



## PROGRAMACIÓN III

### **Docente:**

Ing. Jimmy Nataniel Requena Llorentty

### **Integrantes:**

Jaquelin Priscila Bellido Duran

### **Materia:**

Programación III

Santa Cruz - Bolivia

# Clase 1: El Poder de Apuntar

## ¡Entendiendo Punteros y la Memoria!

Hoy aprendimos:

### ¿Qué es un STACK?

Un **stack** es una estructura de datos y también una región de memoria que sigue un orden muy específico para almacenar y recuperar datos.

Se caracteriza por ser: muy rápida, tamaño limitado, se libera automáticamente (cuando la función termina)

### ¿Qué es un HEAP?

Es una zona de memoria usada para almacenar datos dinámicos, es decir, datos que el programa reserva manualmente en tiempo de ejecución

Se caracteriza por ser: Acceso más lento Debes liberar la memoria manualmente (en C/C++ con free o delete) Riesgo de *memory leaks* (si no liberas)

### Ampersand (&) → Operador de dirección

- Se usa para **obtener la dirección de memoria** de una variable.
- Es lo que necesitas para inicializar un puntero.

### Asterisco (\*) → Operador de indirección (o desreferenciación)

- Se usa para **acceder al valor almacenado en la dirección** a la que apunta un puntero.
- También se usa para **declarar punteros**

```
9
10  std::cout << "--- Información de 'variable' ---" << std::endl;
11  std::cout << "Valor de 'variable': " << variable << std::endl;
12  std::cout << "Dirección de 'variable' (&variable): " << &variable << std::endl;
13
14  std::cout << "\n--- Informacion de 'puntero' ---" << std::endl;
15  std::cout << "Contenido de 'puntero' (la direccion que guarda): " << puntero << std::endl;
16  std::cout << "Dirección donde esta guardado el propio 'puntero' (&puntero): " << &puntero << std::endl;
17
18  std::cout << "\n--- Accediendo al valor A TRAVES del puntero ---" << std::endl;
19  std::cout << "Valor al que apunta 'puntero' (*puntero): " << *puntero << std::endl; // DEREFERENCIA
20
21  // Modificando 'variable' A TRAVÉS del puntero
22  std::cout << "\n--- Modificando a traves del puntero ---" << std::endl;
23  *puntero = 30; // Ve a la dirección que guarda 'puntero' y cambia el valor allí a 30
24  std::cout << "Nuevo valor de 'variable' (despues de *puntero = 30): " << variable << std::endl;
25  std::cout << "Nuevo valor apuntado por 'puntero' (*puntero): " << *puntero << std::endl;
26  std::cout << "\nPriscila Bellido - Fin deCodigo" << std::endl;
27
```

input

```
Contenido de 'puntero' (la direccion que guarda): 0x7ffefceabeac
Dirección donde esta guardado el propio 'puntero' (&puntero): 0x7ffefceabeb0

--- Accediendo al valor A TRAVES del puntero ---
Valor al que apunta 'puntero' (*puntero): 20

--- Modificando a traves del puntero ---
Nuevo valor de 'variable' (despues de *puntero = 30): 30
Nuevo valor apuntado por 'puntero' (*puntero): 30

Priscila Bellido - Fin deCodigo

...Program finished with exit code 0
Press ENTER to exit console.
```

# Clase 2: ¡Forjando Memoria a Voluntad!

## new y delete

Hoy aprendimos:

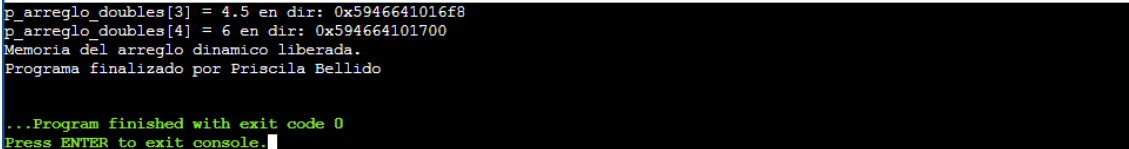
### ¿Qué es un operador NEW?

El **operador new** en C++ se utiliza para **asignar memoria dinámica** en el **heap** (memoria libre), a diferencia de las variables normales que se asignan en la **pila (stack)**. Es decir, cuando no sabes cuánta memoria necesitas hasta que el programa se esté ejecutando, **new** te permite reservar esa memoria en tiempo de ejecución.

### ¿Qué es un operador DELETE?

El **operador delete** en C++ se usa para **liberar** la memoria que fue previamente **asignada dinámicamente con new**. Liberar memoria es **crucial** para evitar **fugas de memoria (memory leaks)**, que ocurren cuando el programa pierde acceso a la memoria asignada pero nunca la libera.

```
28     p_arreglo_doubles[i] = i * 1.5; // Asigna valores al arreglo
29 }
30
31 std::cout << "Arreglo dinamico creado y llenado:" << std::endl;
32 for (int i = 0; i < tamano_arreglo; ++i) {
33     std::cout << "p_arreglo_doubles[" << i << "] = " << p_arreglo_doubles[i]
34         << " en dir: " << (p_arreglo_doubles + i) << std::endl;
35 }
36
37 delete[] p_arreglo_doubles; // ¡IMPORTANTE! Usar delete[] para arreglos
38 p_arreglo_doubles = nullptr; // Buena práctica
39 std::cout << "Memoria del arreglo dinamico liberada." << std::endl;
40 } else {
41     std::cout << "ERROR: No se pudo asignar memoria para p_arreglo_doubles." << std::endl;
42 }
43
44 // Intentar usar un puntero nulo (solo para demostrar, usualmente causa error o comportamiento indefinido)
45 // if (p_entero == nullptr) {
46 //     std::cout << "\np_entero es ahora nullptr." << std::endl;
47 //     // *p_entero = 789; // ¡Esto causaría un error de segmentación! (Descomentar con precaución)
48 // }
49 std::cout << "Programa finalizado por Priscila Bellido" << std::endl;
50
51 return 0;
52
```



```
p_arreglo_doubles[3] = 4.5 en dir: 0x5946641016f8
p_arreglo_doubles[4] = 6 en dir: 0x594664101700
Memoria del arreglo dinamico liberada.
Programa finalizado por Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream> // Para std::cout, std::endl
```

```
int main() {
    // 1. Asignar memoria para un solo entero
    int *p_entero = nullptr; // Siempre inicializar punteros
    p_entero = new int;      // Solicita memoria en el Heap para un int

    if (p_entero != nullptr) { // Buena práctica: verificar si new tuvo
        éxito (aunque suele lanzar excepción)
    }
}
```

```

        *p_entero = 123;        // Asigna un valor a la memoria recién
reservada
        std::cout << "Entero dinamico creado. Valor: " << *p_entero
                << " en direccion: " << p_entero << std::endl;

        delete p_entero;        // Libera la memoria
        p_entero = nullptr;      // ¡Buena práctica! Evita puntero colgante.
        std::cout << "Memoria del entero dinamico liberada." << std::endl;
    } else {
        std::cout << "ERROR: No se pudo asignar memoria para p_entero." <<
std::endl;
    }

    // 2. Asignar memoria para un arreglo de doubles
    std::cout << "\n--- Arreglo Dinamico ---" << std::endl;
    double *p_arreglo_doubles = nullptr;
    int tamano_arreglo = 5;
    p_arreglo_doubles = new double[tamano_arreglo]; // Solicita memoria para
5 doubles

    if (p_arreglo_doubles != nullptr) {
        for (int i = 0; i < tamano_arreglo; ++i) {
            p_arreglo_doubles[i] = i * 1.5; // Asigna valores al arreglo
        }

        std::cout << "Arreglo dinamico creado y llenado:" << std::endl;
        for (int i = 0; i < tamano_arreglo; ++i) {
            std::cout << "p_arreglo_doubles[" << i << "] = " <<
p_arreglo_doubles[i]
                << " en dir: " << (p_arreglo_doubles + i) <<
std::endl;
        }

        delete[] p_arreglo_doubles; // ¡IMPORTANTE! Usar delete[] para
arreglos
        p_arreglo_doubles = nullptr; // Buena práctica
        std::cout << "Memoria del arreglo dinamico liberada." << std::endl;
    } else {
        std::cout << "ERROR: No se pudo asignar memoria para
p_arreglo_doubles." << std::endl;
    }

    // Intentar usar un puntero nulo (solo para demostrar, usualmente causa
error o comportamiento indefinido)
    // if (p_entero == nullptr) {
    //     std::cout << "\np_entero es ahora nullptr." << std::endl;
    //     // *p_entero = 789; // ¡Esto causaría un error de segmentación!
    // }
    // (Descomentar con precaución)
    std::cout << "Programa finalizado por Priscila Bellido" << std::endl;

    return 0;

```

# Clase 3: ¡Construyendo Cadenas de Datos!

## Introducción a Listas Enlazadas

Hoy aprendimos:

### ¿Qué es un ARREGLO (ARRAY)?

Un **arreglo** (también llamado *array*) es una **estructura de datos** que almacena una **colección de elementos del mismo tipo** en posiciones contiguas de memoria.

### ¿Qué es una lista enlazada simple?

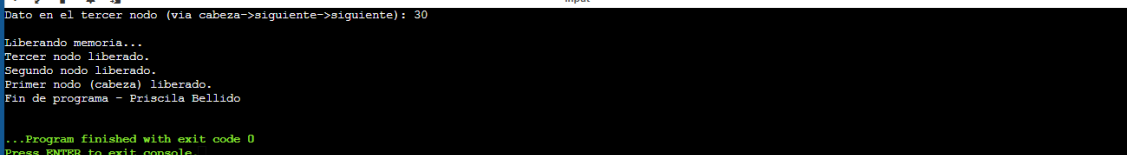
Una **lista enlazada simple** es una estructura de datos **dinámica** que consiste en **nodos conectados entre sí** mediante punteros. A diferencia de los arreglos, **no requiere tamaño fijo**, y puedes insertar/eliminar elementos fácilmente en cualquier posición.

### ¿Qué es un nodo simple?

Un **nodo simple** es la **unidad básica de una lista enlazada simple**. Contiene dos cosas:

1. **Un dato** (el valor que guarda).
2. **Un puntero al siguiente nodo**.

```
34 std::cout << "Dato en cabeza: " << cabeza->dato << std::endl;
35 std::cout << "Dato en el segundo nodo (via cabeza->siguiente): " << cabeza->siguiente->dato << std::endl;
36 std::cout << "Dato en el tercer nodo (via cabeza->siguiente->siguiente): "
37     << cabeza->siguiente->siguiente->dato << std::endl;
38
39 // ¡IMPORTANTE! Liberar la memoria dinámica cuando ya no se necesite
40 // Se debe hacer en orden inverso a con cuidado para no perder punteros
41 std::cout << "\nLiberando memoria..." << std::endl;
42 delete cabeza->siguiente->siguiente; // Borra el tercer nodo (tercerNodo)
43 cabeza->siguiente->siguiente = nullptr; // Buena práctica
44 std::cout << "Tercer nodo liberado." << std::endl;
45
46 delete cabeza->siguiente; // Borra el segundo nodo (segundoNodo)
47 cabeza->siguiente = nullptr; // Buena práctica
48 std::cout << "Segundo nodo liberado." << std::endl;
49
50 delete cabeza; // Borra el primer nodo
51 cabeza = nullptr; // Buena práctica
52 std::cout << "Primer nodo (cabeza) liberado." << std::endl;
53
54 std::cout << "Fin de programa - Priscila Bellido" << std::endl;
55
```



```
#include <iostream>
```

```
struct Nodo {
    int dato;
    Nodo* siguiente;
```

```
    Nodo(int valor_dato) : dato(valor_dato), siguiente(nullptr) {} //
Constructor conciso
};
```

```
int main() {
```

```

    // 1. Crear el primer nodo (cabeza de nuestra mini-lista)
    Nodo* cabeza = new Nodo(10); // Usamos 'new' porque queremos memoria
dinámica
    std::cout << "Creado primer nodo (cabeza) con dato: " << cabeza->dato <<
std::endl;

    // 2. Crear un segundo nodo
    Nodo* segundoNodo = new Nodo(20);
    std::cout << "Creado segundo nodo con dato: " << segundoNodo->dato <<
std::endl;

    // 3. ¡ENLAZARLOS!
    // El puntero 'siguiente' del primer nodo (cabeza) ahora apunta al
segundoNodo
    cabeza->siguiente = segundoNodo;
    std::cout << "Enlazando cabeza->siguiente con segundoNodo." <<
std::endl;

    // 4. Crear un tercer nodo
    Nodo* tercerNodo = new Nodo(30);
    std::cout << "Creado tercer nodo con dato: " << tercerNodo->dato <<
std::endl;

    // 5. Enlazar el segundo nodo con el tercero
    segundoNodo->siguiente = tercerNodo; // 0 cabeza->siguiente->siguiente =
tercerNodo;
    std::cout << "Enlazando segundoNodo->siguiente con tercerNodo." <<
std::endl;

    // ¿Cómo accedemos a los datos ahora?
    std::cout << "\nRecorriendo la mini-lista:" << std::endl;
    std::cout << "Dato en cabeza: " << cabeza->dato << std::endl;
    std::cout << "Dato en el segundo nodo (via cabeza->siguiente): " <<
cabeza->siguiente->dato << std::endl;
    std::cout << "Dato en el tercer nodo (via cabeza->siguiente->siguiente):
"
    << cabeza->siguiente->siguiente->dato << std::endl;

    // ¡IMPORTANTE! Liberar la memoria dinámica cuando ya no se necesite
    // Se debe hacer en orden inverso o con cuidado para no perder punteros
    std::cout << "\nLiberando memoria..." << std::endl;
    delete cabeza->siguiente->siguiente; // Borra el tercer nodo
(tercerNodo)
    cabeza->siguiente->siguiente = nullptr; // Buena práctica
    std::cout << "Tercer nodo liberado." << std::endl;

    delete cabeza->siguiente; // Borra el segundo nodo (segundoNodo)
    cabeza->siguiente = nullptr; // Buena práctica
    std::cout << "Segundo nodo liberado." << std::endl;

    delete cabeza; // Borra el primer nodo
    cabeza = nullptr; // Buena práctica
    std::cout << "Primer nodo (cabeza) liberado." << std::endl;

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;

    return 0;
}

```

# Clase 4: ¡Funciones con Múltiples Talentos!

## La Sobrecarga de Funciones

Hoy aprendimos:

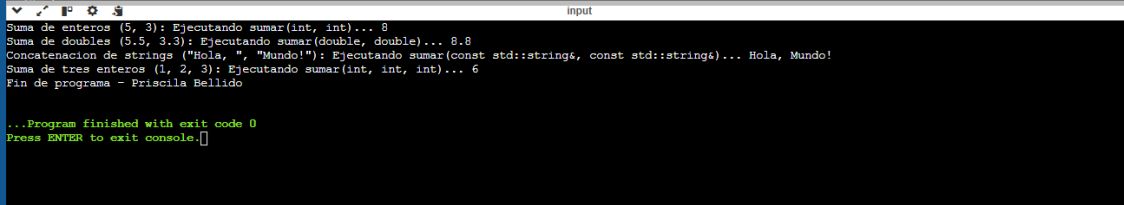
¿Qué es SOBRECARGA?

La **sobrecarga** es una técnica en programación (especialmente en C++) que permite **usar el mismo nombre** para **funciones** o **operadores**, pero con **diferentes comportamientos** según los **argumentos** o el **tipo de datos**.

Ventajas

- Código más **limpio y legible**
- Permite **usar el mismo nombre** para realizar tareas similares
- Esencial en programación orientada a objetos (POO)

```
21     return a + b;
22 }
23
24 // Versión 4: Suma tres enteros
25 // ¡Sobrecargada! Mismo nombre, diferente número de parámetros.
26 int sumar(int a, int b, int c) {
27     std::cout << "Ejecutando sumar(int, int, int)... ";
28     return a + b + c;
29 }
30
31 int main() {
32     std::cout << "Suma de enteros (5, 3): " << sumar(5, 3) << std::endl;
33     std::cout << "Suma de doubles (5.5, 3.3): " << sumar(5.5, 3.3) << std::endl;
34     std::cout << "Concatenacion de strings (\\"Hola, \\", \\"Mundo!\"): "
35     << sumar(std::string("Hola, "), std::string("Mundo!")) << std::endl;
36     std::cout << "Suma de tres enteros (1, 2, 3): " << sumar(1, 2, 3) << std::endl;
37
38     std::cout << "fin de programa - Priscila Bellido" << std::endl;
39     return 0;
40 }
```



```
#include <iostream> // Para std::cout, std::endl
#include <string>    // Para std::string

// Versión 1: Suma dos enteros
int sumar(int a, int b) {
    std::cout << "Ejecutando sumar(int, int)... ";
    return a + b;
}

// Versión 2: Suma dos números de punto flotante (double)
// ¡Sobrecargada! Mismo nombre, diferente tipo de parámetros.
double sumar(double a, double b) {
    std::cout << "Ejecutando sumar(double, double)... ";
    return a + b;
}

// Versión 3: Concatena dos cadenas (std::string)
// ¡Sobrecargada! Mismo nombre, diferente tipo de parámetros.
std::string sumar(const std::string& a, const std::string& b) {
```

```

        std::cout << "Ejecutando sumar(const std::string&, const
std::string&)... ";
        return a + b;
    }

    // Versión 4: Suma tres enteros
    // ¡Sobrecargada! Mismo nombre, diferente número de parámetros.
    int sumar(int a, int b, int c) {
        std::cout << "Ejecutando sumar(int, int, int)... ";
        return a + b + c;
    }

    int main() {
        std::cout << "Suma de enteros (5, 3): " << sumar(5, 3) << std::endl;
        std::cout << "Suma de doubles (5.5, 3.3): " << sumar(5.5, 3.3) <<
std::endl;
        std::cout << "Concatenacion de strings (\"Hola, \", \"Mundo!\"): "
        << sumar(std::string("Hola, "), std::string("Mundo!")) <<
std::endl;
        std::cout << "Suma de tres enteros (1, 2, 3): " << sumar(1, 2, 3) <<
std::endl;

        // Ejemplo de llamada ambigua (si no tuviéramos una versión exacta)
        // Si solo tuviéramos sumar(double, double) y llamáramos sumar(5, 3),
        // los 'int' se promocionarían a 'double'. ¡Pero aquí tenemos una
        exacta!
        // std::cout << "Llamada con promocion (si no hubiera int,int): " <<
sumar(5, (int)3.0) << std::endl;
        // El casteo (int)3.0 no es necesario aquí, solo es para ilustrar.

        std::cout << "Fin de programa - Priscila Bellido" << std::endl;
        return 0;
    }

```



# Clase 5: ¡La Sobrecarga en Acción!

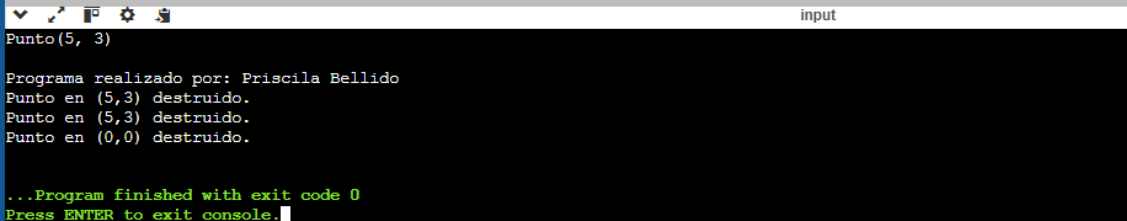
## Aplicaciones y Utilidades practicas

Hoy aprendimos:

¿Qué son constructores?

Un **constructor** es una función **especial** dentro de una **clase** que se **ejecuta automáticamente** cuando se **crea un objeto**. Sirve para **inicializar atributos** y preparar el objeto para su uso.

```
30 }
31 };
32
33 // Función main para probar la clase
34 int main() {
35     std::cout << "=== Demostrando constructores ===" << std::endl;
36
37     // Constructor por defecto
38     Punto p1;
39
40     // Constructor con coordenadas
41     Punto p2(5.0, 3.0);
42
43     // Constructor de copia
44     Punto p3(p2);
45
46     std::cout << "\n=== Mostrando puntos ===" << std::endl;
47     p1.mostrar();
48     p2.mostrar();
49     p3.mostrar();
50
51     std::cout << "\nPrograma realizado por: Priscila Bellido" << std::endl;
52 }
```



```
#include <iostream> // ¡Esta línea es la que faltaba!
```

```
class Punto {
public:
    double x, y;

    // Constructor 1: Por defecto (en el origen)
    Punto() : x(0.0), y(0.0) {
        std::cout << "Punto creado en el origen (0,0) por constructor por defecto." << std::endl;
    }

    // Constructor 2: Con coordenadas específicas
    Punto(double coord_x, double coord_y) : x(coord_x), y(coord_y) {
        std::cout << "Punto creado en (" << x << ", " << y << ") por constructor con coords." << std::endl;
    }
}
```

```

    // Constructor 3: Copia (se genera uno por defecto, pero podemos hacerlo
    explícito)
    Punto(const Punto& otroPunto) : x(otroPunto.x), y(otroPunto.y) {
        std::cout << "Punto copiado de (" << otroPunto.x << ", " <<
otroPunto.y << ")." << std::endl;
    }

    // Destructor (opcional, para demostrar el ciclo de vida)
    ~Punto() {
        std::cout << "Punto en (" << x << ", " << y << ") destruido." <<
std::endl;
    }

    // Método para mostrar las coordenadas
    void mostrar() const {
        std::cout << "Punto(" << x << ", " << y << ")" << std::endl;
    }
};

// Función main para probar la clase
int main() {
    std::cout << "=== Demostrando constructores ===" << std::endl;

    // Constructor por defecto
    Punto p1;

    // Constructor con coordenadas
    Punto p2(5.0, 3.0);

    // Constructor de copia
    Punto p3(p2);

    std::cout << "\n=== Mostrando puntos ===" << std::endl;
    p1.mostrar();
    p2.mostrar();
    p3.mostrar();

    std::cout << "\nPrograma realizado por: Priscila Bellido" << std::endl;

    return 0;
}

```

## ¿Por qué sobrecargar?

La **sobrecarga** permite **definir varias versiones de una misma función** (o constructor) con **diferentes parámetros**. Esto mejora la **flexibilidad, claridad del código** y **reutilización**.

## ¿Cuándo deberías sobrecargar?

- Quieres **una misma acción** que varía según los **argumentos**.
- Buscas **hacer tu clase o función más flexible**.
- Necesitas versiones con **valores por defecto** o inicializaciones distintas.

## ¿Qué son las colecciones polimórficas?

Una **colección polimórfica** es una **estructura de datos** (como una lista, vector, arreglo, etc.) que puede **almacenar objetos de diferentes tipos derivados** de una misma clase base, y **tratarlos de forma genérica** usando **polimorfismo**.

Esto permite trabajar con objetos de distintas clases (que comparten herencia) **a través de punteros o referencias a la clase base**.

## ¿Qué son los smart pointers?

Los smart pointers o punteros inteligentes son objetos de la STL (Standard Template Library) que gestionan automáticamente la memoria dinámica en C++. Su objetivo es evitar fugas de memoria (memory leaks) y errores como dobles liberaciones o accesos a memoria liberada.

## Mostrar completo:

```
42
43 void mostrar(double valor) {
44     std::cout << "Tipo Decimal (double): " << valor << std::endl;
45 }
46
47 void mostrar(const std::string& valor) {
48     std::cout << "Tipo Cadena (std::string): \"" << valor << "\"" << std::endl;
49 }
50
51 void mostrar(char valor) {
52     std::cout << "Tipo Caracter (char): '" << valor << "'" << std::endl;
53 }
54
55 void mostrar(const std::vector<int>& miVector) {
56     std::cout << "Tipo Vector de Enteros (std::vector<int>): [ ";
57     for (size_t i = 0; i < miVector.size(); ++i) {
58         std::cout << miVector[i] << (i == miVector.size() - 1 ? "" : ", ");
59     }
60     std::cout << "]" << std::endl;
61 }
```

input

```
--- Demostracion de 'mostrar' sobrecargado ---
Tipo Entero (int): 100
Tipo Decimal (double): 3.14159
Tipo Cadena (std::string): "Hola desde Programacion III!"
Tipo Caracter (char): 'Z'
Tipo Vector de Enteros (std::vector<int>): [ 10, 20, 30, 40, 50 ]
Tipo Cadena (std::string): "Esto es un literal de C-string."
Tipo Cadena (std::string): "Priscila Bellido - Fin del codigo"

...Program finished with exit code 0
Press ENTER to exit console.
```

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

// Declaraciones de las funciones 'mostrar' (prototipos)
void mostrar(int valor);
void mostrar(double valor);
void mostrar(const std::string& valor);
void mostrar(char valor);
void mostrar(const std::vector<int>& miVector);

int main() {
    std::cout << "--- Demostracion de 'mostrar' sobrecargado ---" <<
    std::endl;
```

```

mostrar(100);
mostrar(3.14159);
mostrar(std::string("Hola desde Programacion III!"));
mostrar('Z');

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

// Llamada con un literal de cadena de C (const char*)
// El compilador puede convertirlo a std::string si no hay otra
sobrecarga mejor
mostrar("Esto es un literal de C-string.");
mostrar("Priscila Bellido - Fin del codigo");

return 0;
}

// Implementaciones de las funciones 'mostrar'
void mostrar(int valor) {
    std::cout << "Tipo Entero (int): " << valor << std::endl;
}

void mostrar(double valor) {
    std::cout << "Tipo Decimal (double): " << valor << std::endl;
}

void mostrar(const std::string& valor) {
    std::cout << "Tipo Cadena (std::string): \"" << valor << "\"" <<
std::endl;
}

void mostrar(char valor) {
    std::cout << "Tipo Character (char): '" << valor << "'" << std::endl;
}

void mostrar(const std::vector<int>& miVector) {
    std::cout << "Tipo Vector de Enteros (std::vector<int>): [ ";
    for (size_t i = 0; i < miVector.size(); ++i) {
        std::cout << miVector[i] << (i == miVector.size() - 1 ? "" : ", ");
    }
    std::cout << " ]" << std::endl;
}

```

# Clase 6: ¡El Arte de las Funciones que se Piensan a Sí Mismas

**Hoy aprendimos:**

**¿Qué es la recursividad?**

La **recursividad** es una técnica donde una **función se llama a sí misma** para resolver un problema más grande dividiéndolo en **subproblemas más pequeños**.

**¿Por qué se usa?**

Se utiliza cuando un problema puede **dividirse en partes similares a sí mismo**, lo que permite una solución elegante y más natural en algunos casos (como estructuras de árbol, matemáticas, etc.).

**Función Recursiva: Sus Partes Vitales**

Una función recursiva tiene 3 componentes esenciales que garantizan su funcionamiento correcto y seguro:

**Caso Base (o condición de parada)**

Es la condición que detiene la recursión.

Evita que la función se llame infinitamente.

Siempre debe resolverse sin llamadas recursivas.

**Paso Recursivo (la llamada a sí misma)**

- Es el corazón de la recursividad.
- La función se **llama a sí misma**, pero con un problema **más pequeño o más simple**.

**Progresión hacia el caso base**

- Es la **garantía de que nos acercamos** al caso base en cada llamada.
- Si no hay progresión, la recursión será infinita.

# ¡La Recursividad en Acción! Calculando el Factorial

```
17     long long resultadoRecursion = factorial(n - 1); // Llamada recursiva
18     long long resultadoFinal = n * resultadoRecursion;
19
20     std::cout << " factorial(" << n << ") -> Retornando " << n << " * "
21     << resultadoRecursion << " = " << resultadoFinal << std::endl;
22
23     return resultadoFinal;
24 }
25 }
26
27 int main() {
28     int numero = 4; // Probar con 4!
29     std::cout << "Iniciando calculo del factorial de " << numero << "." << std::endl;
30
31     long long resultado = factorial(numero);
32
33     std::cout << "\nEl factorial de " << numero << " es: " << resultado << std::endl;
34
35     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
36
37     return 0;
}
```

input

```
Calculando factorial(1)...
factorial(1) -> Caso Base! Retorna 1.
factorial(2) -> Retornando 2 * 1 = 2
factorial(3) -> Retornando 3 * 2 = 6
factorial(4) -> Retornando 4 * 6 = 24

El factorial de 4 es: 24
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>

// Función recursiva para calcular el factorial
long long factorial(int n) {
    std::cout << "Calculando factorial(" << n << ")..." << std::endl;

    // Caso Base
    if (n == 0 || n == 1) {
        std::cout << " factorial(" << n << ") -> Caso Base! Retorna 1." <<
std::endl;
        return 1;
    }
    // Paso Recursivo
    else {
        std::cout << " factorial(" << n << ") -> Paso Recursivo. Llama a
factorial("
            << (n - 1) << ")." << std::endl;

        long long resultadoRecursion = factorial(n - 1); // Llamada
recursiva
        long long resultadoFinal = n * resultadoRecursion;

        std::cout << " factorial(" << n << ") -> Retornando " << n << " * "
            << resultadoRecursion << " = " << resultadoFinal <<
std::endl;

        return resultadoFinal;
    }
}
```

```
int main() {
    int numero = 4; // Probar con 4!
    std::cout << "Iniciando calculo del factorial de " << numero << "." <<
std::endl;

    long long resultado = factorial(numero);

    std::cout << "\nEl factorial de " << numero << " es: " << resultado <<
std::endl;

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;

    return 0;
}
```

# Clase 7: ¡La Recursividad se Expande!

## Aplicaciones y Análisis de Eficiencia

Hoy aprendimos:

¿Qué es la secuencia de Fibonacci?

La **secuencia de Fibonacci** es una serie de números donde cada número es la **suma de los dos anteriores**.

### Fibonacci: La Naturaleza Hecha Números (y Recursión)

```
13     return fibonacci(n - 1) + fibonacci(n - 2);
14 }
15
16 int main() {
17     int terminos = 7; // Calcular hasta F(6)
18     std::cout << "Secuencia de Fibonacci (primeros " << terminos << " terminos):" << std::endl;
19
20     for (int i = 0; i < terminos; ++i) {
21         std::cout << fibonacci(i) << " ";
22     }
23
24     std::cout << std::endl;
25
26     // Prueba adicional
27     std::cout << "F(7) es: " << fibonacci(7) << std::endl;
28
29     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
30
31     return 0;
32 }
```

input

```
Secuencia de Fibonacci (primeros 7 terminos):
0 1 1 2 3 5 8
F(7) es: 13
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>
```

```
// Función Fibonacci recursiva
```

```
int fibonacci(int n) {
    // Casos base
    if (n <= 0) {
        return 0;
    }
    if (n == 1) {
        return 1;
    }
    // Paso recursivo
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

```
int main() {
    int terminos = 7; // Calcular hasta F(6)
    std::cout << "Secuencia de Fibonacci (primeros " << terminos << "
terminos):" << std::endl;
```



```

    for (int i = 0; i < terminos; ++i) {
        std::cout << fibonacci(i) << " ";
    }

    std::cout << std::endl;

    // Prueba adicional
    std::cout << "F(7) es: " << fibonacci(7) << std::endl;

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;

    return 0;
}

```

## Sumando en Cadena: Recursión con Arreglos

```

5 // Caso Base: Si el índice está fuera de los límites del vector,
6 // significa que no hay más elementos que sumar.
7 if (idx >= arr.size()) {
8     return 0;
9 }
10 // Paso Recursivo: Suma el elemento actual (arr[idx])
11 // con la suma del resto del arreglo (desde idx + 1).
12 else {
13     return arr[idx] + sumarArreglo(arr, idx + 1);
14 }
15 }
16 int main() {
17     std::vector<int> misNumeros = {10, 5, 15, 20, 50}; // Suma = 100
18     int sumaTotal = sumarArreglo(misNumeros, 0); // Empezar desde el índice 0
19     std::cout << "La suma recursiva del arreglo es: " << sumaTotal <<
20     std::endl;
21
22     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
23
24     return 0;
25 }

```

input

```

La suma recursiva del arreglo es: 100
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.

```

```

#include <iostream>
#include <vector>
// Suma los elementos de 'arr' desde el índice 'idx' hasta el final
int sumarArreglo(const std::vector<int>& arr, int idx) {
    // Caso Base: Si el índice está fuera de los límites del vector,
    // significa que no hay más elementos que sumar.
    if (idx >= arr.size()) {
        return 0;
    }
    // Paso Recursivo: Suma el elemento actual (arr[idx])
    // con la suma del resto del arreglo (desde idx + 1).
    else {
        return arr[idx] + sumarArreglo(arr, idx + 1);
    }
}
int main() {

```

```
std::vector<int> misNumeros = {10, 5, 15, 20, 50}; // Suma = 100
int sumaTotal = sumarArreglo(misNumeros, 0); // Empezar desde el índice 0
std::cout << "La suma recursiva del arreglo es: " << sumaTotal <<
std::endl;

std::cout << "Fin de programa - Priscila Bellido" << std::endl;

return 0;

}
```

# Clase 8: ¡Arquitectos de la Recursión!

## Diseñando Nuestras Propias Soluciones

Hoy aprendimos:

¿Qué es Diseño Recursivo?

El **diseño recursivo** es una técnica para resolver problemas dividiéndolos en subproblemas más pequeños y similares, hasta llegar a un caso base trivial que puede resolverse directamente.

```
15 // std::cout << " Primer caracter: '" << primerCaracter
16 // << "'", Resto: '\"' << restoDeLaCadena << "\"' << std::endl;
17 std::string restoInvertido = invertirRecursiva(restoDeLaCadena); // ¡Fe recursiva!
18 // std::cout << " Resto invertido: '\"' << restoInvertido << "\"' << std::endl;
19 return restoInvertido + primerCaracter; // Combinar
20 }
21 }
22 int main() {
23     std::string original = "recursividad";
24     std::string invertida = invertirRecursiva(original);
25     std::cout << "Original: " << original << std::endl;
26     std::cout << "Invertida (Recursiva): " << invertida << std::endl;
27     // Para comparar (no es parte de la solución recursiva)
28     std::string comparacion = original;
29     std::reverse(comparacion.begin(), comparacion.end());
30     std::cout << "Invertida (con std::reverse): " << comparacion << std::endl;
31     std::cout << "Probando con 'abc': " << invertirRecursiva("abc") << std::endl;
32     std::cout << "Probando con 'a': " << invertirRecursiva("a") << std::endl;
33     std::cout << "Probando con '\"': " << invertirRecursiva("") << std::endl;
34     |
35     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
```

input

```
Original: recursividad
Invertida (Recursiva): dadivisruocer
Invertida (con std::reverse): dadivisruocer
Probando con 'abc': cba
Probando con 'a': a
Probando con '\"': 
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>
#include <string>
#include <algorithm> // Para std::reverse (solo para comparar)
std::string invertirRecursiva(const std::string& s) {
    // std::cout << "Llamada con: '\"' << s << "\"' << std::endl; // Para traza
    // Caso Base: Cadena vacía o de un solo carácter
    if (s.length() <= 1) {
        // std::cout << " Caso Base, retorna: '\"' << s << "\"' << std::endl;
        return s;
    }
    // Paso Recursivo:
    else {
        char primerCaracter = s[0];
        std::string restoDeLaCadena = s.substr(1);
        // std::cout << " Primer caracter: '" << primerCaracter
        // << "'", Resto: '\"' << restoDeLaCadena << "\"' << std::endl;
        std::string restoInvertido = invertirRecursiva(restoDeLaCadena); // ¡Fe
recursiva!
        // std::cout << " Resto invertido: '\"' << restoInvertido << "\"' <<
std::endl;
        return restoInvertido + primerCaracter; // Combinar
    }
}
```

```

    }
}
int main() {
    std::string original = "recursividad";
    std::string invertida = invertirRecursiva(original);
    std::cout << "Original: " << original << std::endl;
    std::cout << "Invertida (Recursiva): " << invertida << std::endl;
    // Para comparar (no es parte de la solución recursiva)
    std::string comparacion = original;
    std::reverse(comparacion.begin(), comparacion.end());
    std::cout << "Invertida (con std::reverse): " << comparacion << std::endl;
    std::cout << "Probando con 'abc': " << invertirRecursiva("abc") <<
std::endl;
    std::cout << "Probando con 'a': " << invertirRecursiva("a") << std::endl;
    std::cout << "Probando con \"\": " << invertirRecursiva("") << std::endl;

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;
    return 0;
}

```

# Clase 9: ¡Un Nuevo Universo! Introducción a la Programación Orientada a Objetos (POO)

Hoy aprendimos:

## ¿Qué es Diseño Recursivo?

POO significa Programación Orientada a Objetos. Es un paradigma de programación que organiza el código en torno a objetos en lugar de funciones o lógica pura.

## ¿Qué es un objeto?

Un objeto es una **unidad que contiene datos (atributos) y comportamientos (métodos)** relacionados. Es una representación en código de algo del mundo real o conceptual.

## ¿Qué es una clase?

Piensa en una clase como el **plano de una casa**. No es una casa real, pero describe cómo será: cuántas habitaciones, de qué tamaño, etc. A partir de ese plano, puedes construir muchas casas (**objetos**) con sus propias características.

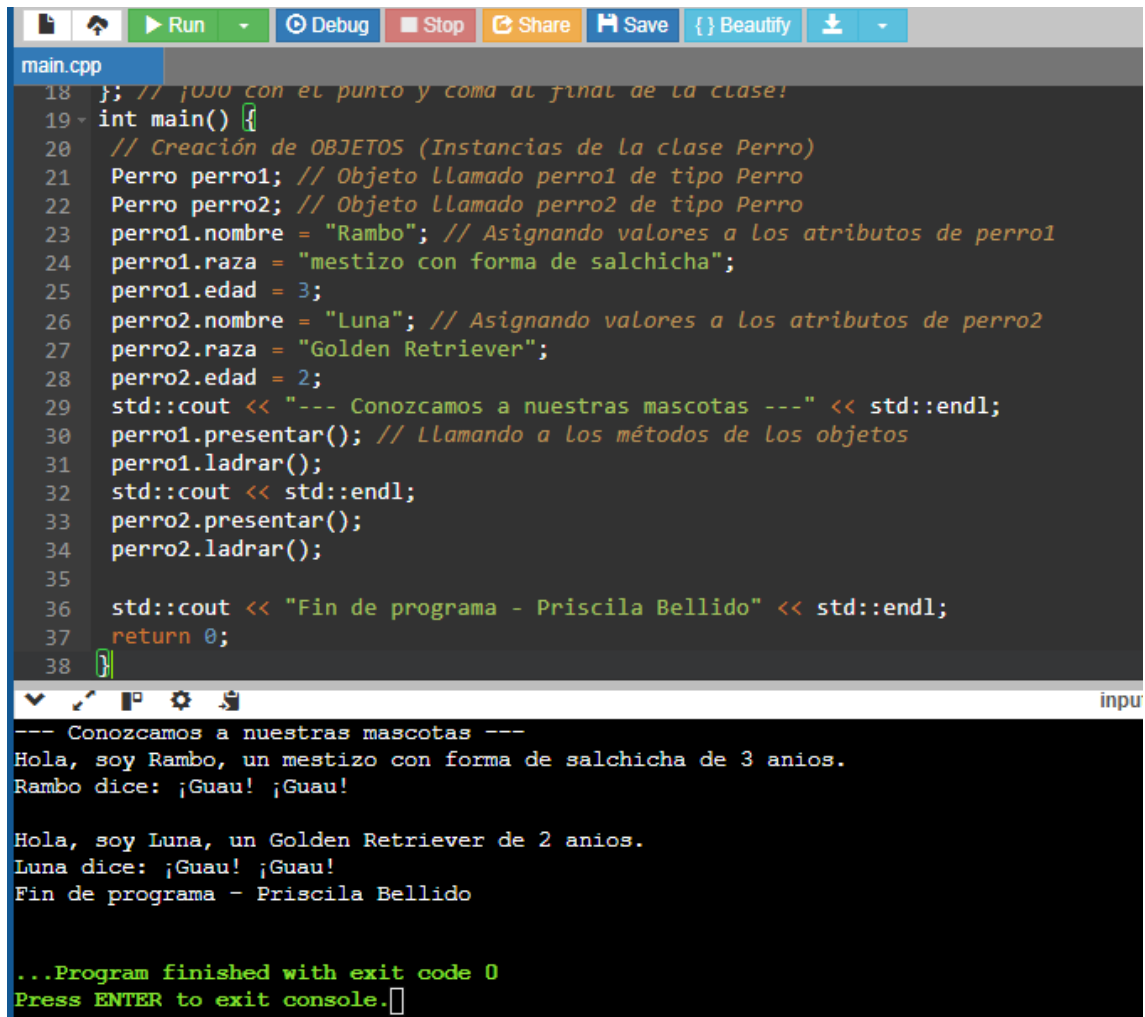
## Atributos

- También llamados **propiedades** o **campos**.
- Son **variables** que almacenan datos dentro de un objeto.
- Representan el **estado** o **características** del objeto.

## Métodos

- Son **funciones** dentro de la clase.
- Definen lo que el objeto **puede hacer** o cómo puede **comportarse**.
- Pueden acceder y modificar los atributos del objeto.

## ¡Creando mascotas!



```
main.cpp
18 }; // ¡OJO con el punto y coma al final de la clase!
19 int main() {
20     // Creación de OBJETOS (Instancias de la clase Perro)
21     Perro perro1; // Objeto llamado perro1 de tipo Perro
22     Perro perro2; // Objeto llamado perro2 de tipo Perro
23     perro1.nombre = "Rambo"; // Asignando valores a los atributos de perro1
24     perro1.raza = "mestizo con forma de salchicha";
25     perro1.edad = 3;
26     perro2.nombre = "Luna"; // Asignando valores a los atributos de perro2
27     perro2.raza = "Golden Retriever";
28     perro2.edad = 2;
29     std::cout << "--- Conozcamos a nuestras mascotas ---" << std::endl;
30     perro1.presentar(); // Llamando a los métodos de los objetos
31     perro1.ladrrar();
32     std::cout << std::endl;
33     perro2.presentar();
34     perro2.ladrrar();
35
36     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
37     return 0;
38 }
```

input

```
--- Conozcamos a nuestras mascotas ---
Hola, soy Rambo, un mestizo con forma de salchicha de 3 años.
Rambo dice: ¡Guau! ¡Guau!

Hola, soy Luna, un Golden Retriever de 2 años.
Luna dice: ¡Guau! ¡Guau!
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>
#include <string>
// Definición de la CLASE 'Perro'
class Perro {
public: // Especificador de acceso (lo veremos más adelante)
    // Atributos (Variables Miembro)
    std::string nombre;
    std::string raza;
    int edad;
    // Métodos (Funciones Miembro)
    void ladrrar() {
        std::cout << nombre << " dice: ¡Guau! ¡Guau!" << std::endl;
    }
    void presentar() {
        std::cout << "Hola, soy " << nombre << ", un " << raza
        << " de " << edad << " años." << std::endl;
    }
}; // ¡OJO con el punto y coma al final de la clase!
int main() {
    // Creación de OBJETOS (Instancias de la clase Perro)
    Perro perro1; // Objeto llamado perro1 de tipo Perro
    Perro perro2; // Objeto llamado perro2 de tipo Perro
```

```

perro1.nombre = "Rambo"; // Asignando valores a los atributos de perro1
perro1.raza = "mestizo con forma de salchicha";
perro1.edad = 3;
perro2.nombre = "Luna"; // Asignando valores a los atributos de perro2
perro2.raza = "Golden Retriever";
perro2.edad = 2;
std::cout << "--- Conozcamos a nuestras mascotas ---" << std::endl;
perro1.presentar(); // Llamando a los métodos de los objetos
perro1.ladRAR();
std::cout << std::endl;
perro2.presentar();
perro2.ladRAR();

std::cout << "Fin de programa - Priscila Bellido" << std::endl;
return 0;
}

```

### Los cuatro pilares de la Programación Orientada a Objetos (POO):

- **Abstracción:** mostrar solo lo esencial y ocultar lo complicado.
- **Encapsulamiento:** proteger los datos agrupándolos con los métodos en una clase.
- **Herencia:** crear nuevas clases a partir de otras para reutilizar código.
- **Polimorfismo:** permitir que diferentes objetos respondan de forma distinta al mismo mensaje.

# Clase 10: ¡Protegiendo Tesoros! Ventajas de POO y el Poder del Encapsulamiento

## Hoy aprendimos:

Los beneficios de usar la **Programación Orientada a Objetos (POO)** en el desarrollo de software:

- **Modularidad:** el programa se divide en partes (clases) más fáciles de entender y mantener.
- **Reutilización de código:** una clase bien hecha se puede usar en varios lugares sin reescribir.
- **Mantenibilidad y escalabilidad:** se pueden hacer cambios o agregar funciones sin dañar el resto del sistema.
- **Abstracción:** se representan objetos del mundo real de forma clara y sencilla.
- **Encapsulamiento:** se protege la información interna de los objetos, mejorando seguridad y organización.

## ¿Qué es encapsulamiento?

El encapsulamiento consiste en agrupar los datos y métodos de un objeto dentro de una clase, y ocultar los detalles internos para proteger la información.

Este principio ayuda a:

- **Proteger los datos** del acceso incorrecto desde fuera de la clase.
- **Facilitar el mantenimiento** del código, permitiendo cambios internos sin afectar al resto.
- **Reducir la complejidad**, mostrando solo lo necesario al usuario.
- **Mejorar la seguridad**, ocultando partes sensibles del objeto.

## PUBLIC Y PRIVATE

En C++, usamos `public` y `private` para **controlar qué partes de una clase son accesibles desde fuera**:

- **`**public**`:** permite el acceso desde cualquier parte del programa. Se usa para los **métodos** que forman la interfaz pública del objeto.
- **`**private**`:** restringe el acceso solo al interior de la clase. Se usa comúnmente para **atributos**, protegiendo los datos y aplicando el **encapsulamiento**.

## Getters y Setters: Guardianes de la Información

Cuando los atributos de una clase son **private**, usamos **getters** y **setters** para acceder a ellos de forma segura:

- **Getters** (`getAtributo()`): permiten **leer** el valor de un atributo sin modificarlo.
- **Setters** (`setAtributo(valor)`): permiten **modificar** el valor de un atributo de forma **controlada**, incluso con **validación**.

Estos métodos mantienen la protección del objeto mientras permiten interactuar con sus datos desde fuera.



## Encapsulando al Estudiante

```
main.cpp
49 }
50 // Otros métodos públicos
51 void mostrarInformacion() const {
52     std::cout << "-----" << std::endl;
53     std::cout << "Nombre: " << nombre << std::endl;
54     std::cout << "Edad: " << edad << " años" << std::endl;
55     std::cout << "Matricula: " << matricula << std::endl;
56     std::cout << "Promedio: " << promedio << std::endl;
57     std::cout << "-----" << std::endl;
58 }
59 }; // Fin de la clase Estudiante
60 int main() {
61     Estudiante estudiante1("Juaquin Soliz", 20, "A123");
62     estudiante1.mostrarInformacion();
63     // Intentando modificar datos
64     // estudiante1.edad = 21; // ;ERROR! 'edad' es private.
65     std::cout << "\nIntentando actualizar edad y promedio..." << std::endl;
66     estudiante1.setEdad(21); // Uso correcto del setter
67     estudiante1.setPromedio(8.5);
68     estudiante1.setEdad(150); // Intentando edad inválida
69     estudiante1.setPromedio(-2.0); // Intentando promedio inválido
70     std::cout << "\nInformacion actualizada de " << estudiante1.getNombre() << ":" << std::endl;
71     estudiante1.mostrarInformacion();
72     Estudiante estudiante2("Priscila Vaca", -10, "B456"); // Prueba de validación en constructor
73     estudiante2.mostrarInformacion(); // Veremos qué edad tiene Luis
74
75     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
76     return 0;
77 }
78
```

```
Estudiante 'Juaquin Soliz' creado.
-----
Nombre: Juaquin Soliz
Edad: 20 años
Matricula: A123
Promedio: 0
-----

Intentando actualizar edad y promedio...
Error: Edad '150' invalida para el estudiante Juaquin Soliz. Edad no modificada.
Error: Promedio '-2' invalido para Juaquin Soliz. Promedio no modificado.

Informacion actualizada de Juaquin Soliz:
-----
Nombre: Juaquin Soliz
Edad: 21 años
Matricula: A123
Promedio: 8.5
-----

Error: Edad '-10' invalida para el estudiante Priscila Vaca. Edad no modificada.
Estudiante 'Priscila Vaca' creado.
-----
Nombre: Priscila Vaca
Edad: 979427408 años
Matricula: B456
Promedio: 0
-----

Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```

#include <iostream>
#include <string>
class Estudiante {
private: // DATOS PROTEGIDOS
    std::string nombre;
    int edad;
    std::string matricula;
    double promedio;
public:
    // Constructor
    Estudiante(std::string nom, int ed, std::string matr) {
        nombre = nom;
        // Usamos el setter para edad para aprovechar la validación
        setEdad(ed); // ¡Llamando a nuestro propio setter!
        matricula = matr;
        promedio = 0.0; // Promedio inicial
        std::cout << "Estudiante '" << nombre << "' creado." << std::endl;
    }
    // Getters
    std::string getNombre() const { return nombre; }
    int getEdad() const { return edad; }
    std::string getMatricula() const { return matricula; }
    double getPromedio() const { return promedio; }
    // Setters (con validación donde aplique)
    void setNombre(const std::string& nuevoNombre){
        if (!nuevoNombre.empty()) {
            nombre = nuevoNombre;
        } else {
            std::cout << "Error: El nombre no puede estar vacio." <<
std::endl;
        }
    }
    void setEdad(int nuevaEdad) {
        if (nuevaEdad >= 5 && nuevaEdad <= 100) { // Ejemplo de validación
            edad = nuevaEdad;
        } else {
            std::cout << "Error: Edad '" << nuevaEdad << "' invalida para el
estudiante "
                << nombre << ". Edad no modificada." << std::endl;
        }
    }
    // No ponemos un setter para matricula, asumiendo que no cambia.
    // Pero podríamos tener uno si fuera necesario.
    void setPromedio(double nuevoPromedio) {
        if (nuevoPromedio >= 0.0 && nuevoPromedio <= 10.0) { // Asumiendo
escala 0-10
            promedio = nuevoPromedio;
        } else {
            std::cout << "Error: Promedio '" << nuevoPromedio << "' invalido
para "
                << nombre << ". Promedio no modificado." << std::endl;
        }
    }
    // Otros métodos públicos
    void mostrarInformacion() const {
        std::cout << "-----" << std::endl;
        std::cout << "Nombre: " << nombre << std::endl;
        std::cout << "Edad: " << edad << " años" << std::endl;
        std::cout << "Matricula: " << matricula << std::endl;
        std::cout << "Promedio: " << promedio << std::endl;
        std::cout << "-----" << std::endl;
    }
}; // Fin de la clase Estudiante

```

```

int main() {
    Estudiante estudiante1("Juaquin Soliz", 20, "A123");
    estudiante1.mostrarInformacion();
    // Intentando modificar datos
    // estudiante1.edad = 21; // ¡ERROR! 'edad' es private.
    std::cout << "\nIntentando actualizar edad y promedio..." << std::endl;
    estudiante1.setEdad(21); // Uso correcto del setter
    estudiante1.setPromedio(8.5);
    estudiante1.setEdad(150); // Intentando edad inválida
    estudiante1.setPromedio(-2.0); // Intentando promedio inválido
    std::cout << "\nInformacion actualizada de " << estudiante1.getNombre()
    << ":" << std::endl;
    estudiante1.mostrarInformacion();
    Estudiante estudiante2("Priscila Vaca", -10, "B456"); // Prueba de
    validación en constructor
    estudiante2.mostrarInformacion(); // Veremos qué edad tiene Luis

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;
    return 0;
}

```

### ¿Qué ganamos encapsulando al Estudiante?

- **Protección de datos:** evitamos asignar valores inválidos (como edad negativa) usando validaciones dentro de los setters.
- **Flexibilidad:** podemos cambiar cómo se guardan o calculan datos internamente sin afectar el código externo, siempre que la interfaz pública se mantenga.
- **Interfaz clara:** los usuarios solo interactúan con los métodos públicos, sin preocuparse por los detalles internos.
- **Mejor mantenibilidad:** los errores relacionados con datos se localizan dentro de la clase, facilitando la corrección sin afectar todo el programa.

### Secretos Más Profundos del Encapsulamiento

- **protected:** acceso restringido para clases derivadas, no para el resto del programa.
- **Clases y funciones friend:** permiten acceso especial a miembros privados o protegidos, pero se usan con cuidado.
- **Principio de menor privilegio:** dar solo los permisos necesarios para mejorar la seguridad y el control.
- **Invariantes de clase:** reglas que garantizan que los datos del objeto siempre estén en un estado válido, mantenidas por setters y constructores.

# Clase 11: Abstracción, Reusabilidad, y los Ciclos de Vida de los Objetos (Constructores y Destructores)

Hoy aprendimos:

## ¿Qué es la abstracción?

- La abstracción es mostrar solo lo esencial de un objeto y ocultar los detalles complejos de cómo funciona internamente.
- Se centra en **qué hace** el objeto, no en **cómo** lo hace.
- Ayuda a reducir la complejidad, facilita la reutilización y mejora el mantenimiento del código.
- Ejemplo real: conducir un auto usando solo volante y pedales, sin preocuparse por el motor.
- En POO, la interfaz pública de la clase es la abstracción, y el encapsulamiento oculta los detalles.

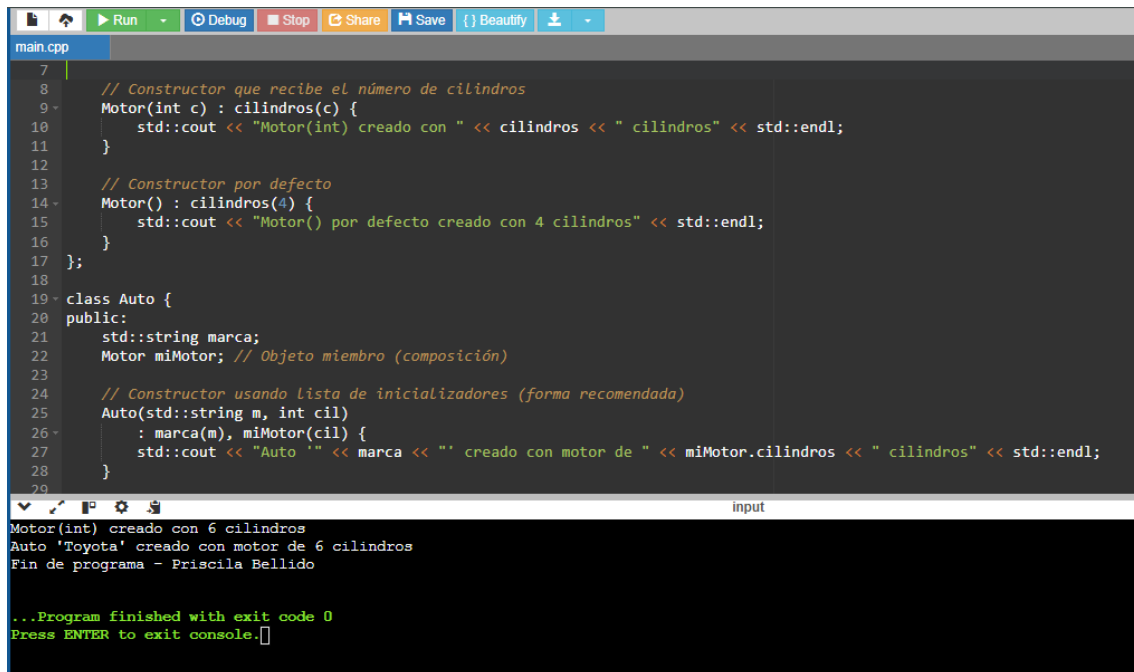
## ¿Qué es la reusabilidad?

La **reusabilidad** es la capacidad de utilizar componentes de software ya creados, como clases o funciones, en diferentes programas o partes de un mismo programa sin tener que modificarlos mucho. Esto ayuda a ahorrar tiempo, mejorar la calidad del código y facilitar su mantenimiento.

## ¿Qué son los constructores?

- Son métodos especiales con el mismo nombre de la clase, sin tipo de retorno, que se ejecutan automáticamente al crear un objeto.
- Su función principal es inicializar los atributos para que el objeto tenga un estado válido.
- El **constructor por defecto** no recibe argumentos y es generado automáticamente si no defines ninguno.
- Puedes tener **múltiples constructores** (sobrecarga) para inicializar objetos de diferentes maneras.
- La **lista de inicializadores** es la forma eficiente y recomendada para asignar valores a los atributos, especialmente a objetos, constantes o referencias.

# MOTOR



The screenshot shows a C++ IDE with a file named 'main.cpp'. The code defines two classes: 'Motor' and 'Auto'. The 'Motor' class has two constructors: one that takes an integer 'c' and prints 'Motor(int) creado con ' followed by 'c' and 'cilindros', and another default constructor that sets 'cilindros' to 4 and prints 'Motor() por defecto creado con 4 cilindros'. The 'Auto' class has a public member 'std::string marca', a private member 'Motor miMotor', and a constructor that takes 'std::string m' and 'int cil', sets 'marca' to 'm' and 'miMotor' to a 'Motor' object with 'cil' cylinders, and prints 'Auto ' followed by 'marca', ' creado con motor de ' followed by 'cil' and 'cilindros'. The output window shows the execution results: 'Motor(int) creado con 6 cilindros', 'Auto 'Toyota' creado con motor de 6 cilindros', and 'Fin de programa - Priscila Bellido'. The program finished with exit code 0.

```
7 |
8 | // Constructor que recibe el número de cilindros
9 | Motor(int c) : cilindros(c) {
10 |     std::cout << "Motor(int) creado con " << cilindros << " cilindros" << std::endl;
11 | }
12 |
13 | // Constructor por defecto
14 | Motor() : cilindros(4) {
15 |     std::cout << "Motor() por defecto creado con 4 cilindros" << std::endl;
16 | }
17 | };
18 |
19 | class Auto {
20 | public:
21 |     std::string marca;
22 |     Motor miMotor; // Objeto miembro (composición)
23 |
24 |     // Constructor usando lista de inicializadores (forma recomendada)
25 |     Auto(std::string m, int cil)
26 |     : marca(m), miMotor(cil) {
27 |         std::cout << "Auto '" << marca << "' creado con motor de " << miMotor.cilindros << " cilindros" << std::endl;
28 |     }
29 | }
```

Motor(int) creado con 6 cilindros  
Auto 'Toyota' creado con motor de 6 cilindros  
Fin de programa - Priscila Bellido

...Program finished with exit code 0  
Press ENTER to exit console.

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

class Motor {
public:
    int cilindros;

    // Constructor que recibe el número de cilindros
    Motor(int c) : cilindros(c) {
        std::cout << "Motor(int) creado con " << cilindros << " cilindros"
<< std::endl;
    }

    // Constructor por defecto
    Motor() : cilindros(4) {
        std::cout << "Motor() por defecto creado con 4 cilindros" <<
std::endl;
    }
};

class Auto {
public:
    std::string marca;
    Motor miMotor; // Objeto miembro (composición)

    // Constructor usando lista de inicializadores (forma recomendada)
    Auto(std::string m, int cil)
        : marca(m), miMotor(cil) {
        std::cout << "Auto '" << marca << "' creado con motor de " <<
miMotor.cilindros << " cilindros" << std::endl;
    }

    /*
    // Forma alternativa (menos eficiente)
    Auto(std::string m, int cil) {
        marca = m;
    }
    */
};
```

```

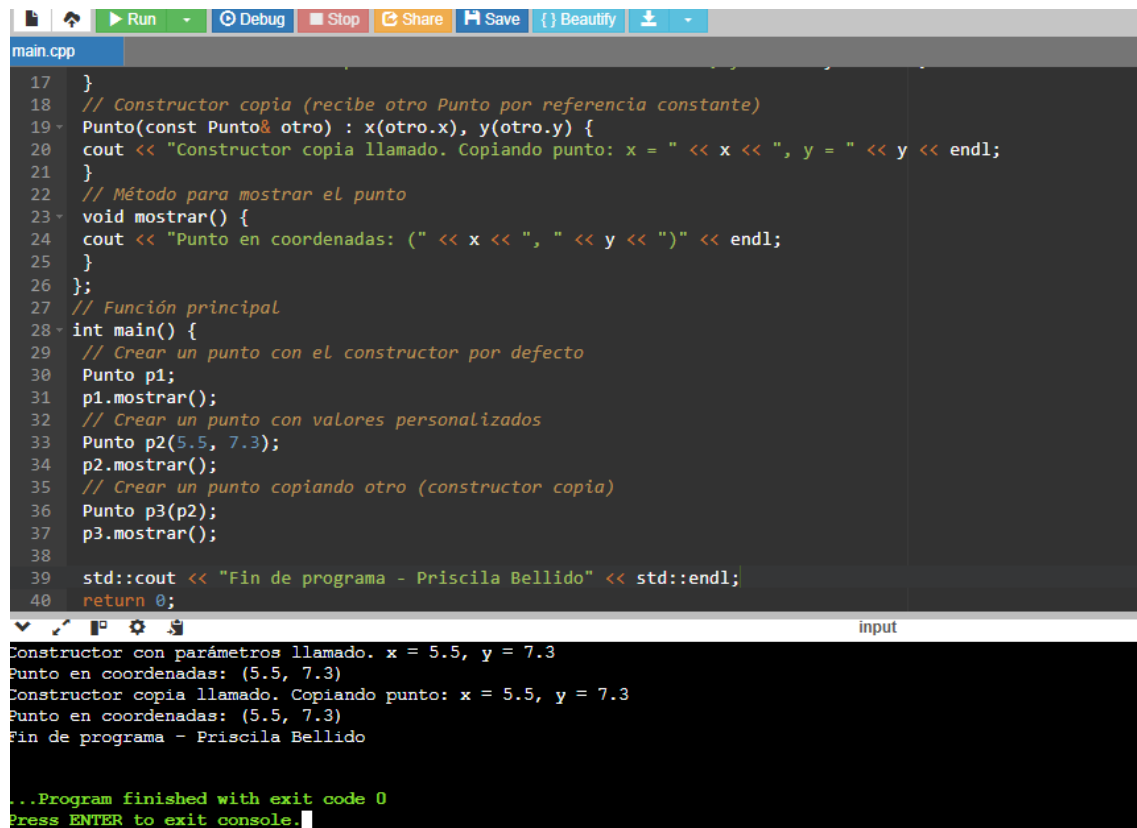
        // Aquí primero se crea miMotor usando el constructor por defecto
        (Motor())
        // Luego se reasigna con un nuevo Motor(cil), lo cual es menos
        eficiente:
        // - Se hacen dos operaciones: construcción y luego reasignación.
        // - Puede ser problemático si Motor no tiene constructor por
        defecto.
        // - No se puede usar así si el miembro es const o una referencia.
        miMotor = Motor(cil);
        std::cout << "Auto '" << marca << "' creado (forma menos eficiente)"
        << std::endl;
    }
    */
};

int main() {
    // Crear un Auto utilizando la forma recomendada (con lista de
    inicializadores)
    Auto miAuto("Toyota", 6);

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;
    return 0;
}

```

## El uso de constructores en una clase



```

main.cpp
17 }
18 // Constructor copia (recibe otro Punto por referencia constante)
19 Punto(const Punto& otro) : x(otro.x), y(otro.y) {
20     cout << "Constructor copia llamado. Copiando punto: x = " << x << ", y = " << y << endl;
21 }
22 // Método para mostrar el punto
23 void mostrar() {
24     cout << "Punto en coordenadas: (" << x << ", " << y << ")" << endl;
25 }
26 };
27 // Función principal
28 int main() {
29     // Crear un punto con el constructor por defecto
30     Punto p1;
31     p1.mostrar();
32     // Crear un punto con valores personalizados
33     Punto p2(5.5, 7.3);
34     p2.mostrar();
35     // Crear un punto copiando otro (constructor copia)
36     Punto p3(p2);
37     p3.mostrar();
38
39     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
40     return 0;
}

```

input

```

Constructor con parámetros llamado. x = 5.5, y = 7.3
Punto en coordenadas: (5.5, 7.3)
Constructor copia llamado. Copiando punto: x = 5.5, y = 7.3
Punto en coordenadas: (5.5, 7.3)
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.

```

```

#include <iostream>
#include <string>
using namespace std;
// Clase Punto para representar coordenadas en 2D
class Punto {
private:
    double x;
    double y;
public:
    // Constructor por defecto (sin argumentos)
    Punto() : x(0.0), y(0.0) {
        cout << "Constructor por defecto llamado. x = " << x << ", y = " << y <<
endl;
    }
    // Constructor con parámetros
    Punto(double xVal, double yVal) : x(xVal), y(yVal) {
        cout << "Constructor con parámetros llamado. x = " << x << ", y = " << y <<
endl;
    }
    // Constructor copia (recibe otro Punto por referencia constante)
    Punto(const Punto& otro) : x(otro.x), y(otro.y) {
        cout << "Constructor copia llamado. Copiando punto: x = " << x << ", y = "
<< y << endl;
    }
    // Método para mostrar el punto
    void mostrar() {
        cout << "Punto en coordenadas: (" << x << ", " << y << ")" << endl;
    }
};
// Función principal
int main() {
    // Crear un punto con el constructor por defecto
    Punto p1;
    p1.mostrar();
    // Crear un punto con valores personalizados
    Punto p2(5.5, 7.3);
    p2.mostrar();
    // Crear un punto copiando otro (constructor copia)
    Punto p3(p2);
    p3.mostrar();

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;
    return 0;
}

```

## ¿Qué es un destructor?

Un **destructor** es un método especial de una clase que se ejecuta automáticamente cuando un objeto deja de existir (por ejemplo, cuando termina su alcance o se elimina). Su función principal es **liberar recursos** que el objeto haya utilizado, como memoria dinámica o archivos abiertos, para evitar fugas de memoria o problemas en el programa.

En C++, el destructor tiene el mismo nombre que la clase, pero precedido por un símbolo ~, y no recibe parámetros ni devuelve nada.

## Ciclo de Vida en Acción: RecursoSimple

```
main.cpp
22  datosDinamicos = nullptr; // Buena práctica
23  std::cout << " RecursoSimple '" << nombreRecurso << "' libero su memoria dinamica." << std::endl;
24  }
25  void usarRecurso() const {
26  std::cout << "Usando RecursoSimple '" << nombreRecurso << "'. Datos[0]: "
27  << (datosDinamicos ? datosDinamicos[0] : -1) << std::endl;
28  }
29  }; // Fin de la clase RecursoSimple
30  void funcionDePrueba() {
31  std::cout << "\n-- Entrando a funcionDePrueba --" << std::endl;
32  RecursoSimple recursoLocal("LocalEnFuncion"); // Objeto en el Stack
33  recursoLocal.usarRecurso();
34  std::cout << "-- Saliendo de funcionDePrueba (recursoLocal se destruirá) --" << std::endl;
35  // El destructor de 'recursoLocal' se llama aquí automáticamente
36  }
37  int main() {
38  std::cout << "-- Inicio de main --" << std::endl;
39  RecursoSimple* recursoEnHeap = nullptr;
40  // Creando objeto en el Heap
41  recursoEnHeap = new RecursoSimple("DinamicoEnHeap");
42  if (recursoEnHeap) {
43  recursoEnHeap->usarRecurso();
44  }
45  funcionDePrueba(); // Llamamos a la función que crea un objeto local
46  // Liberando objeto del Heap
47  std::cout << "\n-- Antes de delete recursoEnHeap --" << std::endl;
48  delete recursoEnHeap; // ¡Se llama al destructor de recursoEnHeap!
49  recursoEnHeap = nullptr;
50  std::cout << "\n-- Fin de main --" << std::endl;
51
52  std::cout << "Fin de programa - Priscila Bellido" << std::endl;
53  return 0;
54 }
```

```
-- Inicio de main --
CONSTRUCTOR: Creando RecursoSimple 'DinamicoEnHeap'.
RecursoSimple 'DinamicoEnHeap' asigno memoria dinamica en 0x5a21c7fc26f0
Usando RecursoSimple 'DinamicoEnHeap'. Datos[0]: 0

-- Entrando a funcionDePrueba --
CONSTRUCTOR: Creando RecursoSimple 'LocalEnFuncion'.
RecursoSimple 'LocalEnFuncion' asigno memoria dinamica en 0x5a21c7fc2710
Usando RecursoSimple 'LocalEnFuncion'. Datos[0]: 0
-- Saliendo de funcionDePrueba (recursoLocal se destruirá) --
DESTRUCTOR: Destruyendo RecursoSimple 'LocalEnFuncion'.
RecursoSimple 'LocalEnFuncion' libero su memoria dinamica.

-- Antes de delete recursoEnHeap --
DESTRUCTOR: Destruyendo RecursoSimple 'DinamicoEnHeap'.
RecursoSimple 'DinamicoEnHeap' libero su memoria dinamica.

-- Fin de main --
Fin de programa - Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```
#include <iostream>
#include <string>
class RecursoSimple {
private:
    std::string nombreRecurso;
    int* datosDinamicos; // Un puntero para demostrar la gestión de memoria
    dinámica
public:
```



```

// Constructor
RecursoSimple(const std::string& nombre) : nombreRecurso(nombre) {
std::cout << "CONSTRUCTOR: Creando RecursoSimple '" << nombreRecurso <<
"'. " << std::endl;
// Simulamos que este recurso necesita un pequeño bloque de memoria
dinamica
datosDinamicos = new int[5]; // Pide memoria para 5 enteros
std::cout << " RecursoSimple '" << nombreRecurso << "' asigno memoria
dinamica en "
<< datosDinamicos << std::endl;
for (int i=0; i<5; ++i) datosDinamicos[i] = i*10; // Inicializa
}
// Destructor
~RecursoSimple() {
std::cout << "DESTRUCTOR: Destruyendo RecursoSimple '" << nombreRecurso <<
"'. " << std::endl;
// ¡MUY IMPORTANTE! Liberar la memoria dinámica asignada en el constructor
delete[] datosDinamicos;
datosDinamicos = nullptr; // Buena práctica
std::cout << " RecursoSimple '" << nombreRecurso << "' libero su memoria
dinamica." << std::endl;
}
void usarRecurso() const {
std::cout << "Usando RecursoSimple '" << nombreRecurso << "'. Datos[0]: "
<< (datosDinamicos ? datosDinamicos[0] : -1) << std::endl;
}
}; // Fin de la clase RecursoSimple
void funcionDePrueba() {
std::cout << "\n-- Entrando a funcionDePrueba --" << std::endl;
RecursoSimple recursoLocal("LocalEnFuncion"); // Objeto en el Stack
recursoLocal.usarRecurso();
std::cout << "-- Saliendo de funcionDePrueba (recursoLocal se destruió) --
" << std::endl;
// El destructor de 'recursoLocal' se llama aquí automáticamente
}
int main() {
std::cout << "-- Inicio de main --" << std::endl;
RecursoSimple* recursoEnHeap = nullptr;
// Creando objeto en el Heap
recursoEnHeap = new RecursoSimple("DinamicoEnHeap");
if (recursoEnHeap) {
recursoEnHeap->usarRecurso();
}
funcionDePrueba(); // Llamamos a la función que crea un objeto local
// Liberando objeto del Heap
std::cout << "\n-- Antes de delete recursoEnHeap --" << std::endl;
delete recursoEnHeap; // ¡Se llama al destructor de recursoEnHeap!
recursoEnHeap = nullptr;
std::cout << "\n-- Fin de main --" << std::endl;

std::cout << "Fin de programa - Priscila Bellido" << std::endl;
return 0;
}

```

# Clase 12: Objetos que Perduran y Se Construyen con Otros Objetos

Hoy aprendimos:

## La Persistencia de Objetos

- Los objetos en memoria desaparecen cuando el programa termina, por eso necesitamos guardar su estado para usarlo después.
- La **persistencia** es la capacidad de guardar el estado de un objeto en almacenamiento no volátil y recuperarlo luego.
- Es necesaria para guardar trabajo, configuraciones, bases de datos y compartir información entre ejecuciones o programas.
- Métodos comunes incluyen archivos de texto, serialización (binaria, XML, JSON) y bases de datos (relacionales, orientadas a objetos y NoSQL).

## ¿Qué es la composición?

- La **composición** representa una relación fuerte entre objetos: uno **“tiene”** o está compuesto por otros.
- Ejemplo: un automóvil **tiene un motor** y **tiene ruedas**.
- Las **partes** (como el motor) son atributos del objeto principal (el automóvil).
- Las partes se crean y destruyen junto con el objeto completo: **no existen de forma independiente**.
- Esta relación se representa en UML con un **rombo negro** en el lado del objeto que contiene.

## Composición ("tiene una parte propia")

- Relación **fuerte** "parte-de".
- El objeto contenido **no puede existir sin** el objeto contenedor.
- El contenedor **crea y destruye** sus partes.
- Ejemplo:

Un **Auto** tiene un **Motor** → si el auto se destruye, el motor también.

- **UML:** Rombo negro (●) en el lado del contenedor.

## Agregación ("tiene una parte compartida")

- Relación **débil** "tiene un".
- El objeto contenido **puede existir por separado** del contenedor.
- El contenedor **no es responsable** de crear ni destruir el contenido.
- Ejemplo:

Un **Curso** tiene **Estudiantes**, pero los estudiantes existen aunque el curso se elimine.

- **UML:** Rombo blanco (○) en el lado del agregador.

## ¡Ensamblando Nuestro Automóvil con su Motor!

```
main.cpp
44 }
45 void encender() {
46     std::cout << modelo << ": Intentando encender..." << std::endl;
47     motorInterno.arrancar(); // Delega la acción al objeto Motor
48 }
49 void apagar() {
50     std::cout << modelo << ": Intentando apagar..." << std::endl;
51     motorInterno.detener();
52 }
53 void verDiagnostico() const {
54     std::cout << "Diagnostico del " << modelo << ":" << std::endl;
55     motorInterno.mostrarEstado();
56 }
57 };
58 int main() {
59     std::cout << "--- Creando un Automovil en el Stack ---" << std::endl;
60     Automovil miAuto("SuperMarca", "ModeloX", 200); // Llama al constructor de Automovil
61     // que llama al constructor de Motor
62     miAuto.verDiagnostico();
63     miAuto.encender();
64     miAuto.verDiagnostico();
65     miAuto.apagar();
66     std::cout << "\n--- Saliendo de main (miAuto se destruirá) ---" << std::endl;
67     // Al salir de main, se llama al destructor de miAuto,
68     // el cual IMPLÍCITAMENTE llama al destructor de motorInterno.
69
70     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
71     return 0;
72 }
```

```
--- Creando un Automovil en el Stack ---
CONSTRUCTOR Motor: Creado motor de 200 HP.
CONSTRUCTOR Automovil: Ensamblado un SuperMarca ModeloX con un motor.
Diagnostico del ModeloX:
Estado del Motor: Apagado, HP: 200
ModeloX: Intentando encender...
Motor: ¡BRUM! Encendido.
Diagnostico del ModeloX:
Estado del Motor: Encendido, HP: 200
ModeloX: Intentando apagar...
Motor: ...silencio. Apagado.

--- Saliendo de main (miAuto se destruirá) ---
Fin de programa - Priscila Bellido
DESTRUCTOR Automovil: Desguazando el SuperMarca ModeloX.
DESTRUCTOR Motor: Destruído motor de 200 HP.

...Program finished with exit code 0
Press ENTER to exit console.
```

```

#include <iostream>
#include <string>
// Clase 'Parte': Motor
class Motor {
private:
    int caballosDeFuerza;
    bool encendido;
public:
    Motor(int hp = 150) : caballosDeFuerza(hp), encendido(false) {
        std::cout << " CONSTRUCTOR Motor: Creado motor de " << caballosDeFuerza <<
" HP." << std::endl;
    }
    ~Motor() {
        std::cout << " DESTRUCTOR Motor: Destruido motor de " << caballosDeFuerza
<< " HP." << std::endl;
    }
    void arrancar() {
        if (!encendido) { encendido = true; std::cout << " Motor: ¡BRUM!
Encendido." << std::endl;}
        else { std::cout << " Motor: Ya estaba encendido." << std::endl; }
    }
    void detener() {
        if (encendido) { encendido = false; std::cout << " Motor: ...silencio.
Apagado." << std::endl;}
        else { std::cout << " Motor: Ya estaba apagado." << std::endl; }
    }
    void mostrarEstado() const {
        std::cout << " Estado del Motor: " << (encendido ? "Encendido" : "Apagado")
<< ", HP: " << caballosDeFuerza << std::endl;
    }
};
// Clase 'Todo' o 'Contenedora': Automovil
class Automovil {
private:
    std::string marca;
    std::string modelo;
    Motor motorInterno; // ¡COMPOSICIÓN! Un objeto Motor es MIEMBRO de
Automovil.
public:
    // Constructor de Automovil: INICIALIZA motorInterno usando la lista de
inicializadores
    Automovil(std::string ma, std::string mo, int hpDelMotor)
        : marca(ma), modelo(mo), motorInterno(hpDelMotor) {
        std::cout << "CONSTRUCTOR Automovil: Ensamblado un " << marca << " " <<
modelo
<< " con un motor." << std::endl;
    }
    ~Automovil() {
        std::cout << "DESTRUCTOR Automovil: Desguazando el " << marca << " " <<
modelo << "." << std::endl;
        // El destructor de motorInterno se llamará AUTOMÁTICAMENTE aquí.
    }
    void encender() {
        std::cout << modelo << ": Intentando encender..." << std::endl;
        motorInterno.arrancar(); // Delega la acción al objeto Motor
    }
    void apagar() {
        std::cout << modelo << ": Intentando apagar..." << std::endl;
        motorInterno.detener();
    }
    void verDiagnostico() const {
        std::cout << "Diagnostico del " << modelo << ":" << std::endl;
        motorInterno.mostrarEstado();
    }
};

```

```

    }
};
int main() {
    std::cout << "--- Creando un Automovil en el Stack ---" << std::endl;
    Automovil miAuto("SuperMarca", "ModeloX", 200); // Llama al constructor de
Automovil
    // que llama al constructor de Motor
    miAuto.verDiagnostico();
    miAuto.encender();
    miAuto.verDiagnostico();
    miAuto.apagar();
    std::cout << "\n--- Saliendo de main (miAuto se destrui) ---" <<
std::endl;
    // Al salir de main, se llama al destructor de miAuto,
    // el cual IMPLÍCITAMENTE llama al destructor de motorInterno.

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;
    return 0;
}

```

## Composición en Automóvil: Entendiendo el Ensamblaje

- En la composición, un **objeto miembro** como `Motor motorInterno` forma parte de la clase `Automovil`.
- Se **inicializa en la lista de inicialización** del constructor de `Automovil`, asegurando su creación correcta.
- Cuando un `Automovil` se destruye, su `Motor` también se destruye **automáticamente**, sin necesidad de usar `delete`, ya que no es un puntero dinámico.

# Clase 13: Lazos de Familia en POO - ¡Introducción a la Herencia!

Hoy aprendimos:

## ¿Qué es la herencia?

La **herencia** en programación orientada a objetos (POO) es un mecanismo que permite crear una nueva clase **basada en otra existente**.

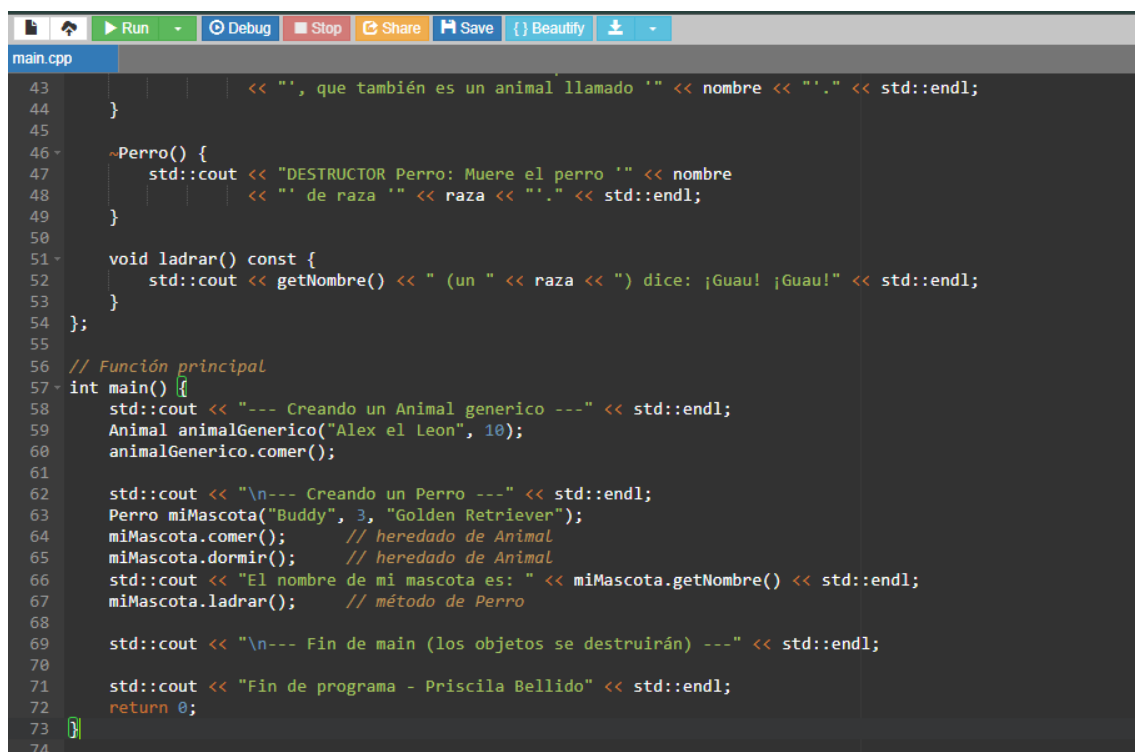
La nueva clase, llamada **clase derivada** o **subclase**, **hereda** los atributos y métodos de la clase existente, llamada **clase base** o **superclase**. Esto permite **reutilizar código** y establecer una relación tipo "es un tipo de".

## La herencia y su terminología

### Herencia: Términos y Código

- **Clase base (padre):** de donde se hereda.
- **Clase derivada (hija):** la que hereda atributos y métodos.
- La **herencia pública** es la más común y significa que la clase hija “es un tipo de” la clase padre.
  - Los miembros `public` de la base → siguen siendo `public` en la derivada.
  - Los miembros `protected` → siguen siendo `protected`.
  - Los miembros `private` → no son accesibles directamente en la derivada.
- Existen también **herencia `protected`** y **`private`**, pero se usan menos y cambian la visibilidad de los miembros heredados.

## De Animal a Perro: Herencia en Código



```
main.cpp
43         << "", que también es un animal llamado "" << nombre << "." << std::endl;
44     }
45
46     ~Perro() {
47         std::cout << "DESTRUCTOR Perro: Muere el perro "" << nombre
48         << "" de raza "" << raza << "." << std::endl;
49     }
50
51     void ladrar() const {
52         std::cout << getNombre() << " (un " << raza << ") dice: ¡Guau! ¡Guau!" << std::endl;
53     }
54 };
55
56 // Función principal
57 int main() {
58     std::cout << "--- Creando un Animal generico ---" << std::endl;
59     Animal animalGenerico("Alex el Leon", 10);
60     animalGenerico.comer();
61
62     std::cout << "\n--- Creando un Perro ---" << std::endl;
63     Perro miMascota("Buddy", 3, "Golden Retriever");
64     miMascota.comer(); // heredado de Animal
65     miMascota.dormir(); // heredado de Animal
66     std::cout << "El nombre de mi mascota es: " << miMascota.getNombre() << std::endl;
67     miMascota.ladrar(); // método de Perro
68
69     std::cout << "\n--- Fin de main (los objetos se destruirán) ---" << std::endl;
70
71     std::cout << "Fin de programa - Priscila Bellido" << std::endl;
72     return 0;
73 }
```

```
input
--- Creando un Animal generico ---
CONSTRUCTOR Animal: Nace un animal llamado 'Alex el Leon' de 10 anio(s).
Alex el Leon esta comiendo.

--- Creando un Perro ---
CONSTRUCTOR Animal: Nace un animal llamado 'Buddy' de 3 anio(s).
CONSTRUCTOR Perro: Nace un perro de raza 'Golden Retriever', que también es un animal llamado 'Buddy'.
Buddy esta comiendo.
Buddy esta durmiendo.
El nombre de mi mascota es: Buddy
Buddy (un Golden Retriever) dice: ¡Guau! ¡Guau!

--- Fin de main (los objetos se destruirán) ---
Fin de programa - Priscila Bellido
DESTRUCTOR Perro: Muere el perro 'Buddy' de raza 'Golden Retriever'.
DESTRUCTOR Animal: Muere el animal 'Buddy'.
DESTRUCTOR Animal: Muere el animal 'Alex el Leon'.

...Program finished with exit code 0
Press ENTER to exit console.
```

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

// Clase Base
class Animal {
protected: // Para que las clases derivadas puedan acceder
    std::string nombre;
    int edad;

public:
    Animal(const std::string& n, int e) : nombre(n), edad(e) {
        std::cout << "CONSTRUCTOR Animal: Nace un animal llamado '"
            << nombre << " de " << edad << " anio(s)." << std::endl;
    }

    ~Animal() {
        std::cout << "DESTRUCTOR Animal: Muere el animal '" << nombre <<
            "'." << std::endl;
    }

    void comer() const {
        std::cout << nombre << " esta comiendo." << std::endl;
    }

    void dormir() const {
        std::cout << nombre << " esta durmiendo." << std::endl;
    }

    std::string getNombre() const {
        return nombre;
    }
};

// Clase Derivada
class Perro : public Animal {
private:
    std::string raza;

public:
    // Constructor de Perro llama al constructor de Animal
    Perro(const std::string& n, int e, const std::string& r)
        : Animal(n, e), raza(r) {
        std::cout << "CONSTRUCTOR Perro: Nace un perro de raza '" << raza
            << "', que también es un animal llamado '" << nombre <<
            "'." << std::endl;
    }
};
```

```

    }

    ~Perro() {
        std::cout << "DESTRUCTOR Perro: Muere el perro '" << nombre
                  << "' de raza '" << raza << "'." << std::endl;
    }

    void ladrar() const {
        std::cout << getNombre() << " (un " << raza << ") dice: ¡Guau!
        ¡Guau!" << std::endl;
    }
};

// Función principal
int main() {
    std::cout << "--- Creando un Animal generico ---" << std::endl;
    Animal animalGenerico("Alex el Leon", 10);
    animalGenerico.comer();

    std::cout << "\n--- Creando un Perro ---" << std::endl;
    Perro miMascota("Buddy", 3, "Golden Retriever");
    miMascota.comer();          // heredado de Animal
    miMascota.dormir();         // heredado de Animal
    std::cout << "El nombre de mi mascota es: " << miMascota.getNombre() <<
    std::endl;
    miMascota.ladrar();         // método de Perro

    std::cout << "\n--- Fin de main (los objetos se destruirán) ---" <<
    std::endl;

    std::cout << "Fin de programa - Priscila Bellido" << std::endl;
    return 0;
}

```

## Ramificaciones de la Herencia

- **Herencia múltiple:** una clase puede heredar de varias clases en C++, aunque puede causar conflictos (como el *problema del diamante*).
- **Herencia protegida y privada:** cambian cómo se heredan los miembros respecto al exterior.
- **Sobrescritura de métodos:** una clase derivada puede redefinir un método heredado; clave para el **polimorfismo**.
- **Funciones virtuales y clases abstractas:** conceptos avanzados que permiten un comportamiento flexible y general mediante polimorfismo.
- **Problema del corte (slicing):** ocurre al asignar un objeto derivado a una variable base por valor, perdiendo la información extra de la clase hija.



## Clase 14: Refinando la Herencia - protected y ¡Personalizando Comportamientos!

Hoy aprendimos:

### ¿Qué es el acceso protected en herencia?

Protected es un nivel de acceso intermedio entre public y private.

Los miembros protected:

- **Sí** son accesibles desde la **clase base**.
- **Sí** son accesibles desde las **clases derivadas** (incluso "nietas").
- **NO** son accesibles desde **fuera de la jerarquía**, como desde `main()`.

### ¿Cuándo usar protected?

- Cuando **no quieres exponer** un atributo al público (`main`), pero **sí** necesitas que las **clases hijas lo usen** directamente.
- Es un **compromiso útil**: oculta los detalles al exterior, pero mantiene accesibilidad dentro de la familia de clases.

### ¿Qué es Sobrescribir un Método?

Sobrescribir un método (en inglés *method overriding*) es cuando una clase derivada **redefine un método heredado de la clase base para darle un comportamiento específico**.

## Especializando el Arte: Figura y Circulo

```
main.cpp
52 }
53
54 void dibujar() const override {
55     std::cout << "Rectangulo '" << nombreFigura << "': Dibujando un rectangulo de color " << color
56         << " con base " << base << " y altura " << altura << "." << std::endl;
57     std::cout << " Area: " << (base * altura) << std::endl;
58 }
59
60 ~Rectangulo() override {
61     std::cout << " DESTRUCTOR Rectangulo: Base " << base << ", Altura " << altura << std::endl;
62 }
63 };
64
65 int main() {
66     std::cout << "--- Creando y dibujando una Figura generica ---" << std::endl;
67     Figura fig("Azul", "Figura Misteriosa");
68     fig.dibujar();
69
70     std::cout << "\n--- Creando y dibujando un Circulo ---" << std::endl;
71     Circulo circ("Rojo", 5.0);
72     circ.dibujar();
73
74     std::cout << "\n--- Creando y dibujando un Rectangulo ---" << std::endl;
75     Rectangulo rect("Verde", 4.0, 6.0);
76     rect.dibujar();
77
78     std::cout << "\n--- Fin de main ---" << std::endl;
79
80     std::cout << "Creado por: Priscila Bellido" << std::endl;
81     return 0;
82 }
```

```

--- Creando y dibujando una Figura generica ---
CONSTRUCTOR Figura: 'Figura Misteriosa' de color Azul
Figura 'Figura Misteriosa': Dibujando una figura geometrica generica de color Azul.

--- Creando y dibujando un Circulo ---
CONSTRUCTOR Figura: 'Circulo' de color Rojo
CONSTRUCTOR Circulo: Radio 5
Circulo 'Circulo': Dibujando un circulo perfecto de color Rojo y radio 5.
Area: 78.5397

--- Creando y dibujando un Rectangulo ---
CONSTRUCTOR Figura: 'Rectangulo' de color Verde
CONSTRUCTOR Rectangulo: Base 4, Altura 6
Rectangulo 'Rectangulo': Dibujando un rectangulo de color Verde con base 4 y altura 6.
Area: 24

--- Fin de main ---
Creado por: Priscila Bellido
DESTRUCTOR Rectangulo: Base 4, Altura 6
DESTRUCTOR Figura: 'Rectangulo'
DESTRUCTOR Circulo: Radio 5
DESTRUCTOR Figura: 'Circulo'
DESTRUCTOR Figura: 'Figura Misteriosa'

...Program finished with exit code 0
Press ENTER to exit console.

```

```

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

class Figura {
protected:
    std::string color;
    std::string nombreFigura;

public:
    Figura(std::string c, std::string nf) : color(c), nombreFigura(nf) {
        std::cout << " CONSTRUCTOR Figura: " << nombreFigura << " de color " << color << std::endl;
    }

    virtual void dibujar() const {
        std::cout << "Figura " << nombreFigura << ": Dibujando una figura geometrica generica de color " << color << "." << std::endl;
    }

    virtual ~Figura() {
        std::cout << " DESTRUCTOR Figura: " << nombreFigura << "" << std::endl;
    }
};

class Circulo : public Figura {
private:
    double radio;

public:
    Circulo(std::string c, double r) : Figura(c, "Circulo"), radio(r) {
        std::cout << " CONSTRUCTOR Circulo: Radio " << radio << std::endl;
    }
}

```

```

        void dibujar() const override {
            std::cout << "Circulo '" << nombreFigura << "': Dibujando un circulo
perfecto de color " << color
                << " y radio " << radio << "." << std::endl;
            std::cout << " Area: " << (PI * radio * radio) << std::endl;
        }

        ~Circulo() override {
            std::cout << " DESTRUCTOR Circulo: Radio " << radio << std::endl;
        }
    };

    class Rectangulo : public Figura {
    private:
        double base, altura;

    public:
        Rectangulo(std::string c, double b, double h) : Figura(c, "Rectangulo"),
        base(b), altura(h) {
            std::cout << " CONSTRUCTOR Rectangulo: Base " << base << ", Altura "
<< altura << std::endl;
        }

        void dibujar() const override {
            std::cout << "Rectangulo '" << nombreFigura << "': Dibujando un
rectangulo de color " << color
                << " con base " << base << " y altura " << altura << "."
<< std::endl;
            std::cout << " Area: " << (base * altura) << std::endl;
        }

        ~Rectangulo() override {
            std::cout << " DESTRUCTOR Rectangulo: Base " << base << ", Altura "
<< altura << std::endl;
        }
    };

    int main() {
        std::cout << "--- Creando y dibujando una Figura generica ---" <<
std::endl;
        Figura fig("Azul", "Figura Misteriosa");
        fig.dibujar();

        std::cout << "\n--- Creando y dibujando un Circulo ---" << std::endl;
        Circulo circ("Rojo", 5.0);
        circ.dibujar();

        std::cout << "\n--- Creando y dibujando un Rectangulo ---" << std::endl;
        Rectangulo rect("Verde", 4.0, 6.0);
        rect.dibujar();

        std::cout << "\n--- Fin de main ---" << std::endl;

        std::cout << "Creado por: Priscila Bellido" << std::endl;
        return 0;
    }

```

## La Esencia de la Especialización en POO

- **Sobrescritura (override):**  
Las clases derivadas (como Circulo y Rectangulo) redefinen métodos

existentes en la clase base (`Figura`), proporcionando su propia implementación para `dibujar()`.

La firma del método debe coincidir exactamente y es buena práctica usar la palabra clave `override` para que el compilador verifique.

- **Acceso a miembros de la clase base:**

Desde el método sobrescrito, la clase derivada puede acceder a los miembros `public` y `protected` de la clase base, como atributos `color` y `nombreFigura`.

- **Llamada a la versión base (opcional):**

La clase derivada puede llamar explícitamente al método original de la clase base para reutilizar su lógica y luego añadir código específico.

Ejemplo: `Figura::dibujar()`; dentro de `Circulo::dibujar()`.

## La herencia y el comportamiento polimórfico

- Funciones **virtual**:

Permiten que al usar un puntero o referencia a la clase base, se ejecute automáticamente el método sobrescrito en la clase derivada. Esto habilita el **polimorfismo dinámico**.

- Palabra clave **final**:

- Si se aplica a un **método**, evita que sea sobrescrito en clases derivadas.
- Si se aplica a una **clase**, impide que se herede de ella.

- Tipos de retorno covariantes:

Permiten que un método sobrescrito retorne un tipo más específico (una clase derivada del tipo retornado por el método base). Es útil para mantener compatibilidad con la jerarquía de clases.

## ¿Por qué se usa *Sobrescritura (Override)*?

La **sobrescritura** se usa cuando quieres **personalizar o cambiar el comportamiento de un método heredado** de una clase base en una clase derivada.

## ¿Por qué se usa *Sobrecarga (Overload)*?

La **sobrecarga** se usa cuando quieres que un método tenga **varias versiones con diferentes argumentos**, pero con el mismo nombre.

# Clase 15: Clase 15: ¡Polimorfismo al Descubierto! El Verdadero Significado de "Muchas Formas"

Hoy aprendimos:

## ¿Qué es el Polimorfismo?

La palabra "polimorfismo" viene del griego y significa "**muchas formas**". En POO, significa que **un mismo método puede comportarse de manera diferente dependiendo del objeto que lo use.**

- Permite escribir código **más flexible, reutilizable y extensible.**
- Te da la capacidad de tratar **objetos de diferentes clases derivadas como si fueran de la clase base**, y que aun así, ejecuten su propio comportamiento.

## 1. Polimorfismo en Tiempo de Compilación (*Estático*)

- ✓ Sobrecarga de funciones
- ✓ Plantillas (templates)
- ✓ El compilador decide qué versión usar.

## ¿Qué es virtual?

Es una palabra clave que se usa en métodos de una clase base para permitir que las clases derivadas puedan sobrescribir ese método y que, al llamarlo, se ejecute la versión correcta dependiendo del tipo real del objeto (incluso si se usa un puntero a la clase base)

## ¿Cómo funciona?

- Se basa en un mecanismo llamado vtable (tabla de funciones virtuales).
- Cada objeto tiene un puntero oculto (vptr) que apunta a la tabla de métodos de su clase.
- Al llamar un método virtual a través de un puntero o referencia a la clase base:
  1. Se consulta la vtable en tiempo de ejecución.
  2. Se ejecuta la implementación específica de la clase real (derivada).
- Esto se conoce como enlace dinámico.

## ¡La Orquesta Polimórfica de Figuras!

```
main.cpp
74 std::cout << "Procesando figura (vía puntero a Figura):" << std::endl;
75 figPtr->dibujar();
76 std::cout << " Su área es: " << figPtr->calcularArea() << std::endl;
77 }
78
79 int main() {
80     Figura* ptrFig1 = new Circulo("Círculo Mágico", 10.0);
81     Figura* ptrFig2 = new Rectangulo("Rectángulo Dorado", 5.0, 8.0);
82
83     std::cout << "--- Llamadas directas a través de punteros a Figura ---" << std::endl;
84     ptrFig1->dibujar();
85     ptrFig2->dibujar();
86
87     std::cout << "\n--- Usando la función genérica procesarFigura ---" << std::endl;
88     procesarFigura(ptrFig1);
89     procesarFigura(ptrFig2);
90
91     std::vector<Figura*> figurasParaMostrar = { ptrFig1, ptrFig2 };
92
93     std::cout << "\n--- Dibujando todas las figuras usando un vector ---" << std::endl;
94     for (const Figura* fig : figurasParaMostrar) {
95         fig->dibujar();
96     }
97
98     std::cout << "\n--- Liberando memoria (IMPORTANTE: Destructores Virtuales) ---" << std::endl;
99     delete ptrFig1;
100    delete ptrFig2;
101
102    std::cout << "Creado por: Priscila Bellido" << std::endl;
103    return 0;
104 }
```

```
--- Llamadas directas a través de punteros a Figura ---
Círculo Mágico: Dibujando un CIRCULO de radio 10.
Rectángulo Dorado: Dibujando un RECTANGULO de base 5 y altura 8.

--- Usando la función genérica procesarFigura ---
Procesando figura (vía puntero a Figura):
Círculo Mágico: Dibujando un CIRCULO de radio 10.
Su área es: 314.159
Procesando figura (vía puntero a Figura):
Rectángulo Dorado: Dibujando un RECTANGULO de base 5 y altura 8.
Su área es: 40

--- Dibujando todas las figuras usando un vector ---
Círculo Mágico: Dibujando un CIRCULO de radio 10.
Rectángulo Dorado: Dibujando un RECTANGULO de base 5 y altura 8.

--- Liberando memoria (IMPORTANTE: Destructores Virtuales) ---
DESTRUCTOR Circulo: Círculo Mágico
DESTRUCTOR Figura: Círculo Mágico
DESTRUCTOR Rectangulo: Rectángulo Dorado
DESTRUCTOR Figura: Rectángulo Dorado
Creado por: Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

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

```
#define PI 3.14159
```

```

// Clase base abstracta que representa cualquier figura geométrica
class Figura {
protected:
    std::string nombreFigura;

public:
    Figura(std::string nf) : nombreFigura(nf) {}

    virtual void dibujar() const {
        std::cout << nombreFigura << ": Dibujando figura genérica." <<
std::endl;
    }

    virtual double calcularArea() const {
        std::cout << nombreFigura << ": Área de figura genérica no
definida." << std::endl;
        return 0.0;
    }

    virtual ~Figura() {
        std::cout << "DESTRUCTOR Figura: " << nombreFigura << std::endl;
    }
};

// Clase derivada Circulo
class Circulo : public Figura {
private:
    double radio;

public:
    Circulo(std::string nf, double r) : Figura(nf), radio(r) {}

    void dibujar() const override {
        std::cout << nombreFigura << ": Dibujando un CIRCULO de radio " <<
radio << "." << std::endl;
    }

    double calcularArea() const override {
        return PI * radio * radio;
    }

    ~Circulo() override {
        std::cout << "DESTRUCTOR Circulo: " << nombreFigura << std::endl;
    }
};

// Clase derivada Rectangulo
class Rectangulo : public Figura {
private:
    double base, altura;

public:
    Rectangulo(std::string nf, double b, double h) : Figura(nf), base(b),
altura(h) {}

    void dibujar() const override {
        std::cout << nombreFigura << ": Dibujando un RECTANGULO de base " <<
base
        << " y altura " << altura << "." << std::endl;
    }

    double calcularArea() const override {
        return base * altura;
    }
};

```

```

    }

    ~Rectangulo() override {
        std::cout << "DESTRUCTOR Rectangulo: " << nombreFigura << std::endl;
    }
};

// Función genérica que procesa cualquier figura
void procesarFigura(const Figura* figPtr) {
    std::cout << "Procesando figura (vía puntero a Figura):" << std::endl;
    figPtr->dibujar();
    std::cout << " Su área es: " << figPtr->calcularArea() << std::endl;
}

int main() {
    Figura* ptrFig1 = new Circulo("Círculo Mágico", 10.0);
    Figura* ptrFig2 = new Rectangulo("Rectángulo Dorado", 5.0, 8.0);

    std::cout << "--- Llamadas directas a través de punteros a Figura ---"
    << std::endl;
    ptrFig1->dibujar();
    ptrFig2->dibujar();

    std::cout << "\n--- Usando la función genérica procesarFigura ---" <<
    std::endl;
    procesarFigura(ptrFig1);
    procesarFigura(ptrFig2);

    std::vector<Figura*> figurasParaMostrar = { ptrFig1, ptrFig2 };

    std::cout << "\n--- Dibujando todas las figuras usando un vector ---" <<
    std::endl;
    for (const Figura* fig : figurasParaMostrar) {
        fig->dibujar();
    }

    std::cout << "\n--- Liberando memoria (IMPORTANTE: Destructores
    Virtuales) ---" << std::endl;
    delete ptrFig1;
    delete ptrFig2;

    std::cout << "Creado por: Priscila Bellido" << std::endl;
    return 0;
}

```



# Clase 16: ¡Contratos de Código! Clases Abstractas y Funciones Virtuales Puras

**Hoy aprendimos:**

## ¿Qué es el Polimorfismo?

Las clases abstractas en C++ son clases que no pueden ser instanciadas directamente y están pensadas para ser utilizadas como base para otras clases. Se utilizan cuando se quiere definir una interfaz común para un grupo de clases derivadas, pero dejando que esas clases hijas implementen detalles específicos.

## ¿Para qué sirven?

- Establecer contratos: obligan a las subclases a implementar ciertos métodos.
- Facilitan el uso del polimorfismo.
- Permiten diseñar sistemas más flexibles y extensibles.

## La Firma del Contrato Obligatorio y Funciones Virtuales Puras

- Una función virtual pura es una función virtual en una clase base que no tiene implementación en esa clase, y se declara con `= 0`.

## El "Contrato"

- Al declarar una función virtual pura, la clase base obliga a que cualquier clase derivada concreta (instanciable) la implemente.
- Este compromiso se considera un contrato de implementación.

## Consecuencias del contrato

1. La clase con al menos una función virtual pura se vuelve abstracta (no se puede instanciar).
2. Las clases derivadas que no implementen todas las funciones virtuales puras también serán abstractas.
3. Solo las clases derivadas que implementen todas las funciones virtuales puras heredadas pueden ser concretas (instanciables).

## ¿Puede tener implementación una función virtual pura?

Sí, aunque raro, se puede dar una implementación fuera de la clase. Sin embargo, las clases derivadas aún deben sobrescribirla. Esta implementación puede ser invocada explícitamente por las derivadas.

## Definiendo el "Contrato" de una Forma Geométrica

```
57     }
58 };
59 // Función para describir cualquier forma
60 void describirForma(const FormaGeometrica* forma) {
61     std::cout << "\n--- Descripción de Forma ---" << std::endl;
62     forma->dibujar();
63     std::cout << "    Área: " << forma->calcularArea() << std::endl;
64     std::cout << "    Color: " << forma->getColor() << std::endl;
65 }
66 int main() {
67     FormaGeometrica* ptrCirculo = new Circulo("Mi Circulo", "Rojo", 7.0);
68     FormaGeometrica* ptrRectangulo = new Rectangulo("Mi Rectangulo", "Azul", 4.0, 5.0);
69     describirForma(ptrCirculo);
70     describirForma(ptrRectangulo);
71     std::vector<FormaGeometrica*> misFormas;
72     misFormas.push_back(ptrCirculo);
73     misFormas.push_back(ptrRectangulo);
74     misFormas.push_back(new Circulo("Otro Circulo", "Verde", 3.0));
75     std::cout << "\n--- Procesando todas las formas del vector ---" << std::endl;
76     for (const FormaGeometrica* f : misFormas) {
77         f->dibujar();
78     }
79     std::cout << "\n--- Liberando memoria ---" << std::endl;
80     delete misFormas[2]; // Borra "Otro Circulo"
81     delete ptrRectangulo;
82     delete ptrCirculo;
83
84     std::cout << "Fin del programa Priscila Bellido" << std::endl;
85     return 0;
86 }
87
88
```

```
CONSTRUCTOR FormaGeometrica: Mi Circulo (Rojo)
CONSTRUCTOR FormaGeometrica: Mi Rectangulo (Azul)

--- Descripción de Forma ---
Mi Circulo (Rojo): Dibujando CIRCULO de radio 7.
    Área: 153.938
    Color: Rojo

--- Descripción de Forma ---
Mi Rectangulo (Azul): Dibujando RECTANGULO base 4, altura 5.
    Área: 20
    Color: Azul
CONSTRUCTOR FormaGeometrica: Otro Circulo (Verde)

--- Procesando todas las formas del vector ---
Mi Circulo (Rojo): Dibujando CIRCULO de radio 7.
Mi Rectangulo (Azul): Dibujando RECTANGULO base 4, altura 5.
Otro Circulo (Verde): Dibujando CIRCULO de radio 3.

--- Liberando memoria ---
DESTRUCTOR Circulo: Otro Circulo
DESTRUCTOR FormaGeometrica: Otro Circulo
DESTRUCTOR Rectangulo: Mi Rectangulo
DESTRUCTOR FormaGeometrica: Mi Rectangulo
DESTRUCTOR Circulo: Mi Circulo
DESTRUCTOR FormaGeometrica: Mi Circulo
Fin del programa Priscila Bellido

...Program finished with exit code 0
Press ENTER to exit console.
```

```

#include <iostream>
#include <string>
#include <vector>
#include <cmath>
// Clase base abstracta
class FormaGeometrica {
protected:
    std::string nombreFigura;
    std::string color;
public:
    FormaGeometrica(std::string nf, std::string c) : nombreFigura(nf),
    color(c) {
        std::cout << "CONSTRUCTOR FormaGeometrica: " << nombreFigura << " ("
<< color << ")" << std::endl;
    }
    virtual void dibujar() const = 0; // Método puro
    virtual double calcularArea() const = 0; // Método puro
    std::string getColor() const { return color; }
    std::string getNombre() const { return nombreFigura; }
    virtual ~FormaGeometrica() {
        std::cout << "DESTRUCTOR FormaGeometrica: " << nombreFigura <<
std::endl;
    }
};
// Clase derivada: Circulo
class Circulo : public FormaGeometrica {
private:
    double radio;
public:
    Circulo(std::string nf, std::string c, double r) : FormaGeometrica(nf,
c), radio(r) {}

    void dibujar() const override {
        std::cout << getNombre() << " (" << getColor() << "): Dibujando
CIRCULO de radio "
            << radio << "." << std::endl;
    }
    double calcularArea() const override {
        return 3.14159 * radio * radio;
    }

    ~Circulo() override {
        std::cout << " DESTRUCTOR Circulo: " << getNombre() << std::endl;
    }
};
// Clase derivada: Rectangulo
class Rectangulo : public FormaGeometrica {
private:
    double base, altura;
public:
    Rectangulo(std::string nf, std::string c, double b, double h) :
FormaGeometrica(nf, c), base(b), altura(h) {}

    void dibujar() const override {
        std::cout << getNombre() << " (" << getColor() << "): Dibujando
RECTANGULO base "
            << base << ", altura " << altura << "." << std::endl;
    }
    double calcularArea() const override {
        return base * altura;
    }
    ~Rectangulo() override {

```

```

        std::cout << "  DESTRUCTOR Rectangulo: " << getNombre() <<
std::endl;
    }
};
// Función para describir cualquier forma
void describirForma(const FormaGeometrica* forma) {
    std::cout << "\n--- Descripcion de Forma ---" << std::endl;
    forma->dibujar();
    std::cout << "    Area: " << forma->calcularArea() << std::endl;
    std::cout << "    Color: " << forma->getColor() << std::endl;
}
int main() {
    FormaGeometrica* ptrCirculo = new Circulo("Mi Circulo", "Rojo", 7.0);
    FormaGeometrica* ptrRectangulo = new Rectangulo("Mi Rectangulo", "Azul",
4.0, 5.0);
    describirForma(ptrCirculo);
    describirForma(ptrRectangulo);
    std::vector<FormaGeometrica*> misFormas;
    misFormas.push_back(ptrCirculo);
    misFormas.push_back(ptrRectangulo);
    misFormas.push_back(new Circulo("Otro Circulo", "Verde", 3.0));
    std::cout << "\n--- Procesando todas las formas del vector ---" <<
std::endl;
    for (const FormaGeometrica* f : misFormas) {
        f->dibujar();
    }
    std::cout << "\n--- Liberando memoria ---" << std::endl;
    delete misFormas[2]; // Borra "Otro Circulo"
    delete ptrRectangulo;
    delete ptrCirculo;

    std::cout << "Fin del programa Priscila Bellido" << std::endl;
    return 0;
}

```