)
    return 0;
}

```

9. Asignación con AND bit a bit (&=)

Realiza una operación AND bit a bit entre la variable de la izquierda y el valor de la derecha.

```

#include <iostream>

int main() {
    int x = 6; // Binario: 0110
    x &= 3; // Equivalente a: x = x & 3 (Binario: 0011)
    std::cout << "x = " << x << std::endl; // Resultado: 2 (Binario: 0010)
    return 0;
}

```

10. Asignación con OR bit a bit (|=)

Realiza una operación OR bit a bit entre la variable de la izquierda y el valor de la derecha.

```

#include <iostream>

int main() {
    int x = 6; // Binario: 0110
    x |= 3; // Equivalente a: x = x | 3 (Binario: 0011)
    std::cout << "x = " << x << std::endl; // Resultado: 7 (Binario: 0111)
    return 0;
}

```

11. Asignación con XOR bit a bit (^=)

Realiza una operación XOR bit a bit entre la variable de la izquierda y el valor de la derecha.

```
#include <iostream>

int main() {
    int x = 6; // Binario: 0110
    x ^= 3;    // Equivalente a: x = x ^ 3 (Binario: 0011)
    std::cout << "x = " << x << std::endl; // Resultado: 5 (Binario: 0101)
    return 0;
}
```

Resumen de los operadores compuestos:

Operador	Descripción	Ejemplo
=	Asignación simple	x = 10
+=	Asignación con suma	x += 5
-=	Asignación con resta	x -= 3
*=	Asignación con multiplicación	x *= 2
/=	Asignación con división	x /= 4
%=	Asignación con módulo	x %= 2
<<=	Desplazamiento a la izquierda	x <<= 1
>>=	Desplazamiento a la derecha	x >>= 2
&=	Asignación con AND bit a bit	x &= y
'	='	Asignación con OR bit a bit
^=	Asignación con XOR bit a bit	x ^= y

Caracteres de Escape

Carácter de Escape	Significado	Ejemplo
<code>\n</code>	Nueva línea	<code>std::cout << "Hola\nMundo";</code>
<code>\t</code>	Tabulación horizontal	<code>std::cout << "Hola\tMundo";</code>
<code>\r</code>	Retorno de carro	<code>std::cout << "Hola\rMundo";</code>
<code>\b</code>	Retroceso	<code>std::cout << "Hola\bMundo";</code>
<code>\f</code>	Salto de página	<code>std::cout << "Hola\fMundo";</code>
<code>\v</code>	Tabulación vertical	<code>std::cout << "Hola\vMundo";</code>
<code>\\</code>	Barra invertida (<code>\</code>)	<code>std::cout << "Hola\\Mundo";</code>
<code>\'</code>	Comilla simple (<code>'</code>)	<code>std::cout << "Hola\'Mundo";</code>
<code>\"</code>	Comilla doble (<code>"</code>)	<code>std::cout << "Hola\"Mundo";</code>
<code>\?</code>	Signo de interrogación (<code>?</code>)	<code>std::cout << "Hola?Mundo";</code>
<code>\0</code>	Caracter nulo (fin de cadena)	Usado internamente en cadenas terminadas en <code>\0</code> .
<code>\a</code>	Alerta sonora (beep)	<code>std::cout << "\a"; // Hace un beep</code>

Detalles Importantes:

`\\`: Se utiliza para imprimir la barra invertida en pantalla, ya que la barra invertida por sí sola tiene un significado especial como carácter de escape.

`\'` y `\"`: Son útiles para incluir comillas dentro de cadenas. Por ejemplo:

```
std::cout << "Ella dijo: \"Hola Mundo\"";
```

`'\0'`: Es un carácter especial que marca el final de una cadena en C++ al trabajar con arreglos de caracteres (**tipo `char[]`**).

`'\a'`: Puede no funcionar en todos los sistemas, dependiendo del soporte para sonidos de alerta.

`'\r'`: Es menos común en C++, pero mueve el cursor al inicio de la línea actual, sobrescribiendo lo que estaba previamente.

Ejemplo Práctico

```
#include <iostream>

int main() {
    std::cout << "Hola\nMundo\n";    // Salto de línea
    std::cout << "Hola\tMundo\n";    // Tabulación
    std::cout << "Hola\\Mundo\n";    // Barra invertida
    std::cout << "Hola\"Mundo\"\\n"; // Comillas dobles
    std::cout << "Hola\bMundo\n";    // Retroceso (Borra 'a')

    return 0;
}
```

DATOS TIPO CHAR

Para trabajar con el tipo `char` en C++, puedes usar tanto operaciones directas sobre variables de tipo `char` como funciones de la biblioteca estándar de C++ que te ayudarán a manipular caracteres y cadenas de caracteres.

1. Librería necesaria:

Para funciones relacionadas con el tipo `char` y cadenas de caracteres, puedes utilizar principalmente la librería:

```
#include <iostream> // Para entrada y salida estándar
#include <cstring>   // Para funciones de manipulación de cadenas estilo C
#include <cctype>    // Para manipulación de caracteres individuales (tipo char)
```

2. Comandos y funciones:

Función	Descripción	Ejemplo
<code>isalnum(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es alfanumérico (letra o dígito).	<code>isalnum('A') → true</code> , <code>isalnum('%') → false</code>
<code>isalpha(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es una letra (mayúscula o minúscula).	<code>isalpha('A') → true</code> , <code>isalpha('1') → false</code>
<code>isdigit(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es un dígito (0-9).	<code>isdigit('5') → true</code> , <code>isdigit('A') → false</code>
<code>isxdigit(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es un dígito hexadecimal (0-9, A-F, a-f).	<code>isxdigit('F') → true</code> , <code>isxdigit('G') → false</code>
<code>islower(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es una letra minúscula (a-z).	<code>islower('a') → true</code> , <code>islower('A') → false</code>
<code>isupper(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es una letra mayúscula (A-Z).	<code>isupper('A') → true</code> , <code>isupper('a') → false</code>
<code>isprint(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es imprimible (incluye letras, dígitos y símbolos).	<code>isprint(' ') → true</code> , <code>isprint('\n') → false</code>
<code>isgraph(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es imprimible y no es un espacio (' ').	<code>isgraph('A') → true</code> , <code>isgraph(' ') → false</code>
<code>isspace(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es un espacio en blanco (como ' ', \t, \n, \f, etc.).	<code>isspace(' ') → true</code> , <code>isspace('A') → false</code>
<code>ispunct(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es un signo de puntuación o símbolo.	<code>ispunct('!') → true</code> , <code>ispunct('A') → false</code>
<code>isctrl(c)</code>	Devuelve <code>true</code> si el carácter <code>c</code> es un carácter de control (como \n, \t, \0, etc.).	<code>isctrl('\n') → true</code> , <code>isctrl('A') → false</code>
<code>tolower(c)</code>	Convierte un carácter a minúscula, si es posible.	<code>tolower('A') → 'a'</code> , <code>tolower('a') → 'a'</code>
<code>toupper(c)</code>	Convierte un carácter a mayúscula, si es posible.	<code>toupper('a') → 'A'</code> , <code>toupper('A') → 'A'</code>

Notas Importantes

Tipo de argumento: Todas estas funciones reciben un único carácter de tipo `int`. Es común pasar valores `char`, ya que se convierten implícitamente a `int`.

Rango válido:

Los caracteres pasados deben estar en el rango representado por un `unsigned char` o el valor especial `EOF`.-(end of file)
Si se pasa un valor fuera de este rango, el comportamiento no está definido.

Ejemplo Práctico:

```
#include <iostream>
#include <cctype>

int main() {
    char c = 'A';

    if (isalpha(c)) {
        std::cout << c << " es una letra." << std::endl;
        if (islower(c))
            std::cout << c << " es minúscula." << std::endl;
        else if (isupper(c))
            std::cout << c << " es mayúscula." << std::endl;
    } else if (isdigit(c)) {
        std::cout << c << " es un dígito." << std::endl;
    } else if (isspace(c)) {
        std::cout << c << " es un espacio en blanco." << std::endl;
    } else {
        std::cout << c << " es un carácter especial." << std::endl;
    }
    // Conversión de mayúscula a minúscula
    char lower = tolower(c);
    std::cout << "Minúscula de " << c << ": " << lower << std::endl;

    return 0;
}
```

Salida:

A es una letra.
A es mayúscula.
Minúscula de A: a

Casos Prácticos

Validar entradas de usuario: Asegúrate de que los caracteres ingresados sean válidos (alfabéticos, numéricos, etc.).

Análisis de texto: Detecta letras, dígitos, espacios en blanco, etc.

Conversión de texto: Convierte caracteres a mayúsculas o minúsculas.

a. Convertir un carácter a mayúscula o minúscula:
Utiliza las funciones de la librería <cctype>.

Convertir a mayúscula:

```
char letra = 'a';  
letra = toupper(letra); // Convierte 'a' a 'A'
```

Convertir a minúscula:

```
char letra = 'A';  
letra = tolower(letra); // Convierte 'A' a 'a'
```

/*Conversión de una palabra completa*/

```
cout << "\n\nconversion con un texto mas largo\n";  
char txt[10];  
cout << "Digite una palabra en MAYUSCULAS: ";  
cin >> txt;  
  
for(int i = 0; txt[i]; i++)  
{  
    cout << (txt[i] = tolower(txt[i]));  
}
```

*el mismo modo para **minúsculas** o **mayúsculas** cambiando el comando.

b. Comprobar si un carácter es una letra, número, o símbolo:
Estas funciones también están en <cctype>.

Es letra:

```
char letra = 'A';  
if (isalpha(letra)) {  
    std::cout << "Es una letra.\n";  
}
```

Es dígito:

```
char digito = '5';  
if (isdigit(digito)) {  
    std::cout << "Es un número.\n";  
}
```

Es un espacio en blanco:

```
char espacio = ' ';  
if (isspace(espacio)) {  
    std::cout << "Es un espacio en blanco.\n";  
}
```


ESPACIOS (en blanco)

```
#include <iostream>
#include <cctype> // Para isspace

using namespace std;

int main() {
    cout << "Digite la barra espaciadora: ";
    char espacio;
    cin >> noskipws >> espacio; // Leer incluso espacios, si marcara error usar std::noskipws

    if (espacio == ' ') { // Comparar directamente con el carácter espacio
        cout << "Digitó la barra espaciadora..." << endl;
    } else {
        cout << "No presionó la barra espaciadora..." << endl;
    }

    return 0;
}
```

Cambios y detalles importantes:

Uso de **isspace**:

```
isspace verifica cualquier carácter considerado "espacio en blanco", como:
Espacio (' ').
Tabulación ('\t').
Nueva línea ('\n'), entre otros.
```

En tu caso, quieres verificar exclusivamente si el carácter es la barra espaciadora (' '). Por eso, usamos directamente la comparación espacio == ' '.

noskipws en **cin**:

Por defecto, **cin** ignora los espacios en blanco al leer caracteres.

El manipulador **noskipws** evita que **cin** salte caracteres en blanco (como el espacio). Sin esto, el programa no reconocerá correctamente la barra espaciadora como entrada.

Digite la barra espaciadora:
(El usuario presiona la barra espaciadora).

Digitó la barra espaciadora...
Entrada:

Digite la barra espaciadora: A

(El usuario ingresa la letra A).

Salida:
No presionó la barra espaciadora...

Diferencia entre noskipws y isspace

noskipws:

Es un manipulador de entrada que **evita** que **cin** ignore caracteres de espacio en blanco (como espacios, tabulaciones o saltos de línea). Esto es útil cuando deseas leer y procesar directamente espacios u otros caracteres que, de otro modo, serían saltados por **cin**.

```
Ejemplo de uso con cin:
char c;
cin >> noskipws >> c; // Leerá incluso espacios y otros caracteres de espacio en blanco
```

isspace:

Es una función que verifica si un carácter es un "carácter de espacio en blanco" (esto incluye espacio, tabulación, nueva línea, etc.). No es necesario usarla para verificar exclusivamente si un carácter es un espacio normal (' '); en ese caso, simplemente comparas con ' '.
Ejemplo de uso:

```
char c = ' ';
if (isspace(c)) {
    cout << "Es un carácter de espacio en blanco." << endl;
}
```

Casos donde `isspace` devuelve verdadero:

- Espacio normal (' ')
- Tabulación ('\t')
- Nueva línea ('\n')
- Retorno de carro ('\r')
- Form feed ('\f')
- Vertical tab ('\v')

¿Cuándo usar cada uno?

Usa `noskipws` si quieres leer y procesar caracteres de espacio en blanco desde la entrada:

Ejemplo: *Si el usuario debe presionar específicamente la barra espaciadora, necesitas capturar espacios con `noskipws` porque, de lo contrario, `cin` los ignorará.*
Usa `isspace` cuando quieras verificar si un carácter pertenece al grupo de "espacios en blanco":

Ejemplo: *Si procesas texto (ya capturado o definido en el código) y quieres determinar si un carácter es espacio, tabulación o nueva línea.*

Ejemplo combinado:

Si quieres un programa donde captures un carácter y verifiques si es un espacio específico o cualquier espacio en blanco, lo harías así:

```
#include <iostream>
#include <cctype> // Para isspace

using namespace std;

int main() {
    cout << "Digite un carácter (incluidos espacios): ";
    char c;
    cin >> noskipws >> c; // Leer espacios también

    if (c == ' ') {
        cout << "Específicamente presionó la barra espaciadora." << endl;
    } else if (isspace(c)) {
        cout << "Es un carácter de espacio en blanco (tabulación, nueva línea, etc.)." << endl;
    } else {
        cout << "No es un carácter de espacio." << endl;
    }

    return 0;
}
```

Salida esperada:

Si el usuario presiona la barra espaciadora:

Digite un carácter (incluidos espacios):
Específicamente presionó la barra espaciadora.
*Si el usuario presiona **Enter** o ingresa una tabulación:*

Digite un carácter (incluidos espacios): [TAB]
Es un carácter de espacio en blanco (tabulación, nueva línea, etc.).
Si el usuario ingresa cualquier otro carácter:

Digite un carácter (incluidos espacios): A
No es un carácter de espacio.

En resumen:

Usa `noskipws` para capturar espacios desde la entrada.
Usa `isspace` para verificar si un carácter es un espacio en blanco en general.

c. Copiar cadenas de caracteres:

Para copiar cadenas de caracteres, puedes usar `strcpy` de la librería `<cstring>`.

```
char origen[] = "Hola";
char destino[20];
strcpy(destino, origen); // Copia "Hola" en destino
```

d. Comparar cadenas de caracteres:

La función `strcmp` de `<cstring>` te permite comparar cadenas de caracteres. Devuelve:

```
0 si son iguales,
valor negativo si la primera es menor,
valor positivo si la primera es mayor.
```

```
char cadena1[] = "Hola";
char cadena2[] = "Mundo";

if (strcmp(cadena1, cadena2) == 0) {
    std::cout << "Las cadenas son iguales.\n";
} else {
    std::cout << "Las cadenas son diferentes.\n";
}
```

e. Calcular la longitud de una cadena:

La función `strlen` te da la longitud de una cadena de caracteres.

```
char cadena[] = "Hola Mundo";
int longitud = strlen(cadena); // Longitud de la cadena
std::cout << "La longitud de la cadena es: " << longitud << "\n";
```

f. Concatenar cadenas:

Usa `strcat` para concatenar dos cadenas.

```
char cadena1[50] = "Hola";
char cadena2[] = " Mundo";

strcat(cadena1, cadena2); // Concatena "Mundo" a "Hola"
std::cout << cadena1 << "\n"; // Imprime "Hola Mundo"
```

g. Buscar un carácter en una cadena:

La función `strchr` te permite buscar un carácter dentro de una cadena. Retorna un puntero al primer lugar donde aparece el carácter, o `nullptr` si no lo encuentra.

```
char cadena[] = "Hola Mundo";
char *pos = strchr(cadena, 'M'); // Busca la 'M'

if (pos != nullptr) {
    std::cout << "Carácter encontrado en la posición: " << (pos - cadena) << "\n";
} else {
    std::cout << "Carácter no encontrado.\n";
}
```

h. Reemplazar caracteres en una cadena:

Puedes iterar por una cadena y cambiar los caracteres que desees.

```
char cadena[] = "Hola Mundo";

for (int i = 0; cadena[i] != '\0'; i++) {
    if (cadena[i] == 'o') {
        cadena[i] = 'x'; // Reemplaza 'o' por 'x'
    }
}

std::cout << cadena << "\n"; // Imprime "Hx!x Mundo"
```

i. Convertir de un carácter numérico a entero:

Si tienes un carácter que representa un número ('0' a '9'), puedes convertirlo a su valor entero restando '0'.

```
char digito = '5';
int valor = digito - '0'; // Convierte '5' a 5
```

3. Ejemplo completo:

Aquí tienes un ejemplo que utiliza varias de estas funciones:

```
#include <iostream>
#include <cstring>
#include <cctype>

int main() {
    char cadena1[] = "Hola";
    char cadena2[20];

    // Copiar cadena
    strcpy(cadena2, cadena1);
    std::cout << "Cadena copiada: " << cadena2 << "\n";

    // Convertir a mayúsculas
    for (int i = 0; cadena2[i] != '\0'; i++) {
        cadena2[i] = toupper(cadena2[i]);
    }
    std::cout << "Cadena en mayúsculas: " << cadena2 << "\n";

    // Comparar cadenas
    if (strcmp(cadena1, cadena2) == 0) {
        std::cout << "Las cadenas son iguales.\n";
    } else {
        std::cout << "Las cadenas son diferentes.\n";
    }

    // Longitud de cadena
    std::cout << "Longitud de la cadena1: " << strlen(cadena1) << "\n";

    return 0;
}
```

Resumen:

Librerías:

<iostream>: *Entrada/salida estándar.*

<cstring>: *Manipulación de cadenas estilo C (funciones como strcpy, strlen, etc.).*

<cctype>: *Manipulación de caracteres (toupper, tolower, isalpha, etc.).*

static_cast<>

El operador **static_cast<>** en C++ es una herramienta utilizada para realizar conversiones de tipos de datos de manera explícita y controlada. Es parte del conjunto de operadores de conversión proporcionados por el lenguaje y es preferido sobre las conversiones de tipo estilo C ((**type**)) debido a su claridad y seguridad.

¿Qué es **static_cast<>**?

Definición: Es un operador de conversión que realiza conversiones entre tipos compatibles en tiempo de compilación.

Propósito: Facilitar conversiones seguras y específicas, verificadas por el compilador.

Uso principal: Conversiones entre tipos de datos primitivos, punteros, referencias y clases relacionadas por herencia.

¿Cómo se usa?

La sintaxis básica de **static_cast<>** es:

static_cast<tipo_destino>(expresión)

tipo_destino: El tipo al que se desea convertir.

expresión: La expresión que se va a convertir.

Ejemplo básico: Conversión de datos primitivos

```
#include <iostream>
```

```
int main() {
    double valor = 9.7;
    int entero = static_cast<int>(valor); // Conversión explícita de double a int
    std::cout << "Valor original: " << valor << "\n";
    std::cout << "Valor convertido: " << entero << "\n";
    return 0;
}
```

¿Dónde se usa?

static_cast<> es útil en una variedad de situaciones. Aquí están las más comunes:

1. Conversión entre tipos de datos primitivos

Convierte entre tipos como int, float, double, etc.

```
float x = 3.14;
int y = static_cast<int>(x); // Redondea hacia abajo.
```

2. Conversión de punteros

Permite convertir punteros hacia arriba (**upcasting**) o hacia abajo (**downcasting**) en una jerarquía de herencia, aunque no garantiza que sea seguro.

```
class Base {};
class Derivada : public Base {};
```

```
Base* basePtr = new Derivada;
Derivada* derivadaPtr = static_cast<Derivada*>(basePtr);
```

3. Conversión entre enumeraciones y enteros

```
enum class Color { Rojo, Verde, Azul };
```

```
int colorValue = static_cast<int>(Color::Verde); // Convierte un enum a int
Color color = static_cast<Color>(2); // Convierte un int a enum
```

4. Eliminación de **const** en expresiones temporales

Convierte tipos **no-const** a tipos **const**.

```
const int valor = 5;
int copia = static_cast<int>(valor); // Quita const (en expresiones temporales).
```

5. Conversión de punteros **void**

Convierte punteros **void** a un tipo específico.

```
void* ptr = &valor;
int* intPtr = static_cast<int*>(ptr);
```

¿Cuántos tipos de conversiones admite?

static_cast<> puede realizar las siguientes conversiones:

Datos primitivos: Entre int, float, double, char, etc.

Clases relacionadas por herencia: Para conversiones hacia arriba y hacia abajo.

Enumeraciones: A tipos enteros y viceversa.

Punteros void: A cualquier tipo de puntero.
Eliminación de const: En expresiones temporales.

Ventajas de `static_cast<>` sobre `cast` estilo C
Claridad: Indica explícitamente el propósito de la conversión.
Mayor seguridad: El compilador puede verificar la validez de la conversión en tiempo de compilación.
Legibilidad: Es más fácil de entender que una conversión estilo C.
Prevención de errores: Ayuda a evitar conversiones no intencionadas.

Limitaciones de `static_cast<>`
No es seguro en tiempo de ejecución: Para conversiones hacia abajo (downcasting) en jerarquías de herencia, no realiza comprobaciones en tiempo de ejecución. Si la conversión es inválida, el comportamiento es indefinido.
No se utiliza para conversiones de tipos no relacionados: Para conversiones seguras en jerarquías de herencia, se debe usar `dynamic_cast<>`.

```
Ejemplo práctico completo
Generar una clave aleatoria utilizando static_cast<>:
#include <iostream>
#include <random>

int main() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distribucion(48, 122); // Números, letras mayúsculas y
minúsculas.

    std::cout << "Clave aleatoria: ";
    for (int i = 0; i < 12; ++i) {
        char caracter = static_cast<char>(distribucion(gen)); // Convierte un número a su
carácter ASCII.
        std::cout << caracter;
    }
    std::cout << "\n";

    return 0;
}
```

Este ejemplo utiliza `static_cast<>` para convertir números enteros generados aleatoriamente en caracteres ASCII.

Conclusión
`static_cast<>` es una herramienta fundamental para realizar conversiones explícitas en C++. Al dominarlo, puedes escribir código más seguro, legible y compatible con las normas modernas de C++.

Tema Avanzado: `dynamic_cast<>`

El operador `dynamic_cast<>` en C++ es una herramienta utilizada para realizar conversiones entre tipos de *datos que están relacionados mediante herencia*. A diferencia de `static_cast<>`, este operador realiza comprobaciones en tiempo de ejecución para garantizar que la conversión sea válida y segura.

¿Qué es `dynamic_cast<>`?
Definición: Es un operador de conversión que convierte punteros o referencias de un tipo base a un tipo derivado (o viceversa) dentro de una jerarquía de herencia.
Propósito: Realizar conversiones seguras entre tipos relacionados, verificando en tiempo de ejecución si la conversión es válida.
Uso principal: Se utiliza comúnmente en herencia polimórfica (es decir, cuando las clases tienen al menos una función virtual).

¿Cómo se usa?
La sintaxis básica de `dynamic_cast<>` es:

```
dynamic_cast<tipo_destino>(expresión)
```

tipo_destino: El tipo al que se desea convertir.
expresión: La expresión que se va a convertir, usualmente un puntero o referencia.
Ejemplo básico: Conversión segura entre clases

```
#include <iostream>
#include <typeinfo> // Para obtener información sobre tipos

class Base {
public:
    virtual ~Base() {} // Necesario para el uso de `dynamic_cast`
};

class Derivada : public Base {
public:
    void mostrar() { std::cout << "Soy una clase derivada\n"; }
};

int main() {
    Base* basePtr = new Derivada; // Puntero base apuntando a una clase derivada

    // Intentamos convertirlo a Derivada
    Derivada* derivadaPtr = dynamic_cast<Derivada*>(basePtr);

    if (derivadaPtr) {
        derivadaPtr->mostrar(); // Éxito en la conversión
    } else {
        std::cout << "Conversión fallida\n";
    }

    delete basePtr;
    return 0;
}
```

¿Dónde se usa?

`dynamic_cast<>` se utiliza principalmente en las siguientes situaciones:

1. Herencia polimórfica

Cuando trabajas con clases que tienen al menos una función virtual, puedes convertir punteros o referencias de un tipo base a un tipo derivado.

```
class Animal {
public:
    virtual void sonido() = 0; // Clase abstracta
};

class Perro : public Animal {
public:
    void sonido() override { std::cout << "Guau\n"; }
};

class Gato : public Animal {
public:
    void sonido() override { std::cout << "Miau\n"; }
};

Animal* animal = new Perro;
Perro* perro = dynamic_cast<Perro*>(animal);

if (perro) {
    perro->sonido(); // "Guau"
}
```

2. Verificar el tipo en tiempo de ejecución

`dynamic_cast<>` devuelve `nullptr` para punteros o lanza una excepción para referencias si la conversión no es válida, lo que permite manejar conversiones erróneas.

Características de `dynamic_cast<>`

Para punteros: Si la conversión no es válida, devuelve `nullptr`.

Para referencias: Si la conversión no es válida, lanza una excepción de tipo `std::bad_cast`.

Requiere herencia polimórfica: Las clases deben tener al menos una función virtual para usar `dynamic_cast<>`.

```
Ejemplo con referencias
#include <iostream>
#include <typeinfo>

class Base {
public:
    virtual ~Base() {} // Necesario para `dynamic_cast`
};

class Derivada : public Base {};

int main() {
    Base baseObj;
    try {
        Derivada& ref = dynamic_cast<Derivada&>(baseObj); // Lanza std::bad_cast
    } catch (const std::bad_cast& e) {
        std::cout << "Conversión fallida: " << e.what() << "\n";
    }
    return 0;
}
```

¿Cómo se diferencia de **static_cast**<>?

Aspecto	<code>dynamic_cast<></code>	<code>static_cast<></code>
Verificación de tipo	En tiempo de ejecución	Solo en tiempo de compilación
Herencia polimórfica	Requiere al menos una función virtual	No la requiere
Resultado inválido	Devuelve <code>nullptr</code> (punteros) o lanza una excepción	Comportamiento indefinido
Rendimiento	Más lento debido a la verificación de tipo dinámica	Más rápido, pero menos seguro

Ventajas de **dynamic_cast**<>

- Seguridad en tiempo de ejecución:** Garantiza que la conversión sea válida.
- Manejo de errores:** Devuelve `nullptr` o lanza una excepción en caso de error.
- Flexibilidad:** Permite trabajar con jerarquías polimórficas de manera segura.

Limitaciones de **dynamic_cast**<>

- Requiere funciones virtuales:** No puede usarse en clases sin funciones virtuales.
- Costo en rendimiento:** Más lento que **static_cast**<> debido a la verificación en tiempo de ejecución.
- Solo para jerarquías de herencia:** No puede usarse para conversiones fuera de este contexto.

Ejemplo práctico completo
Supongamos que tienes una lista de animales y necesitas verificar su tipo en tiempo de ejecución para realizar acciones específicas:

```
#include <iostream>
#include <vector>
#include <memory>

class Animal {
public:
    virtual ~Animal() {}
    virtual void sonido() = 0;
};

class Perro : public Animal {
public:
    void sonido() override { std::cout << "Guau\n"; }
    void ladrar() { std::cout << "Ladrando fuerte\n"; }
};

class Gato : public Animal {
public:
    void sonido() override { std::cout << "Miau\n"; }
};

int main() {
```



```

std::vector<std::unique_ptr<Animal>> animales;
animales.emplace_back(std::make_unique<Perro>());
animales.emplace_back(std::make_unique<Gato>());

for (auto& animal : animales) {
    animal->sonido();

    if (auto* perro = dynamic_cast<Perro*>(animal.get())) {
        perro->ladrar(); // Llamar a un método específico de Perro
    }
}

return 0;
}

```

Conclusión

***dynamic_cast**<> es una herramienta poderosa para trabajar con jerarquías polimórficas en C++. Proporciona seguridad y flexibilidad al realizar conversiones entre tipos, verificando que estas sean válidas en tiempo de ejecución. Sin embargo, su uso debe ser justificado debido a su impacto en el rendimiento.*

Conversión:

En C++, puedes trabajar con el tipo de dato **char** para manejar caracteres y utilizar su valor **ASCII** fácilmente. Cada carácter tiene un valor numérico asociado en el estándar ASCII, y puedes mostrarlo simplemente convirtiendo el **char** a un entero (**int**). A continuación, te muestro cómo hacerlo con un ejemplo práctico:

Ejemplo: Obtener el valor ASCII de un carácter ingresado

```

#include <iostream>

using namespace std;

int main() {
    char letra; // Variable para almacenar el carácter

    cout << "Introduce una letra: ";
    cin >> letra; // Leer un carácter desde el usuario

    // Mostrar el carácter y su valor ASCII
    cout << "El carácter ingresado es: " << letra << endl;
    cout << "Su valor ASCII es: " << static_cast<int>(letra) << endl;

    return 0;
}

```

Explicación del código:

Declaración de char:

Usamos **char letra**; para definir una variable que almacena un único carácter.

Entrada del usuario (**cin**):

cin >> letra; permite que el usuario ingrese un carácter. Solo se capturará el primer carácter introducido.

Conversión a entero:

static_cast<int>(letra) convierte el carácter almacenado en su valor numérico correspondiente en la tabla ASCII.

La conversión también se puede realizar de forma implícita (solo escribiendo el nombre de la variable en un contexto de entero), pero el uso de **static_cast** es más claro y explícito.

Salida (**cout**):

Mostramos el carácter y su valor ASCII.

Ejemplo de ejecución:

Entrada del usuario:

Introduce una letra: **A**

Salida:

El carácter ingresado es: **A**

Su valor ASCII es: **65**

Notas importantes:

Si ingresas un carácter en minúscula, su valor ASCII será diferente del mismo carácter en mayúscula. Por ejemplo:

'A' tiene el valor ASCII 65.

'a' tiene el valor ASCII 97.

Práctica adicional: Tabla de ASCII para una palabra

Si quieres mostrar los valores ASCII de todos los caracteres de una palabra, puedes usar un bucle.

Código:

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string palabra;

    cout << "Introduce una palabra: ";
    cin >> palabra;

    cout << "Valores ASCII de los caracteres: " << endl;
    for (char c : palabra) { // Iterar sobre cada carácter
        cout << c << " -> " << static_cast<int>(c) << endl;
    }

    return 0;
}
```

Entrada del usuario:

Introduce una palabra: **Hola**

Salida:

Valores ASCII de los caracteres:

H -> 72

o -> 111

l -> 108

a -> 97

Nota

En C++, para ingresar un texto que contenga espacios (como "hola mundo") en un array de caracteres (`char[]`), necesitas utilizar una función que permita capturar la línea completa, como `std::cin.getline()` o `std::getline()` con un `std::string`.

Razón por la que aparece solo "hola"

Cuando usas `std::cin >> variable;`, este detiene la lectura en el primer espacio encontrado, capturando solo la palabra inicial. Para capturar todo el texto con espacios, debes emplear las siguientes soluciones:

1. Usando `std::cin.getline()` con un `char[]`

```
#include <iostream>

int main() {
    char texto[100]; // Array para almacenar el texto

    std::cout << "Introduce un texto con espacios: ";
    std::cin.getline(texto, 100); // Captura hasta 99 caracteres (reserva 1 para '\0')

    std::cout << "Texto ingresado: " << texto << std::endl;

    return 0;
}
```

Explicación:

`std::cin.getline(texto, tamaño)` permite leer una línea completa hasta el primer salto de línea (`\n`) o hasta que se alcance el tamaño especificado.

En este ejemplo, la entrada puede contener espacios y se almacenará correctamente en el array `texto`.

2. Usando `std::getline()` con `std::string`

Si prefieres trabajar con `std::string`, puedes usar la función `std::getline` de la biblioteca estándar:

```
#include <iostream>
#include <string>

int main() {
    std::string texto;

    std::cout << "Introduce un texto con espacios: ";
    std::getline(std::cin, texto); // Captura la línea completa

    std::cout << "Texto ingresado: " << texto << std::endl;

    return 0;
}
```

Explicación:

`std::getline(std::cin, texto)` lee todo el texto introducido hasta que se detecta un salto de línea (`\n`).

Funciona únicamente con `std::string`, no con `char[]`.

3. Combinando entradas con `std::cin` y `std::getline()`

Si usas `std::cin` antes de `std::getline`, es posible que se generen problemas debido a caracteres residuales en el búfer (como el `\n` dejado por `std::cin`). Para evitar esto, puedes limpiar el búfer usando `std::cin.ignore()`.

```
#include <iostream>
#include <string>

int main() {
    std::string texto;
    int numero;

    std::cout << "Introduce un número: ";
    std::cin >> numero;
```

```
std::cin.ignore(); // Limpia el búfer antes de usar std::getline

std::cout << "Introduce un texto con espacios: ";
std::getline(std::cin, texto);

std::cout << "Número ingresado: " << numero << std::endl;
std::cout << "Texto ingresado: " << texto << std::endl;

return 0;
}
```

Diferencias entre `std::cin.getline()` y `std::getline()`

Característica	<code>std::cin.getline()</code>	<code>std::getline()</code>
Entrada	Trabaja con <code>char[]</code>	Trabaja con <code>std::string</code>
Límite de caracteres	Requiere un tamaño fijo	Automáticamente ajusta el tamaño
Manejo del búfer	No siempre limpia el búfer	Más seguro con entradas mixtas

Tipo de Dato String

En C++, el tipo de dato **string** pertenece a la Biblioteca Estándar de C++ (STL) y se encuentra en el encabezado `<string>`. Proporciona una manera fácil y eficiente de trabajar con cadenas de caracteres, reemplazando el uso de arreglos de caracteres (`char[]`) que se usaban en el lenguaje C.

A continuación, te explico con detalle sus características, funcionalidades, operaciones básicas y ejemplos prácticos.

1. ¿Qué es `std::string`?

`std::string` es una clase que representa una cadena de caracteres. Proporciona una interfaz rica para manejar texto, como concatenación, comparación, extracción de sub-cadenas, búsqueda, entre otras operaciones. Internamente, administra dinámicamente su memoria, lo que lo hace más flexible y fácil de usar en comparación con arreglos estáticos de caracteres (`char[]`). Para usar **string**, debes incluir el encabezado:

```
#include <string>
```

2. Declaración e inicialización de `std::string`

Ejemplo 1: Declaración e inicialización

```
#include <iostream>
#include <string> // Necesario para usar string

using namespace std;

int main() {
    string saludo = "Hola Mundo"; // Inicialización directa
    string nombre("Juan");         // Constructor
    string mensaje;                 // Declaración sin inicializar

    mensaje = "Bienvenido a C++"; // Asignación posterior

    cout << saludo << endl;
    cout << "Hola, " << nombre << endl;
    cout << mensaje << endl;

    return 0;
}
```

Salida:

Hola Mundo
Hola, Juan
Bienvenido a C++

3. Operaciones comunes con `std::string`

a) Concatenación

Puedes combinar cadenas con el operador `+` o `+=`.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string nombre = "Juan";
    string saludo = "Hola, " + nombre; // Concatenación con +
    saludo += "! Bienvenido.";          // Concatenación con +=

    cout << saludo << endl;

    return 0;
}
```

Salida:

Hola, Juan! Bienvenido.

b) Acceso a caracteres

Puedes acceder a un carácter específico usando índices (como en un arreglo).

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string texto = "Hola";

    // Accediendo al primer carácter
    cout << "Primer carácter: " << texto[0] << endl;

    // Cambiando un carácter
    texto[1] = 'e'; // Cambia 'o' por 'e'
    cout << "Texto modificado: " << texto << endl;

    return 0;
}
```

Salida:

Primer carácter: H
Texto modificado: Hela

c) Longitud de la cadena

Usa el método `.length()` o `.size()`.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string palabra = "Programación";

    cout << "La longitud de la palabra es: " << palabra.length() << endl;

    return 0;
}
```

Salida:

La longitud de la palabra es: 12

d) Subcadenas

Usa el método `.substr()` para *extraer partes de la cadena*.

```
#include <iostream>
#include <string>

using namespace std;
```

```
int main() {
    string frase = "Hola Mundo";

    string subcadena = frase.substr(5, 5); // Extrae desde índice 5, longitud 5
    cout << "Subcadena: " << subcadena << endl;

    return 0;
}
```

Salida:

Subcadena: Mundo

e) Búsqueda

Puedes buscar una **subcadena** o carácter con `.find()`.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string texto = "Bienvenido a C++";

    size_t posicion = texto.find("C++"); // Busca "C++"

    if (posicion != string::npos) {
        cout << "La palabra 'C++' se encuentra en la posición: " << posicion << endl;
    } else {
        cout << "No se encontró 'C++' en el texto." << endl;
    }

    return 0;
}
```

Salida:

La palabra 'C++' se encuentra en la posición: 14

f) Comparación

Puedes comparar cadenas con los operadores habituales (`==`, `!=`, `<`, `>`).

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string cadena1 = "Hola";
    string cadena2 = "Mundo";

    if (cadena1 == "Hola") {
        cout << "Las cadenas son iguales" << endl;
    }

    if (cadena1 < cadena2) {
        cout << cadena1 << " es menor que " << cadena2 << endl;
    }

    return 0;
}
```

Salida:

Las cadenas son iguales
Hola es menor que Mundo

g) Inserción y eliminación

Usa métodos como `.insert()`, `.erase()` y `.replace()`.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
    string texto = "Hola C++";
```

```

texto.insert(5, "Mundo "); // Inserta "Mundo " en la posición 5
texto.erase(9, 3);        // Elimina 3 caracteres desde la posición 9
texto.replace(0, 4, "Adiós"); // Reemplaza los primeros 4 caracteres con "Adiós"

cout << texto << endl;

return 0;
}

```

Salida:

Adiós Mundo

4. Ventajas de `std::string` sobre `char[]`

- Gestión dinámica de memoria:

`std::string` ajusta automáticamente su tamaño según el contenido. Con `char[]`, debes definir manualmente el tamaño.

- Interfaz rica:

`std::string` proporciona métodos listos para usar, como concatenación, búsqueda, extracción de subcadenas, etc.

- Seguridad:

Evita errores comunes como desbordamientos de búfer, que ocurren con `char[]`.

5. Ejemplo avanzado: Reversión de una cadena

```

#include <iostream>
#include <string>
#include <algorithm> // Para std::reverse

using namespace std;

int main() {
    string texto = "Hola Mundo";

    // Reversión de la cadena
    reverse(texto.begin(), texto.end());

    cout << "Texto invertido: " << texto << endl;

    return 0;
}

```

Salida:

odnuM aloH

Solicitar (ingresar) cadenas .

Cómo solicitar el nombre de una persona y guardarlo en una variable `std::string` en `C++`. Es bastante sencillo y similar a lo que hemos visto antes. Aquí tienes un ejemplo:

```

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

int main() {
    string nombre;

    cout << "Ingresa tu nombre: ";
    getline(cin, nombre); // Leer el nombre completo, incluyendo espacios

    cout << "¡Hola, " << nombre << "! Bienvenido." << endl;

    return 0;
}

```

En este ejemplo:

1. Incluimos la biblioteca `<string>` para usar la clase `std::string`.

2. Declaramos una variable nombre de tipo `std::string`.
3. Usamos `getline(cin, nombre)` para leer el nombre del usuario. La función `getline` permite leer una línea completa de entrada, lo cual es útil para capturar nombres completos que pueden incluir espacios.
4. Finalmente, mostramos un mensaje de bienvenida utilizando el nombre ingresado.

Esto te permitirá solicitar el nombre de una persona y almacenarlo correctamente en una variable de tipo `std::string`.

Ejemplo anterior donde solicitamos varios nombres y los guardamos dentro de un arreglo de tipo `std::string`:

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

int main() {
    const int numNombres = 5; // Número de nombres a ingresar
    string nombres[numNombres]; // Arreglo de strings

    // Ingresar los nombres
    for (int i = 0; i < numNombres; ++i) {
        cout << "Ingresa el nombre de la persona " << i + 1 << ": ";
        getline(cin, nombres[i]);
    }

    // Mostrar los nombres ingresados
    cout << "\nNombres ingresados:\n";
    for (int i = 0; i < numNombres; ++i) {
        cout << i + 1 << ". " << nombres[i] << endl;
    }

    return 0;
}
```

En este ejemplo:

1. Definimos una constante `numNombres` para especificar cuántos nombres se van a ingresar.
2. Declaramos un arreglo `nombres` de tipo `std::string` con el tamaño `numNombres`.
3. Utilizamos un ciclo `for` para solicitar al usuario que ingrese los nombres y los almacenamos en el arreglo utilizando `getline(cin, nombres[i])`.
4. Mostramos los nombres ingresados en un ciclo `for` adicional.

Esto te permitirá solicitar varios nombres y almacenarlos en un arreglo, facilitando la gestión de múltiples entradas de texto.

¿Qué es size_t?

size_t es un tipo de dato sin signo (**unsigned**) definido en el encabezado estándar `<cstdint>` (o indirectamente en otros como `<iostream>` o `<vector>`). Está diseñado para representar tamaños y conteos en bytes, por lo que siempre es positivo.

Características principales de **size_t**:

Representa tamaños o índices:

Es ampliamente utilizado para representar tamaños de objetos o índices de **arrays**.

Es el tipo de retorno de la función `sizeof`.

Especificidad del sistema:

Su tamaño depende de la plataforma (32 o 64 bits). Por ejemplo:

En sistemas de **32 bits**: suele ser equivalente a unsigned int.

En sistemas de **64 bits**: suele ser equivalente a unsigned long long.

Evita problemas de compatibilidad:

Está diseñado para manejar correctamente los tamaños máximos de memoria en la arquitectura del sistema, asegurando que los programas sean portables.

¿Dónde se usa size_t?

Índices de **arrays**: Cuando recorres un **array** con un bucle, puedes usar **size_t** para asegurarte de que no habrá problemas de signo.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    for (size_t i = 0; i < vec.size(); ++i) {
        std::cout << "Elemento " << i << ": " << vec[i] << std::endl;
    }
    return 0;
}
```

Función **sizeof**: La función **sizeof** devuelve un valor de tipo **size_t**.

```
#include <iostream>

int main() {
    int arr[10];
    std::cout << "Tamaño del array en bytes: " << sizeof(arr) << std::endl;
    return 0;
}
```

Funciones de las bibliotecas estándar: Muchas funciones de la **STL**, como las que devuelven tamaños (`std::string::length()`, `std::vector::size()`), tienen valores de tipo **size_t**.

Ventajas de usar size_t:

1. Evita errores de signo: Como es sin signo, no puedes asignarle un valor negativo.
2. Portabilidad: Adapta su tamaño automáticamente según la arquitectura del sistema.
3. Eficiencia: Al ser el tipo de dato usado internamente por muchas funciones de la STL, evita conversiones innecesarias.

Precaución al usar size_t:

Comparar **size_t** con tipos con signo como **int** puede generar advertencias o comportamientos no deseados debido a la conversión implícita.

```
int a = -1;
size_t b = 10;

if (a < b) { // ¡Esto puede no comportarse como esperas!
    std::cout << "a es menor que b" << std::endl;
}
```

La comparación aquí puede fallar porque **a** será convertido a **size_t** (sin signo), lo que cambiará su valor a un número grande.

Npos

En C++, `npos` es una constante especial asociada con las clases de cadenas (`std::string`, `std::wstring`, etc.) y otros contenedores relacionados, como `std::string_view`. Esta constante se utiliza para indicar que no se encontró una posición válida en ciertas operaciones.

¿Qué significa npos?

`npos` es una abreviatura de **"no position"** (sin posición).

Representa un valor especial (*generalmente el valor máximo de `size_t`*) que indica que una búsqueda o una operación falló, como cuando no se encuentra un `substring` o un carácter.

¿Cómo se define?

`npos` está definido como un miembro estático constante dentro de las clases relacionadas con cadenas:

```
static const size_t npos = -1;
size_t: El tipo de dato que almacena posiciones o tamaños.
```

`-1`: Como `size_t` es sin signo, el valor de `-1` se convierte en el máximo valor posible de `size_t`.

Usos de npos

Indicar que no se encontró algo:

Cuando buscas un `substring` o carácter en una cadena, las funciones como `find`, `rfind`, `find_first_of`, etc., devuelven `npos` si no encuentran el elemento buscado.

Ejemplo:

```
#include <iostream>
#include <string>

int main() {
    std::string text = "Hola, mundo";

    // Buscar "mundo" en la cadena
    size_t pos = text.find("mundo");
    if (pos != std::string::npos) {
        std::cout << "'mundo' encontrado en la posición: " << pos << std::endl;
    } else {
        std::cout << "'mundo' no encontrado" << std::endl;
    }

    // Buscar "adios" en la cadena
    pos = text.find("adios");
    if (pos == std::string::npos) {
        std::cout << "'adios' no encontrado" << std::endl;
    }

    return 0;
}
```

Salida:

```
'mundo' encontrado en la posición: 6
'adios' no encontrado
```

Recortar cadenas o verificar operaciones:

Puedes usar `npos` para verificar si un índice es válido antes de operar con una cadena:

```
#include <iostream>
#include <string>

int main() {
    std::string text = "Hola, mundo";
    size_t pos = text.find("mundo");

    if (pos != std::string::npos) {
        std::string sub = text.substr(pos); // Obtener el substring desde 'mundo'
        std::cout << "Substring: " << sub << std::endl;
    }
    return 0;
}
```

Salida:

```
Substring: mundo
```

Ventajas de usar npos

1. Legibilidad: Indica claramente que algo no fue encontrado.
2. Portabilidad: npos siempre tiene el valor correcto, incluso si el tamaño de size_t varía entre arquitecturas.
3. Evita errores de rango: Sirve como una forma estándar de manejar búsquedas fallidas.

Cosas a tener en cuenta

Si comparas directamente con `-1` en lugar de `npos`, podrías introducir errores o problemas de portabilidad. Es más seguro y claro usar `npos`.

`npos` solo es válido en el contexto de clases relacionadas con cadenas o contenedores que lo definan (como `std::string`, `std::string_view`).

La biblioteca de funciones

string.h en C

Es una colección de funciones para manejar cadenas de caracteres. Estas funciones trabajan con cadenas que son arreglos de caracteres terminados en un carácter nulo (`\0`). Aunque en `C++` generalmente usamos la clase `std::string`, las funciones de `string.h` son útiles en ciertos casos, especialmente en programación de bajo nivel o sistemas embebidos.

Voy a explicarte las funciones más importantes de `string.h`, incluyendo las que mencionaste.

1. Incluir la biblioteca

Para usar estas funciones, debes incluir el encabezado `<cstring>` en `C++`, que es equivalente a `string.h` en `C`.

```
#include <cstring>
#include <iostream> // Solo necesario si usas cout/cin en C++
```

2. Funciones principales

a) `strlen`: Longitud de una cadena

Esta función calcula la longitud de una cadena (sin contar el carácter nulo `\0`).

Prototipo:

```
size_t strlen(const char *str);
```

Ejemplo:

```
#include <cstring>
#include <iostream>

using namespace std;

int main() {
    char texto[] = "Hola Mundo";
    size_t longitud = strlen(texto);

    cout << "La longitud de \"" << texto << "\" es: " << longitud << endl;

    return 0;
}
```

Salida:

La longitud de "Hola Mundo" es: 10

b) `strcpy`: Copiar una cadena

Copia el contenido de una cadena origen a una cadena destino. Asegúrate de que la cadena destino tenga suficiente espacio.

Prototipo:

```
char *strcpy(char *dest, const char *src);
```

Ejemplo:

```
#include <cstring>
#include <iostream>

using namespace std;

int main() {
    char origen[] = "Hola";
    char destino[10];

    strcpy(destino, origen); // Copia el contenido de 'origen' a 'destino'

    cout << "Cadena destino: " << destino << endl;

    return 0;
}
```

Salida:

Cadena destino: Hola

c) **strcmp: Comparar cadenas**

Compara dos cadenas carácter por carácter.

Devuelve 0 si son iguales.

Devuelve un valor negativo si la primera cadena es menor.

Devuelve un valor positivo si la primera cadena es mayor.

Prototipo:

```
int strcmp(const char *str1, const char *str2);
```

Ejemplo:

```
#include <cstring>
#include <iostream>

using namespace std;

int main() {
    char str1[] = "Hola";
    char str2[] = "Mundo";

    int resultado = strcmp(str1, str2);

    if (resultado == 0) {
        cout << "Las cadenas son iguales." << endl;
    } else if (resultado < 0) {
        cout << "\"" << str1 << "\" es menor que \"" << str2 << "\"" << endl;
    } else {
        cout << "\"" << str1 << "\" es mayor que \"" << str2 << "\"" << endl;
    }

    return 0;
}
```

Salida:

"Hola" es menor que "Mundo"

d) **strcat: Concatenar cadenas**

Agrega el contenido de una cadena al final de otra.

Prototipo:

```
char *strcat(char *dest, const char *src);
```

Ejemplo:

```
#include <cstring>
#include <iostream>

using namespace std;

int main() {
    char cadena1[20] = "Hola";
    char cadena2[] = " Mundo";

    strcat(cadena1, cadena2); // Concatenar 'cadena2' al final de 'cadena1'

    cout << "Cadena resultante: " << cadena1 << endl;

    return 0;
}
```

Salida:

Cadena resultante: Hola Mundo

e) **strstr: Buscar una subcadena**

Busca la primera aparición de una subcadena dentro de otra.

Prototipo:

```
char *strstr(const char *haystack, const char *needle);
```

Ejemplo:

```
#include <cstring>
#include <iostream>

using namespace std;

int main() {
    char texto[] = "Hola Mundo";
    char subcadena[] = "Mundo";

    char *resultado = strstr(texto, subcadena);

    if (resultado) {
        cout << "Subcadena encontrada: " << resultado << endl;
    } else {
        cout << "Subcadena no encontrada." << endl;
    }

    return 0;
}
```

Salida:

Subcadena encontrada: Mundo

f) **strchr**: **Buscar un carácter**

Busca la primera aparición de un carácter en una cadena.

Prototipo:

```
char *strchr(const char *str, int c);
```

Ejemplo:

```
#include <cstring>
#include <iostream>

using namespace std;

int main() {
    char texto[] = "Hola Mundo";

    char *resultado = strchr(texto, 'M');

    if (resultado) {
        cout << "Carácter encontrado en: " << resultado << endl;
    } else {
        cout << "Carácter no encontrado." << endl;
    }

    return 0;
}
```

Salida:

Carácter encontrado en: Mundo

g) **strncpy**: **Copiar una cantidad específica de caracteres**

Copia los primeros n caracteres de una cadena a otra.

Prototipo:

```
char *strncpy(char *dest, const char *src, size_t n);
```

Ejemplo:

```
#include <cstring>
#include <iostream>

using namespace std;

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

    strncpy(destino, origen, 4); // Copiar los primeros 4 caracteres
```

```
destino[4] = '\0';           // Asegurar terminación nula

cout << "Cadena copiada: " << destino << endl;

return 0;
}
```

Salida:

Cadena copiada: Hola

3. *Consideraciones importantes*

Espacio en memoria: Asegúrate de que la cadena destino tenga suficiente espacio para evitar desbordamientos.

Carácter nulo: Muchas funciones asumen que las cadenas terminan con \0. Si no lo tienen, pueden ocurrir errores.

Alternativas modernas: En C++, se recomienda usar std::string en lugar de char[] por seguridad y facilidad de manejo.

c. Operadores

Operadores Aritméticos:

- +: Suma
- : Resta
- *: Multiplicación
- /: División
- ?: Módulo (resto de una división entre enteros)

Operadores de Asignación:

- =: Asignación básica.
- +=, -=, *=, /=, %=: Operadores de asignación compuesta (Ej. `x += 1` es lo mismo que `x = x + 1`).

Operadores Relacionales:

- == Igual a 2==2
- != No igual a 3!=9
- > Mayor que 6 > 2
- < Menor que 4 < 100
- >= Mayor o igual que
- <= Menor o igual que

Operadores Lógicos:

- &&: Y lógico (AND)
- ||: O lógico (OR)
- !: Negación lógica (NOT)

2. Estructuras de Control

En C++, las estructuras de control son fundamentales para dirigir el flujo de ejecución de un programa. Estas incluyen estructuras condicionales, de bucle y de control de salto. A continuación te explico cada una con ejemplos prácticos:

1. Estructuras Condicionales

a) if-else

La estructura **if** evalúa una condición; si es verdadera, ejecuta el bloque de código asociado. Si es falsa, puede haber un bloque **else** que se ejecuta.

Ejemplo:

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

int main() {
    int num;
    cout << "Ingresa un número: ";
    cin >> num;

    if (num > 0) {
        cout << "El número es positivo." << endl;
    } else if (num == 0) {
        cout << "El número es cero." << endl;
    } else {
```

```

        cout << "El número es negativo." << endl;
    }

    return 0;
}

```

Explicación:

Este código evalúa si el número ingresado es positivo, negativo o cero, utilizando varias condiciones **if-else**.

b) switch

El **switch** evalúa el valor de una variable y ejecuta el código dentro del caso que coincida con el valor.

Ejemplo:

```

#include <iostream>
using namespace std;

int main() {
    int opcion;
    cout << "Selecciona una opción (1-3): ";
    cin >> opcion;

    switch (opcion) {
        case 1:
            cout << "Opción 1 seleccionada." << endl;
            break;
        case 2:
            cout << "Opción 2 seleccionada." << endl;
            break;
        case 3:
            cout << "Opción 3 seleccionada." << endl;
            break;
        default:
            cout << "Opción no válida." << endl;
    }

    return 0;
}

```

Explicación:

Aquí, dependiendo del valor de `opcion`, se ejecuta un caso específico. Si el valor no coincide con ningún caso, se ejecuta el bloque **default**.

El **switch** es una estructura de control que permite ejecutar diferentes bloques de código basados en el valor de una expresión. Es particularmente útil cuando se tiene una única variable que puede tomar varios valores discretos y se desea ejecutar un bloque de código diferente para cada uno de esos valores.

Sintaxis Básica del **switch**

```

switch (expresión) {
    case valor1:
        // Bloque de código si expresión == valor1
        break;
    case valor2:
        // Bloque de código si expresión == valor2
        break;
    // Puedes añadir tantos case como necesites
    default:
        // Bloque de código si la expresión no coincide con ningún case
}

```

Elementos del switch

expresión:

La expresión dentro de los paréntesis es evaluada una vez.

Debe ser de un tipo integral o un **enum** (es decir, **int**, **char**, etc.). En **C++11** y versiones posteriores, también se permiten expresiones de tipo **long long**, **unsigned long long**, y **char32_t**.

case valor::

La palabra clave **case** se utiliza para definir un valor con el que se comparará la expresión. Si la expresión coincide con `valor`, el bloque de código correspondiente será ejecutado. Los valores deben ser constantes, literales o constantes definidas con **#define** o **const**. Los valores de los `case` deben ser únicos; no puede haber duplicados.

break:

La instrucción **break** se usa para salir del bloque **switch**.

Sin **break**, el control del programa continúa hacia el siguiente **case**, lo que puede llevar a la ejecución de múltiples bloques de código (esto se conoce como **fall-through**).

El **break** detiene el flujo y pasa el control al siguiente código fuera del **switch**.

default:

La palabra clave **default** es opcional.

Se ejecuta si ninguna de las case coincide con la expresión.

Funciona como el **else** en una estructura **if-else**.

Ejemplo con Explicación

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

int main() {
    int opcion = 2;

    switch (opcion) {
        case 1:
            cout << "Elegiste la opción 1." << endl;
            break;
        case 2:
            cout << "Elegiste la opción 2." << endl;
            break;
        case 3:
            cout << "Elegiste la opción 3." << endl;
            break;
        default:
            cout << "Opción no válida." << endl;
    }

    return 0;
}
```

Explicación:

Si opción es igual a 2, el **switch** evalúa cada case hasta encontrar una coincidencia.

En este caso, coincide con case 2:, por lo que se ejecuta **cout << "Elegiste la opción 2." << endl;**.

Después de ejecutar este bloque, el **break** detiene el **switch**, evitando que el código de los demás case se ejecute.

Si opcion fuera un valor distinto de 1, 2 o 3, se ejecutaría el bloque bajo **default**.

Fall-Through

Si se omite el **break**, el control pasa al siguiente case, incluso si no coincide con la expresión. Esto es conocido como **fall-through**.

```
int dia = 2;

switch (dia) {
    case 1:
        cout << "Lunes" << endl;
    case 2:
        cout << "Martes" << endl;
    case 3:
        cout << "Miércoles" << endl;
        break;
    default:
        cout << "Día no válido" << endl;
}
```

En este caso, si día es igual a 2, se ejecutarán las instrucciones de case 2 y case 3 porque no hay **break** en case 2. Esto puede ser útil en algunos escenarios, pero a menudo lleva a errores si no se maneja cuidadosamente.

Uso de switch con enum (enum → tema 22)

El **switch** también es muy útil en combinación con **enum**, lo que mejora la legibilidad del código.

```
enum Dia {Lunes, Martes, Miercoles, Jueves, Viernes};
```

```
Dia hoy = Martes;
```

```
switch (hoy) {
    case Lunes:
        cout << "Hoy es lunes." << endl;
```

```

        break;
    case Martes:
        cout << "Hoy es martes." << endl;
        break;
    case Miercoles:
        cout << "Hoy es miércoles." << endl;
        break;
    default:
        cout << "Es otro día de la semana." << endl;
}

```

En este ejemplo, el uso de `enum` hace que el código sea más fácil de entender y mantener.

Consideraciones Adicionales

No hay Evaluación de Rangos: `switch` no permite evaluar rangos directamente (e.g., `case x > 5:`). Para esto, deberías usar una estructura `if-else`.

switch con Expresiones de Carácter: También puedes usar `switch` con caracteres, dado que el tipo `char` es tratado como un entero en C++.

Conclusión

El `switch` es una herramienta poderosa para manejar múltiples condiciones basadas en el valor de una sola variable, ofreciendo una alternativa más clara y eficiente que múltiples `if-else`. Sin embargo, es importante recordar que `switch` tiene sus limitaciones y no es adecuado para todas las situaciones.

El comportamiento del `switch` en C++ es muy específico y sigue un flujo estructurado. A continuación, te explico cómo funciona en detalle:

Flujo de Ejecución del switch

Evaluación de la Expresión:

El `switch` comienza evaluando la expresión que se coloca dentro de los paréntesis `switch(expresión)`. Esta expresión puede ser de un tipo de dato integral como `int`, `char`, `enum`, o en versiones más recientes de C++, tipos como `long long` o `char32_t`. La expresión debe dar como resultado un valor que el `switch` pueda comparar con los valores definidos en cada case. Comparación con Casos (`case`):

El valor de la expresión evaluada se compara secuencialmente con cada uno de los valores especificados en los case.

Cada `case` define un valor constante con el que se compara la expresión. Si se encuentra una coincidencia, el flujo del programa entra en el bloque de código correspondiente a ese case.

Ejecución del Código:

Cuando se encuentra un case coincidente, se ejecuta el bloque de código asociado con ese case. Sin `break`: Si no hay una instrucción `break` al final del bloque de código, el flujo continuará ejecutando el siguiente `case` y el siguiente, independientemente de que coincidan o no (esto se llama *fall-through*).

Con `break`: Si el bloque de código incluye una instrucción `break`, el flujo se detiene y el programa sale del `switch`, continuando con la ejecución del código que sigue después del `switch`.

Manejo del Caso default:

Si la expresión no coincide con ningún case, se ejecutará el bloque de código asociado al `default`, si este existe.

El `default` es opcional, pero es útil para manejar cualquier caso que no haya sido específicamente cubierto por los case.

Ejemplo para Demostrar el Comportamiento del `switch`

Consideremos un ejemplo para ilustrar el flujo de ejecución y cómo se comporta `switch` con y sin la instrucción `break`.

```

#include <iostream>
using namespace std;

int main() {
    int numero = 2;

    switch (numero) {
        case 1:
            cout << "El número es 1" << endl;
            break;
        case 2:
            cout << "El número es 2" << endl;
            // Aquí no hay break
    }
}

```

```

        case 3:
            cout << "El número es 3" << endl;
            break;
        case 4:
            cout << "El número es 4" << endl;
            break;
        default:
            cout << "Número no reconocido" << endl;
    }

    return 0;
}

```

Salida del Programa:

El número es 2
El número es 3

Explicación:

La expresión numero se evalúa y da como resultado 2.

El **switch** encuentra el case 2: y ejecuta el **cout << "El número es 2" << endl;**.

Sin **break**: Como no hay **break** después de este bloque, el flujo continúa hacia case 3: y ejecuta también el **cout << "El número es 3" << endl;**.

Luego encuentra un break en el case 3, lo que provoca que el flujo salga del **switch**.

Fall-Through:

Ejemplo Común

En algunos casos, el comportamiento de **fall-through** se usa intencionalmente. Por ejemplo, para agrupar varios valores que ejecuten el mismo código:

```

#include <iostream>
using namespace std;

int main() {
    char grado = 'B';

    switch (grado) {
        case 'A':
        case 'B':
        case 'C':
            cout << "Pasaste con buena calificación." << endl;
            break;
        case 'D':
            cout << "Pasaste con calificación baja." << endl;
            break;
        case 'F':
            cout << "Reprobaste." << endl;
            break;
        default:
            cout << "Grado no válido." << endl;
    }

    return 0;
}

```

Salida del Programa:

Pasaste con buena calificación.

Explicación:

En este ejemplo, los case 'A':, case 'B':, y case 'C': no tienen **break** entre ellos.

Si grado es 'A', 'B', o 'C', se ejecuta el mismo bloque de código que imprime "Pasaste con buena calificación."

El **break** después de este bloque detiene el flujo antes de que se evalúe case 'D':.

Consideraciones Importantes

Evitar **Fall-Through** No Deseado: Si no necesitas un comportamiento de **fall-through**, siempre utiliza **break** para evitar que el flujo continúe accidentalmente.

Tipo de Expresión: La expresión evaluada en el switch debe ser un tipo integral o enum. No se pueden utilizar tipos de datos como **float**, **double**, o **std::string** directamente.

Valores de los **case**: Deben ser valores constantes y únicos. No pueden repetirse dentro del mismo **switch**.

Conclusión

El **switch** es una herramienta eficiente para manejar múltiples condiciones basadas en una única expresión. Su comportamiento está claramente definido por la presencia o ausencia de **break**, lo que determina si el flujo continúa hacia el siguiente case o sale del **switch**. El uso correcto de **switch** puede hacer tu código más legible y organizado cuando se enfrentan múltiples opciones discretas.

2. Bucles

a) for

El bucle **for** se utiliza cuando conoces de antemano cuántas veces deseas repetir un bloque de código.

Ejemplo:

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

int main() {
    for (int i = 1; i <= 5; i++) {
        cout << "Iteración " << i << endl;
    }

    return 0;
}
```

Explicación:

Este bucle imprime las iteraciones desde 1 hasta 5. El bucle consta de una inicialización (**int i = 1**), una condición (**i <= 5**) y un incremento (**i++**).

En C++, el bucle **for** es muy versátil y se puede utilizar de varias maneras, incluyendo combinaciones con funciones o expresiones como **sizeof**, funciones de C++, e incluso operaciones más complejas dentro de su estructura. A continuación, te muestro algunas combinaciones y usos creativos que puedes realizar con el bucle **for**.

1. Uso de sizeof en el bucle for

El operador **sizeof** devuelve el tamaño en bytes de un tipo de dato o variable. Se puede utilizar en la condición del bucle **for** para recorrer un arreglo de forma automática.

Ejemplo:

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

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int size = sizeof(arr) / sizeof(arr[0]); // Calcular el tamaño del arreglo

    for (int i = 0; i < size; i++) {
        cout << "Elemento " << i << ": " << arr[i] << endl;
    }

    return 0;
}
```

Explicación:

sizeof(arr) devuelve el tamaño total del arreglo en bytes.

sizeof(arr[0]) devuelve el tamaño del primer elemento del arreglo (en este caso, un **int**).

Al dividir ambos valores, obtenemos la cantidad de elementos en el arreglo, que es utilizada para definir el número de iteraciones del bucle.

2. Declaración múltiple en la inicialización del for

En la inicialización de un bucle **for**, puedes declarar varias variables utilizando comas.

Ejemplo:

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

int main() {
    for (int i = 0, j = 10; i < j; i++, j--) {
        cout << "i: " << i << ", j: " << j << endl;
    }

    return 0;
}
```

Explicación:

Aquí se declaran dos variables `i` y `j`.
En cada iteración, `i` aumenta y `j` disminuye, hasta que `i` ya no sea menor que `j`.

3. Uso de funciones en la inicialización, condición o incremento

Es posible llamar funciones directamente en las tres partes del bucle `for`: la inicialización, la condición y la expresión de incremento.

Ejemplo:

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

int doble(int num) {
    return num * 2;
}

bool esPar(int num) {
    return num % 2 == 0;
}

int main() {
    for (int i = 1; esPar(i) == false; i = doble(i)) {
        cout << "Valor de i: " << i << endl;
    }

    return 0;
}
```

Explicación:

La función `doble()` se usa en la parte de incremento para duplicar el valor de `i` en cada iteración. La condición del `for` llama a la función `esPar()`, que verifica si `i` es par.

4. Bucle for infinito

Si deseas un bucle `for` que nunca termine, puedes omitir todas las partes del bucle (***inicialización, condición, y incremento***), dejando simplemente las llaves `{}`.

Ejemplo:

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

int main() {
    for (;;) {
        cout << "Bucle infinito" << endl;
        break; // Romper manualmente para evitar un bucle infinito real
    }

    return 0;
}
```

Explicación:

El bucle se ejecuta indefinidamente hasta que se encuentre con una instrucción `break` o similar. Puedes utilizar `for` junto con punteros y `sizeof` para iterar sobre un arreglo de manera más dinámica.

Ejemplo:

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

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int *ptr = arr; // Apuntar al primer elemento del arreglo

    for (int i = 0; i < sizeof(arr) / sizeof(arr[0]); i++) {
        cout << "Elemento " << i << ": " << *(ptr + i) << endl;
    }

    return 0;
}
```

```
}
```

Explicación:

`ptr` apunta al inicio del arreglo, y se utiliza aritmética de punteros (`*(ptr + i)`) para acceder a cada elemento del arreglo.

6. Usar `auto` para recorrer contenedores de la STL

Si trabajas con contenedores de la **Biblioteca Estándar de C++ (STL)**, puedes usar `auto` para simplificar la declaración de variables en el bucle `for`.

Ejemplo:

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

int main() {
    vector<int> numeros = {10, 20, 30, 40};

    for (auto it = numeros.begin(); it != numeros.end(); ++it) {
        cout << "Número: " << *it << endl;
    }

    return 0;
}
```

Explicación:

`auto` deduce automáticamente el tipo de `it`, que es un **iterador** para el contenedor `vector`. Se recorre el vector utilizando `begin()` y `end()`.

7. `for` con expresiones complejas en cada parte

Puedes combinar cualquier expresión en las tres partes del `for`, como condiciones lógicas, cálculos matemáticos, y llamadas a funciones.

Ejemplo:

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

int main() {
    int suma = 0;
    for (int i = 0, j = 1; i < 10 && j < 100; i++, j += 10) {
        suma += i + j;
        cout << "i: " << i << ", j: " << j << ", suma: " << suma << endl;
    }

    return 0;
}
```

Explicación:

En cada iteración, se incrementa `i` en 1 y `j` en 10. Además, se realiza una operación de suma entre `i` y `j`, que se acumula en `suma`.

8. Uso del rango de `for` (**range-based for loop**)

Este tipo de bucle se introdujo en **C++11** para simplificar la iteración sobre contenedores y arreglos. Utiliza el formato `for (auto elemento : contenedor)`.

Ejemplo:

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

int main() {
    vector<int> numeros = {10, 20, 30, 40};

    for (int num : numeros) {
        cout << "Número: " << num << endl;
    }

    return 0;
}
```

Explicación:

Este bucle recorre automáticamente cada elemento en el **vector numeros**, y **num** representa el valor de cada elemento.

Conclusión

El bucle **for** en **C++** es extremadamente flexible y puede combinarse con diversas expresiones y operadores como **sizeof**, funciones personalizadas, punteros, y más. Estas combinaciones permiten un control más detallado sobre las iteraciones y el flujo del programa.

b) while

El bucle **while** repite un bloque de código mientras una condición sea verdadera.

Ejemplo:

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

int main() {
    int contador = 1;

    while (contador <= 5) {
        cout << "Contador: " << contador << endl;
        contador++;
    }

    return 0;
}
```

Explicación:

El bucle **while** sigue ejecutándose mientras la variable contador sea menor o igual a 5.

c) do-while

Este bucle es similar a **while**, pero garantiza que el código se ejecutará al menos una vez, ya que la condición se evalúa después del primer ciclo.

Ejemplo:

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

int main() {
    int num = 0;

    do {
        cout << "Ingresa un número positivo (o negativo para salir): ";
        cin >> num;
    } while (num >= 0);

    return 0;
}
```

Explicación:

El código solicita al usuario ingresar números positivos y se detiene cuando ingresa uno negativo. La condición se evalúa al final, por lo que el código se ejecuta al menos una vez.

3. Estructuras de Control de Salto

a) break

El **break** se utiliza para salir inmediatamente de un bucle o un **switch**.

Ejemplo:

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

int main() {
    for (int i = 1; i <= 10; i++) {
        if (i == 5) {
            break;
        }
        cout << "Número: " << i << endl;
    }

    return 0;
}
```

Explicación:

El bucle se detiene cuando el valor de i es 5 debido al **break**.

b) continue

El `continue` salta la iteración actual de un bucle y pasa a la siguiente.

Ejemplo:

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

int main() {
    for (int i = 1; i <= 5; i++) {
        if (i == 3) {
            continue;
        }
        cout << "Número: " << i << endl;
    }

    return 0;
}
```

Explicación:

Cuando `i` es 3, el `continue` omite esa iteración y no imprime el número 3.

c) return

El `return` se utiliza para salir de una función, devolviendo opcionalmente un valor.

Ejemplo:

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

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

int main() {
    int resultado = suma(5, 7);
    cout << "Resultado: " << resultado << endl;
    return 0;
}
```

Explicación:

La función `suma` devuelve la suma de dos números y el `return` en el `main` indica que la ejecución del programa terminó correctamente.

Resumen

Condicionales: `if`, `if-else`, `switch`.

Bucles: `for`, `while`, `do-while`.

Control de salto: `break`, `continue`, `return`.

Tanto `sizeof` como `length()` se utilizan para obtener información relacionada con el tamaño de estructuras o contenedores en C++, pero tienen **diferencias** importantes en cuanto a su uso y propósito, especialmente cuando los aplicamos dentro de un bucle `for`. Veamos las diferencias entre ellos:

1. sizeof

El operador `sizeof` se utiliza para obtener el tamaño (en bytes) de un tipo de dato o de un objeto en memoria. Su uso más común es en arreglos o tipos de datos primitivos. Funciona en tiempo de compilación, por lo que no depende del estado dinámico de los datos.

Uso con arreglos:

Ejemplo:

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

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

    // Calcular el número de elementos en el arreglo
    int size = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < size; i++) {
        cout << "Elemento " << i << ": " << arr[i] << endl;
    }

    return 0;
}
```


Características de sizeof:

Tipo de dato primitivo: Se usa comúnmente para obtener el tamaño de un arreglo estático o una variable en bytes.

Tiempo de compilación: Opera en tiempo de compilación, lo que significa que `sizeof` no puede determinar el tamaño de estructuras dinámicas (por ejemplo, arreglos dinámicos).

Funciona solo con arreglos estáticos: No puedes usar `sizeof` para obtener el tamaño de arreglos dinámicos (por ejemplo, punteros o `std::vector`).

Limitaciones:

No se puede aplicar directamente en contenedores dinámicos de la STL como `std::vector` o `std::string`.

Si el arreglo es pasado como un puntero a una función, `sizeof` devolverá el tamaño del puntero, no el número de elementos en el arreglo.

2. length()

`length()` es una función miembro que se utiliza para obtener la longitud de ciertos contenedores, como `std::string`, que devuelve el número de caracteres en la cadena. En `std::vector`, se usa una función similar, `size()`, que devuelve el número de elementos en el vector.

Uso con `std::string`:

Ejemplo:

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

int main() {
    string texto = "Hola Mundo";

    for (int i = 0; i < texto.length(); i++) {
        cout << "Caracter " << i << ": " << texto[i] << endl;
    }

    return 0;
}
```

Uso con std::vector:

Ejemplo:

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

int main() {
    vector<int> numeros = {1, 2, 3, 4, 5};

    for (int i = 0; i < numeros.size(); i++) {
        cout << "Elemento " << i << ": " << numeros[i] << endl;
    }

    return 0;
}
```

Características de length()/size():

Contenedores dinámicos: se usa con cadenas (`std::string`) y `size()` se utiliza en contenedores de la STL como `std::vector`, `std::deque`, y otros.

Tiempo de ejecución: `length()` y `size()` son funciones que se evalúan en tiempo de ejecución. Esto significa que son adecuadas para estructuras dinámicas cuyo tamaño puede cambiar.

Pueden aplicarse a objetos: Se usan en contenedores de la STL, que son clases con funciones miembro.

Limitaciones:

No se pueden usar con tipos de datos primitivos o arreglos estáticos.

Diferencias clave entre sizeof y length() / size()		
Característica	sizeof	length() / size()
Propósito	Obtener el tamaño en bytes de un tipo o arreglo estático.	Obtener el número de elementos en un contenedor dinámico.
Aplicable a	Arreglos estáticos, tipos de datos primitivos.	std::string, std::vector, std::deque, etc.
Evaluación	Tiempo de compilación.	Tiempo de ejecución.
Uso en bucle for	Para iterar sobre arreglos estáticos (usando sizeof(arr) / sizeof(arr[0])).	Para iterar sobre contenedores como std::vector o std::string.
Manejo de datos dinámicos	No puede calcular el tamaño de estructuras dinámicas como punteros o std::vector.	Puede calcular el tamaño de estructuras dinámicas, como std::vector y cadenas de texto.
Tamaño retornado	Retorna el tamaño en bytes.	Retorna el número de elementos.

Ejemplo comparativo: sizeof vs size()
Con sizeof (arreglo estático):

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

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

    // Usando sizeof para obtener el número de elementos
    int size = sizeof(arr) / sizeof(arr[0]);

    for (int i = 0; i < size; i++) {
        cout << "Elemento " << i << ": " << arr[i] << endl;
    }

    return 0;
}
```

Con size() (vector dinámico):

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

int main() {
    vector<int> numeros = {10, 20, 30, 40, 50};

    // Usando size() para obtener el número de elementos
    for (int i = 0; i < numeros.size(); i++) {
        cout << "Elemento " << i << ": " << numeros[i] << endl;
    }

    return 0;
}
```

Conclusión:

Usa sizeof cuando trabajes con arreglos estáticos y necesites calcular el número de elementos en un arreglo en tiempo de compilación.
Usa length() (para cadenas) o size() (para contenedores de la STL) cuando trabajes con estructuras dinámicas cuyo tamaño puede cambiar en tiempo de ejecución.

Biblioteca CMATH:

La biblioteca **cmath** en **C++** es una de las más importantes para realizar operaciones matemáticas avanzadas. Proporciona una amplia gama de funciones matemáticas que son esenciales para cálculos científicos, análisis numérico, gráficos y mucho más.

Características de la Biblioteca **cmath**

- 1. **Operaciones básicas:**
 - Potenciación, raíz cuadrada, valor absoluto.
- 2. **Logaritmos:**
 - Incluye logaritmos en diferentes bases y el logaritmo natural.
- 3. **Operaciones trigonométricas:**
 - Seno, coseno, tangente y sus inversas.
- 4. **Funciones hiperbólicas:**
 - Seno hiperbólico, coseno hiperbólico, tangente hiperbólica y sus inversas.
- 5. **Redondeo y truncamiento:**
 - Funciones como redondear hacia arriba, hacia abajo, truncar, etc.
- 6. **Comparaciones matemáticas:**
 - Buscar el mayor o menor entre números.

Algunas Operaciones que Puedes Hacer

A continuación, te explico con ejemplos prácticos las operaciones que mencionaste y más.

1. Logaritmos

- **log(x):** Logaritmo natural (base *e*).
- **log10(x):** Logaritmo en base 10.

Ejemplo:

```
#include <iostream>
#include <cmath>

int main() {
    double num = 100.0;

    std::cout << "Logaritmo natural de " << num << " es: " << log(num) << std::endl;
    std::cout << "Logaritmo base 10 de " << num << " es: " << log10(num) << std::endl;
    return 0;
}
```

2. Operaciones trigonométricas

- **sin(x), cos(x), tan(x):** Seno, coseno y tangente (en radianes).
- **asin(x), acos(x), atan(x):** Arco seno, arco coseno y arco tangente.

Ejemplo:

```
#include <iostream>
#include <cmath>

int main() {
    double angle = 45.0; // en grados
    double radians = angle * M_PI / 180.0; // Convertir a radianes

    std::cout << "Seno de 45 grados: " << sin(radians) << std::endl;
    std::cout << "Coseno de 45 grados: " << cos(radians) << std::endl;
    std::cout << "Tangente de 45 grados: " << tan(radians) << std::endl;

    return 0;
}
```

3. Raíces (como raíz cúbica)

- `sqrt(x)`: Raíz cuadrada.
- `cbrt(x)`: Raíz cúbica.

Ejemplo:

```
#include <iostream>
#include <cmath>

int main() {
    double num = 27.0;

    std::cout << "Raíz cuadrada de " << num << " es: " << sqrt(num) << std::endl;
    std::cout << "Raíz cúbica de " << num << " es: " << cbrt(num) << std::endl;

    return 0;
}
```

4. Comparaciones (Mayor y Menor)

- `fmax(x, y)`: Devuelve el mayor de dos números.
- `fmin(x, y)`: Devuelve el menor de dos números.

Ejemplo:

```
#include <iostream>
#include <cmath>

int main() {
    double a = 10.5, b = 20.3;

    std::cout << "Mayor entre " << a << " y " << b << " es: " << fmax(a, b) << std::endl;
    std::cout << "Menor entre " << a << " y " << b << " es: " << fmin(a, b) << std::endl;

    return 0;
}
```

5. Redondeo y truncamiento

- `ceil(x)`: Redondea hacia arriba.
- `floor(x)`: Redondea hacia abajo.
- `round(x)`: Redondea al entero más cercano.
- `trunc(x)`: Elimina la parte decimal.

```
#include <iostream>
#include <cmath>

int main() {
    double num = 7.8;

    std::cout << "Redondeo hacia arriba: " << ceil(num) << std::endl;
    std::cout << "Redondeo hacia abajo: " << floor(num) << std::endl;
    std::cout << "Redondeo al entero más cercano: " << round(num) << std::endl;
    std::cout << "Elimina parte decimal: " << trunc(num) << std::endl;

    return 0;
}
```

6. Otras Operaciones útiles

- `pow(base, exponente)`: Potenciación.
- `abs(x)` o `fabs(x)`: Valor absoluto.

Ejemplo:

```
#include <iostream>
#include <cmath>

int main() {
```

```
double base = 2.0, exponente = 3.0;
int num = -5;

std::cout << base << " elevado a la " << exponente << " es: " << pow(base, exponente)
<< std::endl;
std::cout << "Valor absoluto de " << num << " es: " << abs(num) << std::endl;

return 0;
}
```

Diferencia entre cmath y math

cmath:

- Es el encabezado estándar de C++ que incluye las funciones matemáticas de la biblioteca de C (**math.h**).
- Utiliza **overloading** y maneja los nombres dentro del **namespace** std.
- Está diseñado para integrarse completamente con las características modernas de C++.
- Ejemplo:

```
#include <cmath>

std::cout << std::sqrt(16.0); // sqrt está en std
```

math:

- Es el encabezado de C heredado en C++ como **math.h**.
- Las funciones no están dentro del espacio de nombres std (funcionan como en C puro).
- No tiene soporte para **overloading**, lo que significa que podría haber problemas con tipos como **float** o **long double** si no se convierten explícitamente.

Ejemplo

```
#include <math.h>
printf("%f", sqrt(16.0)); // sqrt no está en std
```

Diferencias Clave		
Característica	cmath	math.h
Espacio de nombres	Las funciones están en <code>std</code> (<code>std::sqrt</code>)	Funciones están en el ámbito global (<code>sqrt</code>)
Compatibilidad	Orientada a C++ moderno	Legado de C
Sobrecarga de funciones	Sí	No
Cabecera estándar	Recomendado para C++	Usado en programas C

Ejemplo Comparativo

Con **cmath**:

```
#include <iostream>
#include <cmath>

int main() {
    std::cout << "Raíz cuadrada de 16: " << std::sqrt(16.0) << std::endl;
    return 0;
}
```

Con **math.h**:

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

int main() {
    printf("Raíz cuadrada de 16: %f\n", sqrt(16.0));
    return 0;
}
```

Recomendación

- Usa `cmath` siempre que trabajes en `C++` porque:
 - Se integra con el paradigma orientado a objetos.
 - Permite sobrecarga para trabajar con diferentes tipos numéricos.
 - Está en el estándar de C++.
- Usa `math.h` solo si estás `portando` código de C o `trabajando` en un entorno puro de C.