

Programación Orientada a Objetos

Programación de Computadores

Contenido

Qué es POO

Clases

Constructores

Structures

Qué es POO

¿Qué es Programación Orientada a Objetos?

Es un paradigma de programación que organiza el código en torno a "objetos" en lugar de funciones o procedimientos

Los objetos son estructuras que representan tanto datos como el comportamiento asociado a esos datos, lo que facilita la creación de programas más organizados, modulares y reutilizables.

Conceptos fundamentales

- **Clases y Objetos**

- **Clase:** Es el "molde" o "plantilla" que define los atributos (datos) y métodos (funciones) de un objeto. En otras palabras, una clase describe qué características y comportamientos tendrán los objetos que se creen a partir de ella.
- **Objeto:** Es una instancia de una clase. Cada objeto tiene su propio estado y comportamiento, definidos por la clase de la cual proviene.

- **Atributos y Métodos**

- **Atributos:** Representan el estado o características de un objeto.
- **Métodos:** Son funciones asociadas a la clase, que definen los comportamientos o acciones que los objetos pueden realizar.

Pilares/Principios de la POO

Abstracción: Simplifica la realidad modelando solo los aspectos relevantes. Las clases representan conceptos importantes, sin necesidad de incluir todos los detalles posibles.

Encapsulamiento: Permite que los atributos y métodos estén "encapsulados" dentro de los objetos, protegiendo los datos de modificaciones no autorizadas y exponiendo solo las funciones necesarias.

Herencia: Facilita la creación de nuevas clases (subclases) a partir de una clase existente (superclase), heredando sus atributos y métodos. Esto promueve la reutilización del código.

Polimorfismo: Permite que objetos de diferentes clases puedan ser tratados de la misma manera si comparten una misma interfaz, lo que permite que el mismo método funcione de manera diferente según el objeto.

Abstracción

Significa enfocarse en la características importantes para resolver el problema.

Es representar objetos del mundo real, dejando a un lado la complejidad asociada.

En C++ (y otros lenguajes de programación), `class`, `struct`, `interface`, `union` y `enum` son constructos de datos fundamentales que permiten modelar y organizar la información de distintas maneras. A continuación se explica cada uno de ellos:

Tipo	Descripción	Acceso por Defecto	Uso Típico
class	Combina atributos y métodos en POO, admite encapsulación, herencia, polimorfismo.	Privado	Modelar objetos complejos y aplicar principios de POO.
struct	Agrupar datos simples o estructura similar a una clase, pero con acceso público por defecto.	Público	Modelar datos simples, especialmente cuando no se requiere encapsulación estricta.
interface	En C++ se simula con una clase abstracta con métodos virtuales puros. Define una lista de métodos sin implementación.	Privado (cuando se usa <code>class</code>)	Definir contratos de métodos que clases derivadas deben implementar.
union	Almacena múltiples tipos en la misma ubicación de memoria; solo un miembro puede contener un valor a la vez.	Público	Ahorrar memoria en situaciones de bajo nivel, cuando solo un miembro se usa a la vez.
enum	Define un conjunto de constantes con nombre, cada una representando un valor entero.	Público	Listas de valores específicos y conocidos, como días de la semana o estados de un proceso.

Encapsulamiento

permite restringir el acceso directo a los atributos (datos) de una clase y controlar cómo se accede o modifica esa información a través de métodos específicos.

protege los datos internos de una clase y exponiéndolos solo mediante interfaces controladas.

Objetivo del Encapsulamiento

El objetivo principal del encapsulamiento es:

- **Ocultar la complejidad interna** de una clase.
- **Controlar el acceso a los datos** para garantizar la integridad y consistencia del estado del objeto.

Niveles de acceso

Controlan la visibilidad de los miembros (atributos y métodos) de una clase. Los niveles de acceso más comunes son:

- **private**: Los miembros declarados como **private** solo son accesibles dentro de la misma clase. Es el nivel de acceso más restrictivo y se usa para ocultar los detalles internos.
- **protected**: Los miembros **protected** son accesibles dentro de la misma clase y en sus clases derivadas. Se utiliza cuando se desea dar acceso a los miembros a las clases que heredan.
- **public**: Los miembros **public** son accesibles desde cualquier parte del programa. Generalmente, los métodos públicos son interfaces que exponen los servicios que ofrece la clase.

Herencia

permite crear nuevas clases (denominadas **clases derivadas** o **subclases**) a partir de clases existentes (conocidas como **clases base** o **superclases**).

En la herencia, la clase derivada hereda los atributos y métodos de la clase base, lo que permite reutilizar el código y agregar o modificar funcionalidades en la subclase sin necesidad de rehacerlo todo desde cero.

Las relaciones son de “es un” o “es una”. P.ej. Un perro es un animal.

Ventajas de la Herencia

- **Reutilización de Código:** Permite reutilizar el código de la clase base en las clases derivadas, evitando duplicación y manteniendo el código más limpio y fácil de mantener.
- **Organización Jerárquica:** Facilita la creación de una jerarquía de clases, lo que permite representar relaciones "es-un" (por ejemplo, un **Gerente** es un **Empleado**).
- **Polimorfismo:** Permite tratar objetos de una clase derivada como si fueran de la clase base, facilitando el uso de un código más general.

Consideraciones Importantes sobre la Herencia en C++

Constructores y Destructores: La clase derivada hereda los atributos de la clase base, pero no su constructor ni destructor. La subclase debe llamar explícitamente al constructor de la clase base (como se ve en `Gerente`).

Acceso Protegido (`protected`): Los atributos `protected` en la clase base son accesibles en la clase derivada, pero permanecen inaccesibles desde fuera de la jerarquía de clases.

Sobreescritura de Métodos: Una clase derivada puede redefinir los métodos de la clase base, proporcionando su propia implementación. Esto se conoce como **sobreescritura**.

Composición

A diferencia de la herencia (relación "es-un"), la composición representa una **relación "tiene-un"** y permite construir objetos complejos a partir de otros objetos.

se refiere a la creación de una clase que contiene otros objetos como parte de su estructura. Los objetos que forman la clase compuesta **solo existen mientras exista el objeto contenedor**. Si se destruye el objeto contenedor, también se eliminan los objetos contenidos, porque dependen del objeto principal para su existencia.

Por ejemplo, en un sistema de gestión de vehículos, un **Coche** podría estar compuesto por instancias de clases como **Motor**, **Rueda**, y **Asiento**. Si el **Coche** deja de existir, entonces sus **Motor**, **Rueda** y **Asiento** también.

Polimorfismo



permite que un objeto se comporte de diferentes maneras según el contexto en el que se utilice.

El término "polimorfismo" proviene del griego y significa "muchas formas", lo cual describe su esencia: un método o una función puede actuar de distintas formas dependiendo del tipo de objeto que lo esté utilizando.

Se usa en conjunto con herencia y composición.

Tipos de Polimorfismo

Existen dos tipos principales de polimorfismo en C++:

1. **Polimorfismo en tiempo de compilación** (estático) 
2. **Polimorfismo en tiempo de ejecución** (dinámico) 

Polimorfismo en Tiempo de Compilación (Estático)

Este tipo de polimorfismo se resuelve en tiempo de compilación y se logra a través de **sobrecarga de funciones** y **sobrecarga de operadores**.

Sobrecarga de Funciones

La **sobrecarga de funciones** permite definir múltiples versiones de una función con el mismo nombre, pero con diferentes parámetros.

Sobrecarga de Operadores

La **sobrecarga de operadores** permite redefinir el comportamiento de operadores como $+$, $-$, $*$, entre otros, para que funcionen con objetos de clases definidas por el usuario.

Polimorfismo en Tiempo de Ejecución (Dinámico)

El polimorfismo en tiempo de ejecución se logra a través de **herencia** y **métodos virtuales**. Permite que un puntero o una referencia a la clase base pueda llamar a métodos que están sobrescritos en las clases derivadas. En este caso, el método correcto se elige en tiempo de ejecución, dependiendo del tipo real del objeto.

Ejemplo de Polimorfismo en Tiempo de Ejecución

Supongamos que tenemos una clase base `Animal` con un método `hacerSonido()`. Cada clase derivada (`Perro` y `Gato`) implementará su propia versión de este método.

Ventajas del Polimorfismo

Flexibilidad: Permite escribir código más general y reutilizable, ya que se pueden manipular diferentes tipos de objetos de una manera uniforme.

Extensibilidad: Facilita la extensión del sistema, ya que se pueden agregar nuevas clases derivadas que redefinan el comportamiento sin cambiar el código existente.

Mantenibilidad: Promueve la separación de responsabilidades y permite mantener el código de forma más organizada.

Classes

class (Clase)

Plantilla

Representa una abstracción

Todo objeto es una instancia de una clase

Un objeto puede tener muchas instancias.

Cada instancia puede actuar independientemente.

Una **class** es una estructura de datos en la Programación Orientada a Objetos (POO) que combina atributos (datos) y métodos (funciones) para representar un objeto en el código. En C++, las clases permiten especificar **niveles de acceso** (**public**, **protected**, y **private**) para los atributos y métodos.

Sintaxis de clases

```
class <nombre> {  
    // Los miembros son privados por defecto  
  
    <modificadores>:  
        <variables miembro>  
        <funciones miembro>  
};
```

- **class**: Palabra clave que define una clase.
- **<nombre>**: Nombre de la clase.
- **modificadores**: Controlan el nivel de acceso de los miembros de la clase (**public**, **protected**, **private**).
- **variables miembro**: Atributos o datos que pertenecen a la clase.
- **funciones miembro**: Métodos o funciones que manipulan los datos de la clase.

Por defecto, en C++, los miembros de una clase son **privados** si no se especifica un modificador de acceso.

Archivos de encabezado `.h` en C++

¿Qué son los archivos `.h`?

- Archivos de encabezado en C++.
- Contienen **declaraciones** de clases, funciones y variables.

Propósito

- Separar la **interfaz** del código (qué hace) de la **implementación** (cómo lo hace).
- Facilitar la **organización y modularidad** del código.

¿Qué contienen?

- **Declaraciones de clases** y sus métodos.
- **Firmas de funciones**, sin implementación.
- **Definiciones de constantes y variables globales**.
- **Macros y directivas** de preprocesador (`#define`, `#ifndef`).

Beneficios

- Mejora la **reutilización** del código.
- Promueve la **separación de interfaz e implementación**.
- **Evita duplicación** mediante “include guards” (`#ifndef` / `#define`).

```
// Carta.h
#ifndef CARTA_H
#define CARTA_H

class Carta {
public:
    Carta(int valor);
    int obtenerValor() const;
private:
    int valor;
};

#endif
```

#pragma once

Qué es `#pragma once`?

- Una **directiva de preprocesador** que se coloca al inicio de un archivo de encabezado (.h).
- Indica al compilador que incluya el archivo **solo una vez** por cada unidad de compilación.

Función de `#pragma once`

- Evita la **duplicación de código** al prevenir múltiples inclusiones del mismo archivo de encabezado.
- Simplifica el código al **reemplazar los include guards** (`#ifndef`, `#define`, `#endif`).

Ventajas de Usar `#pragma once`

- **Código más limpio:** Se elimina la necesidad de los “include guards” manuales.
- **Menos errores:** Minimiza errores de duplicación y reduce conflictos en proyectos grandes.
- **Rendimiento:** Puede mejorar la velocidad de compilación al reducir el trabajo del preprocesador (depende del compilador).

Compatibilidad

- Es **soportado por la mayoría de compiladores modernos** (GCC, Clang, MSVC).
- No es parte del estándar oficial de C++, pero es ampliamente adoptado.

```
// Carta.h
#pragma once

class Carta {
public:
    Carta(int valor);
    int obtenerValor() const;
private:
    int valor;
};
```

Ejemplo de implementación

C Coche	
▣ float combustible	// Cantidad de combustible
▣ float velocidad	// Velocidad actual
▣ int pasajeros	// Número de pasajeros
● void llenarCombustible(float cantidad)	// Método para llenar combustible
● void acelerar()	// Método para acelerar el coche

1_ejemplo_coche > h Coche.h > ...

```
1  #pragma once
2  class Coche {
3  private:
4      float combustible;
5      float velocidad;
6      int pasajeros;
7  public:
8      void LlenarCombustible(float cantidad);
9      void Acelerar();
10     void Frenar();
11     void SubirPasajeros(int cantidad);
12     void MostrarDatos();
13 };
```

1_ejemplo_coche > C++ Coche.cpp > ...

```
1  #include "Coche.h"
2  #include <iostream>
3
4  void Coche::LlenarCombustible(float cantidad) {
5      combustible = cantidad;
6  }
7
8  void Coche::Acelerar() {
9      velocidad ++; // Aumenta la velocidad
10     combustible -= 5; // Reduce el combustible
11 }
12
13 void Coche::Frenar() {
14     velocidad = 0; // Frena el coche
15 }
16
17 void Coche::SubirPasajeros(int cantidad) {
18     pasajeros = cantidad;
19 }
20
21 void Coche::MostrarDatos() {
22     std::cout << "Combustible: " << combustible << " litros" << std::endl;
23     std::cout << "Velocidad: " << velocidad << " km/h" << std::endl;
24     std::cout << "Pasajeros: " << pasajeros << std::endl;
25 }
```

Ejemplo de implementación

1_ejemplo_coche > C++ main.cpp > ...

```
1  #include "Coche.h"
2
3  int main() {
4      Coche coche;
5      coche.LlenarCombustible(50);
6      coche.Acelerar();
7      coche.Acelerar();
8      coche.Acelerar();
9      coche.MostrarDatos();
10
11     return 0;
12 }
```

Instancia de clase



Constructores y Destrucciones

Constructor

Tipo especial de función miembro en una clase que se llama automáticamente cuando se crea un objeto de esa clase. Su propósito principal es **inicializar** el objeto, es decir, asignar valores iniciales a sus atributos o realizar cualquier configuración necesaria antes de que el objeto se use.

Características Clave de un Constructor

1. **Mismo Nombre que la Clase:** El constructor debe tener el mismo nombre que la clase.
2. **Sin Tipo de Retorno:** Los constructores no tienen tipo de retorno, ni siquiera `void`.
3. **Invocación Automática:** Se llama automáticamente cuando se crea un objeto de la clase.

Tipos de Constructores en C++

Existen varios tipos de constructores en C++, entre ellos:

1. **Constructor por Defecto:** Un constructor que no toma argumentos.
2. **Constructor Parametrizado:** Un constructor que recibe parámetros para inicializar los atributos del objeto con valores específicos.
3. **Constructor de Copia:** Un constructor especial que crea una copia de otro objeto de la misma clase.

Ejemplos de Constructores

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

class Persona {
private:
    string nombre;
    int edad;

public:
    // Constructor por defecto
    Persona() : nombre("Desconocido"), edad(0) {
        cout << "Constructor por defecto llamado." << endl;
    }

    // Constructor parametrizado
    Persona(const string& nombre, int edad) : nombre(nombre), edad(edad) {
        cout << "Constructor parametrizado llamado." << endl;
    }

    // Método para mostrar los datos
    void mostrarDatos() const {
        cout << "Nombre: " << nombre << ", Edad: " << edad << endl;
    }
};

int main() {
    Persona p1;                // Llama al constructor por defecto
    Persona p2("Carlos", 25);  // Llama al constructor parametrizado

    p1.mostrarDatos();
    p2.mostrarDatos();

    return 0;
}
```

Destructor

Un **destructor** es una función especial en C++ que se llama automáticamente cuando un objeto es destruido. Su propósito es **liberar recursos** y realizar cualquier limpieza necesaria antes de que el objeto se elimine de la memoria. Los destructores son especialmente importantes en C++ porque permiten gestionar la memoria y otros recursos de forma eficiente, evitando fugas de memoria (memory leaks).

Características Clave de un Destructor

1. **Mismo Nombre que la Clase, precedido por ~:** El destructor tiene el mismo nombre que la clase, pero se escribe con un símbolo ~ al inicio. Por ejemplo, si la clase es **Persona**, el destructor sería **~Persona**.
2. **Sin Parámetros y Sin Tipo de Retorno:** Los destructores no aceptan parámetros y no tienen tipo de retorno, ni siquiera **void**.
3. **Invocación Automática:** El destructor se llama automáticamente cuando el objeto sale de su alcance o cuando se utiliza **delete** para destruir un objeto dinámico (creado con **new**).
4. **Uno por Clase:** Una clase solo puede tener un destructor, no se pueden sobrecargar destructores.

Ejemplo de Destructor

```
public:
    // Constructor que abre un archivo
    Archivo(const string& nombre) {
        archivo.open(nombre, ios::out);
        if (archivo.is_open()) {
            cout << "Archivo abierto: " << nombre << endl;
        } else {
            cout << "Error al abrir el archivo: " << nombre << endl;
        }
    }

    // Destructor que cierra el archivo
    ~Archivo() {
        if (archivo.is_open()) {
            archivo.close();
            cout << "Archivo cerrado." << endl;
        }
    }
}
```

Structures

Estructuras

En C++, una **estructura** (o **struct**) es un tipo de dato compuesto que permite agrupar varias variables (llamadas **miembros**) en una sola unidad.

Aunque las estructuras y las clases son muy similares en C++ (ambas permiten definir atributos y métodos), existen algunas diferencias sutiles entre ellas, especialmente en cuanto a los **niveles de acceso** y su uso recomendado.

```
struct NombreEstructura {  
    // Miembros de la estructura (atributos y opcionalmente métodos)  
    tipo atributo1;  
    tipo atributo2;  
    ...  
};
```

```
#include <iostream>  
using namespace std;  
  
struct Punto {  
    int x; // Coordenada x  
    int y; // Coordenada y  
  
    void mostrar() const {  
        cout << "Punto(" << x << ", " << y << ")" << endl;  
    }  
};  
  
int main() {  
    Punto p1;  
    p1.x = 5;  
    p1.y = 10;  
  
    p1.mostrar(); // Muestra: Punto(5, 10)  
  
    return 0;  
}
```

Diferencias entre `struct` y `class` en C++

Aunque `struct` y `class` son muy similares en C++, existen dos diferencias importantes:

1. Acceso por Defecto:

- En una **clase** (`class`), los miembros son **privados** (`private`) por defecto.
- En una **estructura** (`struct`), los miembros son **públicos** (`public`) por defecto.

2. Contexto de Uso:

- `struct` se suele usar para agrupar datos simples, sin comportamientos complejos, o en contextos donde solo se necesita almacenar información sin la necesidad de control estricto de acceso.
- `class` se utiliza principalmente en la Programación Orientada a Objetos para modelar objetos con atributos privados y métodos que encapsulan su comportamiento.

Cuándo Usar `struct` en C++

`struct` es útil en los siguientes casos:

1. **Agrupación de Datos Simples:** Cuando necesitas agrupar datos simples sin mucha lógica asociada, como coordenadas, puntos en un plano, o configuraciones.
2. **Compatibilidad con C:** `struct` es una característica heredada del lenguaje C, donde no existe `class`. Esto lo hace útil en situaciones donde necesitas interoperabilidad con código C o en aplicaciones de bajo nivel.
3. **Código de Acceso Público Simple:** Si tienes un conjunto de datos que no necesita encapsulación estricta y puede estar accesible públicamente, usar `struct` hace el código más claro y directo.
4. **Estructuras de Datos en Programación Funcional:** Si trabajas en un contexto de programación funcional o estructural, donde se prefieren estructuras de datos con atributos públicos y sin métodos complejos, `struct` es más adecuado.

Ejemplo de Uso

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

struct Direccion {
    string calle;
    string ciudad;
    string pais;
    int codigo_postal;
};

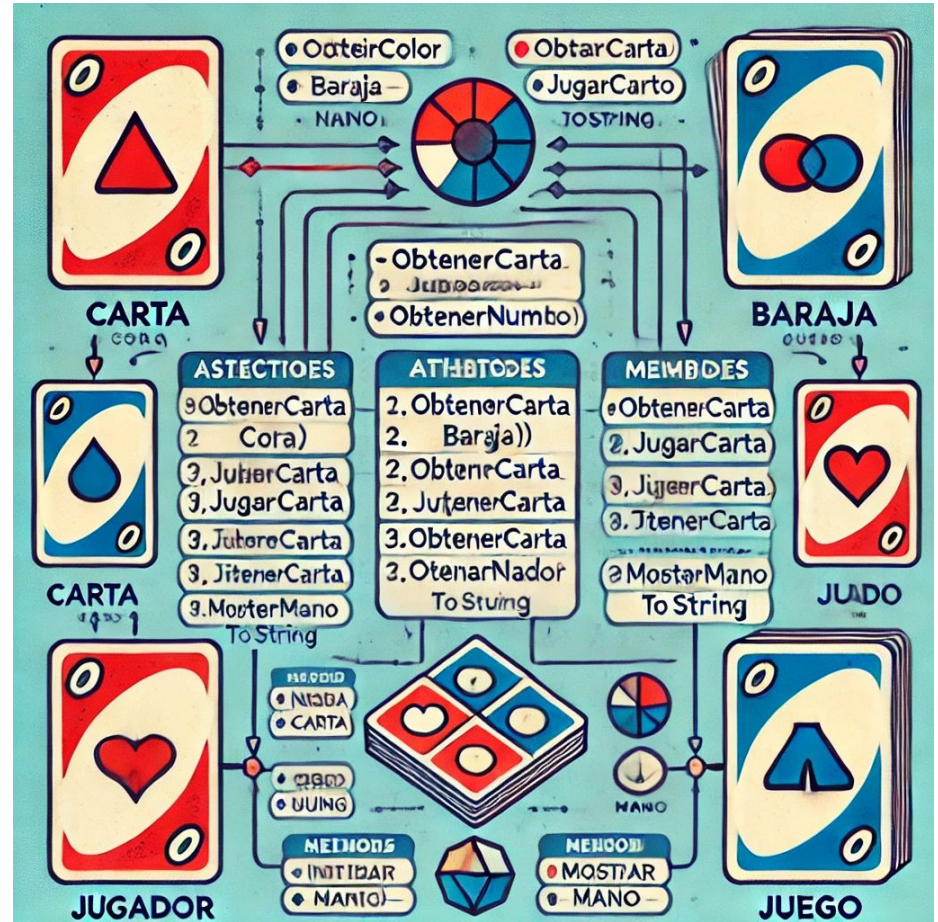
int main() {
    Direccion direccion1 = {"Av. Principal", "Bogotá", "Colombia", 110111};

    cout << "Calle: " << direccion1.calle << endl;
    cout << "Ciudad: " << direccion1.ciudad << endl;
    cout << "País: " << direccion1.pais << endl;
    cout << "Código Postal: " << direccion1.codigo_postal << endl;

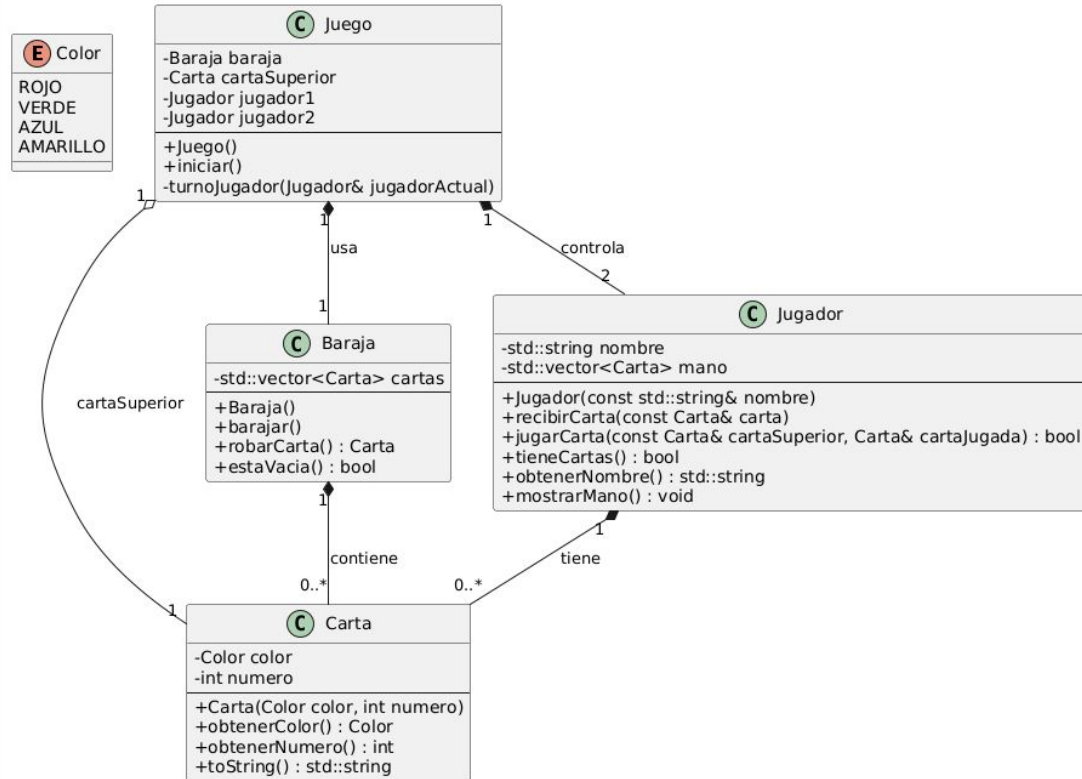
    return 0;
}
```

Práctica # 1

Juego de Uno



Juego de Uno - Diagrama de Clases



Ejercicio Propuesto

1. Entender el código.
2. Cambios a implementar
 - Extiende el juego para incluir más jugadores.
 - Implementa cartas especiales como "Reversa", "Salta" y "Toma dos".
 - Mejora la interacción con el usuario permitiendo ingresar nombres de jugadores y elegir cartas a jugar manualmente.