INVESTIGACION CONCEPTOS BASICOS POO ENFOCADOS EN C++

Victor Manuel Morales Sachica

U.C.C

INTRODUCCION

La programación orientada a objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos", los cuales representan entidades del mundo real y tienen características (atributos) y comportamientos (métodos) asociados. En la POO, los programas se estructuran alrededor de clases y objetos.

INVESTIGACION DE POO EN C++

Clases y Objetos

Una clase es una estructura de programación que encapsula un conjunto de datos y métodos relacionados. Es una plantilla para crear objetos que comparten comportamientos comunes. Una clase define la estructura y el comportamiento de los objetos que se crearán a partir de ella.

Un objeto es una instancia única de una clase. Representa una entidad del mundo real y tiene un estado (almacenado en atributos) y un comportamiento (definido por métodos). Los objetos son fundamentales en la programación orientada a objetos ya que permiten modelar y trabajar con entidades de manera modular y reutilizable.

EJEMPLO:

Una clase Empleado podría representar a cada trabajador, con atributos como nombre, salario y departamento, y métodos para actualizar esta información

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

class Empleado {
public:
    std::string nombre;
    float salario;
    std::string departamento;

    void mostrarInformacion() {
        std::cout << "Nombre: " << nombre << ", Salario: $" << salario << ", Departamento: " << departamento << std::endl;
    }
};

int main() {
    Empleado empleado1;
    empleado1.nombre = "Juan";
    empleado1.salario = 3000.0;
    empleado1.departamento = "Ventas";

    empleado1.mostrarInformacion();

    return 0;
}
```

Encapsulamiento

Encapsulamiento es uno de los principios fundamentales de la programación orientada a objetos que se refiere a la ocultación de los detalles internos de un objeto y el acceso

controlado a sus datos y métodos. En C++, esto se logra definiendo miembros de clase como públicos, privados o protegidos. Los datos privados de una clase solo pueden ser accedidos y modificados por los métodos de esa clase, lo que ayuda a prevenir modificaciones accidentales y asegura la coherencia de los datos.

EJEMPLO:

los atributos de la clase Empleado se vuelven privados y proporcionar métodos públicos para acceder y modificar estos atributos.

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

class Empleado {
private:
    std::string nombre;
    float salario;
    std::string departamento;

public:
    void establecerNombre(std::string nombre) {
        this->nombre = nombre;
    }

    void establecerSalario(float salario) {
        this->salario = salario;
    }

    void establecerDepartamento(std::string departamento) {
        this->departamento = departamento;
    }

    void mostrarInformacion() {
        std::cout << "Nombre: " << nombre << ", Salario: $" << salario << ", Departamento: " << departamento << std::endl;
    }
};

int main() {
    Empleado empleado1;
    empleado1.establecerNombre("Juan");
    empleado1.establecerSalario(3000.0);
    empleado1.establecerDepartamento("Ventas");

    empleado1.mostrarInformacion();

    return 0;
}
```

Herencia

La herencia es un mecanismo que permite que una clase (llamada subclase o clase derivada) herede propiedades y comportamientos de otra clase (llamada superclase o clase base). La subclase puede extender o especializar la funcionalidad de la superclase y también puede agregar nuevos miembros. La herencia facilita la reutilización del código y la organización jerárquica de las clases.

EJEMPLO:

Creamos una clase Gerente que hereda de la clase Empleado y añade atributos y métodos específicos para los gerentes, como el número de subordinados.

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

class Empleado {
protected:
    std::string nombre;
    float salario;
    std::string departamento;

public:
    void establecerNombre(std::string nombre) {
        this->nombre = nombre;
    }

    void establecerSalario(float salario) {
        this->salario = salario;
    }

    void establecerDepartamento(std::string departamento) {
        this->departamento = departamento;
    }

    void mostrarInformacion() {
        std::cout << "Nombre: " << nombre << ", Salario: $" << salario << ", Departamento: " << departamento << std::endl;
    }
};

class Gerente : public Empleado {
private:
    int numSubordinados;

public:
    void establecerNumSubordinados(int num) {
        numSubordinados = num;
    }

    void mostrarInformacion() {
        std::cout << "Nombre: " << nombre << ", Salario: $" << salario << ", Departamento: " << departamento << ", Subordinados: " << numSubordinados << std::endl;
    }
};

int main() {
    Gerente gerente1;
    gerente1.establecerNombre("Laura"); // Aquí se establece el nombre usando el método de la clase base
    gerente1.establecerSalario(5000.0);
    gerente1.establecerDepartamento("Gerencia");
    gerente1.establecerNumSubordinados(5);

    gerente1.mostrarInformacion();

    return 0;
}
```

Polimorfismo

El polimorfismo es la capacidad de objetos de diferentes clases de responder al mismo mensaje de manera diferente. En C++, esto se logra mediante el uso de funciones virtuales y la sobreescritura de métodos. El polimorfismo de tiempo de ejecución permite que un objeto se comporte de manera distinta dependiendo de su tipo en tiempo de ejecución, lo que facilita el diseño de sistemas flexibles y extensibles.

EJEMPLO:

tenemos una función calcularSalario() que acepta un objeto Empleado y calcula el salario.

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

class Empleado {
public :
    std::string nombre;
    float salarioBase;

public:
    Empleado(std::string nombre, float salarioBase) : nombre(nombre), salarioBase(salarioBase) {}

    virtual float calcularSalario() {
        return salarioBase;
    }
};

class Gerente : public Empleado {
private:
    float bono;

public:
    Gerente(std::string nombre, float salarioBase, float bono) : Empleado(nombre, salarioBase), bono(bono) {}

    float calcularSalario() override {
        return salarioBase + bono;
    }
};

void imprimirSalario(Empleado& empleado) {
    std::cout << "El salario de " << empleado.nombre << " es: $" << empleado.calcularSalario() << std::endl;
}

int main() {
    Empleado empleado1("Juan", 3000.0);
    Gerente gerente1("Laura", 5000.0, 2000.0);

    imprimirSalario(empleado1);
    imprimirSalario(gerente1);

    return 0;
}

```

Abstracción

La abstracción es el proceso de identificar las características esenciales de un objeto y definir una interfaz simplificada para interactuar con él, ocultando los detalles de implementación subyacentes. En C++, se logra mediante la definición de clases y la separación entre la interfaz pública y la implementación privada de esas clases. La abstracción permite manejar la complejidad de los sistemas al enfocarse en los aspectos relevantes y ocultar la complejidad innecesaria.

EJEMPLO:

definimos una clase Forma, podemos abstraer los detalles específicos de cada forma geométrica y proporcionar métodos genéricos como calcularArea() y calcularPerimetro().

clases.cpp | encapsulamiento.cpp | herencia.cpp | polimorfismo.cpp | abstraccion.cpp

```cpp
#include <iostream>

class Forma {
public:
    virtual float calcularArea() = 0;
    virtual float calcularPerimetro() = 0;
};

class Rectangulo : public Forma {
private:
    float base, altura;

public:
    Rectangulo(float base, float altura) : base(base), altura(altura) {}

    float calcularArea() override {
        return base * altura;
    }

    float calcularPerimetro() override {
        return 2 * (base + altura);
    }
};

int main() {
    Rectangulo rectangulo(5.0, 3.0);
    std::cout << "Area del rectangulo: " << rectangulo.calcularArea() << std::endl;
    std::cout << "Perimetro del rectangulo: " << rectangulo.calcularPerimetro() << std::endl;

    return 0;
}

```

Mensajes y Métodos

En la programación orientada a objetos, los objetos interactúan entre sí enviándose mensajes. Un mensaje es una solicitud de ejecución de un método en un objeto. Un método es una función miembro de una clase que define el comportamiento de un objeto en respuesta a un mensaje. Los métodos encapsulan la lógica y el comportamiento de un objeto, permitiendo su reutilización y modularidad.

EJEMPLO:

Tener un método ladrar() que representa la acción de ladrar del perro. Y otro objeto envía un mensaje para que ladre.

```cpp
#include <iostream>

class Perro {
public:
    void ladrar() {
        std::cout << "¡Guau guau!" << std::endl;
    }
};

int main() {
    Perro miPerro;
    miPerro.ladrar(); // Mensaje enviado al objeto miPerro para que ladre

    return 0;
}
```

Atributos y Propiedades

Los atributos son variables miembro de una clase que representan el estado de un objeto. Estos atributos definen las características o propiedades de un objeto. Las propiedades son atributos que se exponen al exterior de la clase a través de métodos de acceso (getters) y métodos de modificación (setters). El uso de propiedades proporciona un control más granular sobre el acceso y la modificación de los datos de un objeto, mejorando la encapsulación y la seguridad.

EJEMPLO:

Una clase CuentaBancaria podría tener un atributo privado saldo y proporcionar métodos públicos para acceder y modificar este saldo.

```cpp
#include <iostream>

class CuentaBancaria {
private:
    float saldo;

public:
    void establecerSaldo(float nuevoSaldo) {
        saldo = nuevoSaldo;
    }

    float obtenerSaldo() {
        return saldo;
    }
};

int main() {
    CuentaBancaria cuenta;
    cuenta.establecerSaldo(1000.0);
    std::cout << "Saldo actual: $" << cuenta.obtenerSaldo() << std::endl;

    return 0;
}

```

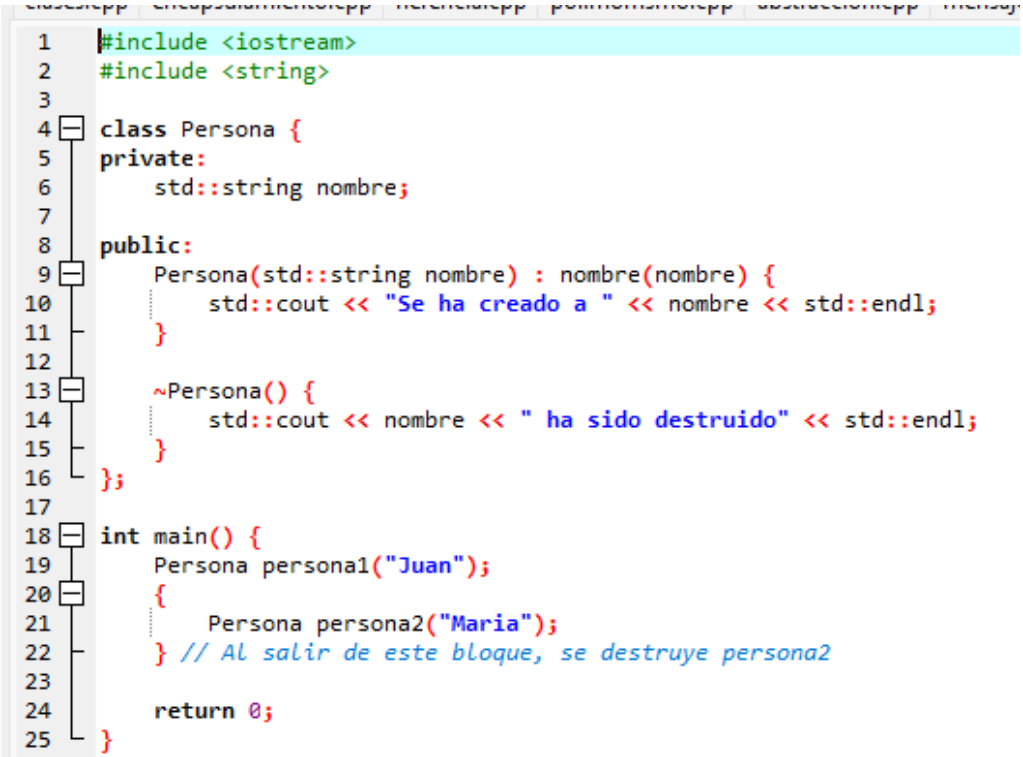
Constructores y Destructores

Los constructores son métodos especiales que se llaman automáticamente cuando se crea un objeto. Su función principal es inicializar el estado inicial del objeto asignando valores a los atributos y realizando cualquier otra inicialización necesaria. Los destructores son métodos especiales que se llaman automáticamente cuando un objeto se destruye o sale del ámbito. Su función principal es liberar cualquier recurso asignado dinámicamente por el objeto y realizar cualquier otra limpieza necesaria antes de que el objeto sea eliminado de la memoria.

Estos conceptos son esenciales para comprender la programación orientada a objetos en C++ y son fundamentales para desarrollar aplicaciones eficientes, modularizadas y mantenibles.

EJEMPLO:

Una clase Persona podría tener un constructor para inicializar el nombre y un destructor para imprimir un mensaje al ser destruida.



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

class Persona {
private:
    std::string nombre;

public:
    Persona(std::string nombre) : nombre(nombre) {
        std::cout << "Se ha creado a " << nombre << std::endl;
    }

    ~Persona() {
        std::cout << nombre << " ha sido destruido" << std::endl;
    }
};

int main() {
    Persona persona1("Juan");
    {
        Persona persona2("Maria");
    } // Al salir de este bloque, se destruye persona2

    return 0;
}
```

REFERENCIAS

https://www.tutorialspoint.com/cplusplus/cpp_object_oriented.htm

http://www.cplusplus.com/doc/tutorial/classes2/

https://www.geeksforgeeks.org/inheritance-in-cpp/