.

Instituto Tecnológico de Las Américas

TEMA:

Resumen Programación orientada a objetos : herencia

PARTICIPANTE:

Robert Adolfo Santana Rodríguez

FACILITADOR:

Francis Ramírez

FECHA:

08-mayo de 2025

Introducción a la Herencia

La programación orientada a objetos (POO) es un paradigma de programación centrado en objetos que interactúan entre sí para resolver problemas. Uno de los pilares fundamentales de la POO es la herencia, un mecanismo que permite crear nuevas clases a partir de clases existentes. La herencia fomenta la reutilización del código y permite modelar relaciones jerárquicas entre tipos de datos.

A través de la herencia, una clase denominada subclase (o clase derivada) puede heredar atributos y comportamientos de otra clase, llamada superclase (o clase base). Este mecanismo se asemeja a la herencia biológica, donde los descendientes heredan características de sus antecesores. En la programación, esto se traduce en sistemas más fáciles de mantener, extender y modificar.

Jerarquías de Clases

En C++, las clases se pueden organizar en jerarquías donde las clases más generales actúan como bases para clases más especializadas. La relación entre una subclase y su superclase es del tipo "es-un" (is-a). Por ejemplo, si tenemos una clase Empleado y una clase Gerente, podríamos decir que un Gerente es un Empleado, y por lo tanto, heredará los atributos comunes como nombre, identificador y salario, y podrá añadir o modificar funcionalidades específicas como el bono de gestión.

Este tipo de diseño permite construir sistemas modulares y escalables. Al modelar entidades del mundo real como jerarquías de clases, el código refleja mejor la lógica del dominio de aplicación.

3. Sintaxis de la Herencia en C++

En C++, se declara una subclase utilizando la siguiente sintaxis:

```
class Subclase : modo_acceso Superclase {
    // Miembros de la subclase
};
```

El modo_acceso puede ser public, protected o private, y determina cómo se heredan los miembros de la superclase. Si se utiliza public, los miembros public y protected de la superclase se mantienen con el mismo nivel de acceso. En cambio, con private, todos los miembros accesibles se vuelven private en la subclase.

Los constructores y destructores no se heredan directamente, pero pueden ser invocados desde la subclase. Por ejemplo, se puede invocar al constructor de la superclase en la lista de inicialización del constructor de la subclase:

```
class Empleado {
public:
    Empleado(string nombre);
};


class Gerente : public Empleado {
public:
    Gerente(string nombre, double bono) : Empleado(nombre) {
        this->bono = bono;
    }
private:
    double bono;
};
```

4. Accesibilidad y Protección de Miembros

Los miembros de una clase pueden ser declarados como public, protected o private. Esta clasificación controla el acceso desde fuera de la clase y también desde las subclases:

public: accesible desde cualquier parte.

protected: accesible desde la misma clase y desde subclases.

private: accesible solo desde la misma clase.

Un diseño cuidadoso de los niveles de acceso mejora la encapsulación y reduce errores. Se recomienda mantener los datos como private y proporcionar acceso mediante funciones public o protected.

5. Sobreescritura de Métodos

Una subclase puede redefinir un método heredado de la superclase. Esto se conoce como sobreescritura (overriding). Es diferente de la sobrecarga (overloading), donde varios métodos tienen el mismo nombre pero diferentes parámetros.

En C++, se recomienda usar la palabra clave override (disponible desde C++11) para indicar que se está sobreescribiendo un método virtual. Esto ayuda al compilador a detectar errores cuando el método de la subclase no coincide exactamente con el de la superclase.

```
class Empleado {

public:

    virtual void imprimir() const {

        cout << "Empleado";

    }

};


class Gerente : public Empleado {

public:

    void imprimir() const override {

        cout << "Gerente";

    }

};
```

6. Funciones Virtuales y Polimorfismo

El polimorfismo permite que una misma llamada a una función tenga diferentes comportamientos según el tipo de objeto que la invoque. En C++, esto se logra mediante funciones virtuales.

Cuando se declara un método como virtual en una superclase, se habilita la vinculación dinámica, lo que significa que en tiempo de ejecución se determinará cuál implementación del método invocar, según el tipo real del objeto apuntado.

Este comportamiento es posible gracias a la tabla de funciones virtuales (vtable), una estructura interna que mantiene punteros a las funciones virtuales del objeto.

7. Clases Abstractas

Una clase abstracta es aquella que contiene al menos una función virtual pura, la cual se declara con = 0:

```
class Figura {

public:

    virtual void dibujar() const = 0; // Método virtual puro

};
```

Estas clases no pueden ser instanciadas directamente, y su propósito es servir como base para otras clases. Las clases derivadas deben proporcionar implementaciones concretas para todos los métodos virtuales puros.

Las clases abstractas son fundamentales en el diseño de arquitecturas orientadas a interfaces y permiten definir contratos claros para los desarrolladores.

8. Composición vs. Herencia

La herencia no es la única forma de reutilizar código. La composición implica que una clase contiene objetos de otras clases como miembros.

Por ejemplo, una clase Auto puede tener un objeto de tipo Motor sin necesidad de heredar de él. Esto permite mayor flexibilidad y desacoplamiento, ya que se pueden cambiar componentes sin afectar la jerarquía de clases.

Se recomienda utilizar composición cuando la relación entre las clases no es del tipo "es-un" sino "tiene-un" (has-a).

9. Problemas Comunes y Buenas Prácticas

Al usar herencia, es importante evitar ciertos errores comunes:

Uso excesivo de la herencia puede llevar a estructuras rígidas y difíciles de mantener.

Herencia múltiple puede introducir ambigüedades (solucionadas en parte con virtual inheritance).

Violación del Principio de Sustitución de Liskov (LSP), que establece que una subclase debe poder reemplazar a su superclase sin alterar el comportamiento esperado.

Buenas prácticas incluyen el uso de final para evitar sobreescrituras no deseadas y diseñar basándose en interfaces y abstracciones, no en implementaciones concretas.

10. Conclusión

La herencia es un pilar esencial de la programación orientada a objetos que permite modelar jerarquías lógicas, reutilizar código y facilitar el mantenimiento y la extensión del software. En C++, la herencia se implementa de forma poderosa, pero requiere un uso cuidadoso para evitar errores comunes y maximizar la claridad del diseño.

A través del uso adecuado de herencia, funciones virtuales, y clases abstractas, los desarrolladores pueden construir sistemas robustos, modulares y escalables. Comprender estos conceptos es clave para dominar el desarrollo orientado a objetos en C++.