1. **Abstracción**: permite identificar las características relevantes dentro de un conjunto (un objeto por ejemplo), comportamientos comunes para definir nuevos tipos de entidades y ocultar los detalles innecesarios. Se enfoca en qué hace un objeto, en lugar de como lo hace.

Ejemplo:

class Figura {

public:

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

virtual ~Figura() = default;

};

class Circulo : public Figura {

public:

void dibujar() const override {

std::cout << "Dibujando un círculo." << std::endl;

}

};

int main() {

Circulo c;

c.dibujar();

return 0;

}

**Encapsulamiento:** Se basa en aislar un objeto del exterior y ocultar los detalles internos de una clase protegiendo la integridad de sus datos. Cada tipo de objeto expone una interfaz a otros objetos donde se presenta el prototipo de las funciones que puede realizar. Ante una variación en el código de la clase el usuario no necesitará cambiar sus líneas de código sino cambia su interfaz.

Ejemplo:

class Persona {

private:

int edad;

public:

void setEdad(int e) { edad = e; } // Setter

Int getEdad() const { return edad; } // Getter

};

int main() {

Persona p;

p.setEdad(25);

std::cout << "Edad: " << p.getEdad() << std::endl;

return 0;

}

**Herencia:** Permite definir nuevas clases (subclase o derivada) en base a otras ya existentes (base o superclase La clase derivada hereda los atributos y métodos de la clase base, además de poder definir miembros adicionales. Las clases derivadas también pueden redefinir métodos definidos por su clase base.

Ejemplo:

class Animal {

public:

void dormir() const { std::cout << "El animal está durmiendo." << std::endl; }

};

class Perro : public Animal {};

int main() {

Perro p;

p.dormir(); // Método heredado

return 0;

}

**Polimorfismo:** permite utilizar el mismo nombre para funciones de comportamientos diferentes (asociadas a objetos distintos). Cuál de los métodos será invocado se resuelve en tiempo de ejecución cuando un objeto particular lo necesite. Puede ser estático (en tiempo de compilación) o dinámico (en tiempo de ejecución).

Ejemplo:

class Animal {

public:

virtual void hacerSonido() const { cout << "Sonido de animal." << endl; }

};

class Gato : public Animal {

public:

void hacerSonido() const override { cout << "¡Miau!" << endl; }

};

int main() {

Animal\* a = new Gato();

a->hacerSonido(); // Llamada polimórfica

delete a;

return 0;

}

1. **Razones para usar friend al sobrecargar operadores binarios:**

* Cuando el operador necesita acceder a miembros privados o protegidos de la clase.
  + Si el operador es implementado como una función no miembro (global), normalmente no tiene acceso directo a los atributos privados/protegidos de una clase. Declararla como friend le otorga este acceso.
  + Ejemplo típico: operadores como + o << que combinan atributos internos de objetos y no modifican el estado del objeto actual.
* Para permitir la simetría del operador.
  + Los métodos miembro solo permiten que el objeto de la izquierda del operador sea una instancia de la clase en cuestión.
  + Ejemplo: Si a + b es una expresión donde a es un objeto de clase y b es un tipo primitivo, un método miembro no funcionará para b + a, pero una función friend lo hará, asegurando simetría.
* Para mejorar la legibilidad del código.
  + Algunos operadores, como << (para salida) o >> (para entrada), deben funcionar con flujos de E/S (std::ostream, std::istream). Implementarlos como friend permite una sintaxis natural y mejora la claridad.
* Cuando la lógica del operador debe ser independiente del objeto en sí.
  + Operadores como ==, !=, <, > a menudo se implementan como funciones no miembros porque su semántica no modifica el estado del objeto, solo evalúa relaciones.

1. **Mecanismo Completo de Manejo de Excepciones en C++**

En C++, el manejo de excepciones se realiza utilizando las palabras clave try, throw, y catch. Este mecanismo permite manejar errores de manera estructurada, evitando que los errores interrumpan la ejecución normal del programa.

1. Lanzamiento de una Excepción: throw

El proceso de manejo de excepciones comienza cuando se detecta un error en alguna parte del código. Para manejarlo, lanzamos una excepción utilizando la palabra clave throw. Esta instrucción crea un objeto de excepción (puede ser de cualquier tipo, generalmente se usa un tipo de datos derivado de la clase std::exception) y lo transfiere a un manejador de excepciones que se encargará de gestionarlo.

Ejemplo:

throw std::runtime\_error("Ocurrió un error");

2. Propagación de la Excepción: Pila de Llamadas

Una vez que se lanza una excepción, el control del programa se transfiere automáticamente a un bloque catch correspondiente que pueda manejar ese tipo de excepción. Si no se encuentra un manejador en el bloque try donde ocurrió la excepción, la excepción se "propaga" hacia la función que llamó a la función donde ocurrió el error. Este proceso se sigue haciendo hasta encontrar un bloque catch adecuado o hasta que el programa termine si no se encuentra ningún manejador.

La propagación de la excepción se realiza a través de la pila de llamadas. Cada función que llama a otra función y no maneja la excepción pasa la responsabilidad hacia la función que la llamó, hasta que la excepción se maneje o se termine el programa.

3. Manejo de la Excepción: catch

Cuando el manejador adecuado se encuentra, el control pasa al bloque catch. Este bloque puede especificar el tipo de excepción que está dispuesto a manejar. Si la excepción lanzada coincide con el tipo esperado, se ejecuta el código dentro de catch.

Ejemplo de bloque try y catch:

try {

// Intentamos realizar una operación que podría fallar

throw std::runtime\_error("Ocurrió un error");

}

catch (const std::exception& e) {

// Captura la excepción y maneja el error

std::cout << "Error: " << e.what() << std::endl;

}

4. Patrón RAII (Resource Acquisition Is Initialization)

El patrón RAII (que se traduce a "Adquisición de Recursos es Inicialización") es un patrón de diseño fundamental en C++ que ayuda a gestionar la liberación automática de recursos cuando las excepciones ocurren.

RAII se basa en la idea de que los recursos (como memoria, archivos abiertos, conexiones de red, etc.) se adquieren en el momento de la creación de un objeto y se liberan automáticamente cuando el objeto sale de su ámbito de validez (por ejemplo, cuando el objeto es destruido). Esto se logra mediante el uso de destructores.

Cómo RAII Asegura la Liberación de Recursos Ante Excepciones:

Cuando se lanza una excepción, la pila de ejecución se despliega, lo que significa que todos los objetos locales que estaban en el ámbito donde ocurrió la excepción son destruidos. Si un objeto RAII (por ejemplo, una clase que maneja recursos como memoria o archivos) se encuentra en ese ámbito, su destructor se llama automáticamente, liberando así el recurso asociado, incluso si una excepción se lanza.

Ejemplo de RAII con manejo de excepciones:

#include <iostream>

#include <fstream>

#include <stdexcept>

class FileManager {

public:

FileManager(const std::string& filename) {

file.open(filename);

if (!file) {

throw std::runtime\_error("No se pudo abrir el archivo.");

}

}

~FileManager() {

if (file.is\_open()) {

std::cout << "Cerrando el archivo." << std::endl;

file.close();

}

}

void writeData(const std::string& data) {

if (file.is\_open()) {

file << data;

}

}

private:

std::ofstream file;

};

int main() {

try {

FileManager fm("test.txt");

fm.writeData("Esto es un texto de prueba.");

// Si ocurriera una excepción en cualquier parte de la función,

// el destructor de fm se llamará automáticamente para cerrar el archivo.

}

catch (const std::exception& e) {

std::cout << "Error: " << e.what() << std::endl;

}

// Aquí, el destructor de FileManager se llama automáticamente cuando el objeto

// 'fm' sale del ámbito (ya sea por una excepción o por el flujo normal de ejecución).

}

1. Conceptos de Alta Cohesión y Bajo Acoplamiento en Diseño de Software

Alta Cohesión:

Definición: Un módulo o clase tiene alta cohesión cuando sus elementos están estrechamente relacionados en cuanto a su funcionalidad. Es decir, los métodos y atributos dentro de una clase deben estar relacionados entre sí y contribuir al mismo propósito o responsabilidad.

Beneficio: Una alta cohesión mejora la mantenibilidad, la legibilidad y la reutilización del código, ya que las clases o módulos que tienen alta cohesión tienden a ser pequeños, especializados y fáciles de entender.

Bajo Acoplamiento:

Definición: El bajo acoplamiento significa que los módulos o clases deben depender lo menos posible entre sí. Si una clase o módulo necesita interactuar con otro, que sea a través de interfaces claras y no directamente con su implementación interna.

Beneficio: Un bajo acoplamiento facilita la reutilización, el mantenimiento y las pruebas, ya que los cambios en una clase o módulo tienen un impacto mínimo en otras partes del sistema.

Ejemplos:

Alta Cohesión

Violación (mala práctica):

class Empleado {

private:

std::string nombre;

double salario;

public:

Empleado(const std::string& nombre, double salario)

: nombre(nombre), salario(salario) {}

void mostrarDetalles() {

std::cout << "Empleado: " << nombre << ", Salario: " << salario << std::endl;

}

void generarReporte() {

std::cout << "Generando reporte de salario..." << std::endl;

mostrarDetalles();

}

};

Problema: Mezcla la lógica de negocio (información del empleado) y la lógica de presentación (generar reportes).

Refactorización (buena práctica):

class Empleado {

private:

std::string nombre;

double salario;

public:

Empleado(const std::string& nombre, double salario)

: nombre(nombre), salario(salario) {}

void mostrarDetalles() const {

std::cout << "Empleado: " << nombre << ", Salario: " << salario << std::endl;

}

};

class Reporte {

public:

void generarReporte(const Empleado& empleado) {

std::cout << "Generando reporte..." << std::endl;

empleado.mostrarDetalles();

}

};

Mejoras: Empleado solo maneja datos, y Reporte maneja la generación de reportes, respetando la cohesión.

Bajo Acoplamiento

Violación (mala práctica):

class Motor {

public:

void encender() { std::cout << "Motor encendido." << std::endl; }

};

class Carro {

private:

Motor motor; // Acoplamiento fuerte

public:

void arrancar() {

motor.encender();

std::cout << "Carro arrancado." << std::endl;

}

};

Problema: Carro está fuertemente acoplado con Motor.

Refactorización (buena práctica):

class IMotor {

public:

virtual void encender() const = 0;

};

class MotorGasolina : public IMotor {

public:

void encender() const override { std::cout << "Motor a gasolina encendido." << std::endl; }

};

class Carro {

private:

IMotor\* motor; // Acoplamiento bajo

public:

Carro(IMotor\* motor) : motor(motor) {}

void arrancar() {

motor->encender();

std::cout << "Carro arrancado." << std::endl;

}

};

Mejoras: Carro usa una interfaz IMotor, reduciendo el acoplamiento y facilitando cambios en la implementación del motor.

1. **El problema del diamante ocurre cuando una clase hereda de dos clases que comparten una clase base común. Esto crea ambigüedad y duplicación de recursos.**

Ejemplo:

class A { public: void saludar() { std::cout << "A"; } };

class B : public A { };

class C : public A { };

class D : public B, public C { }; // Ambigüedad: ¿Qué 'saludar' usar?

Soluciones

Herencia Virtual: Evita la duplicación de la clase base y resuelve la ambigüedad al asegurarse que solo haya una copia de la clase base en la jerarquía.

class A { public: void saludar() { std::cout << "A"; } };

class B : virtual public A { };

class C : virtual public A { };

class D : public B, public C { };

Composición: Usa composición en lugar de herencia múltiple para evitar el problema completamente.

class A { public: void saludar() { std::cout << "A"; } };

class B { A a; public: void saludar() { a.saludar(); } };

class C { A a; public: void saludar() { a.saludar(); } };

class D { B b; C c; public: void saludar() { b.saludar(); c.saludar(); } };

1. **Vtables en C++**

Vtable (tabla de métodos virtuales) es un mecanismo que permite polimorfismo dinámico en C++. Cuando una clase tiene métodos virtuales, el compilador crea una tabla (vtable) para cada clase que contiene punteros a las implementaciones de esos métodos. El objeto almacena un puntero a la vtable correspondiente.

Funcionamiento:

Sobrecarga de métodos virtuales: Si una clase base tiene un método virtual y una clase derivada lo sobrecarga, la vtable de la clase derivada contiene la dirección del método sobrecargado, en lugar del de la clase base.

Resolución en tiempo de ejecución: Al invocar un método virtual, el programa consulta la vtable del objeto (a través de un puntero a la vtable) para ejecutar la implementación correcta.

Implicaciones:

Rendimiento: La resolución de métodos virtuales es más lenta porque requiere indireccionamiento a través de la vtable (en lugar de acceso directo al método).

Memoria: Cada clase con métodos virtuales necesita una vtable, y cada objeto de esa clase tiene un puntero a la vtable, lo que consume más memoria.

1. **Templates en C++**

Un template en C++ es una característica que permite escribir funciones y clases genéricas, capaces de operar con cualquier tipo de dato sin necesidad de duplicar código para cada tipo.

Problemas que resuelve:

Reutilización de código: Permite escribir funciones y clases que funcionan con diferentes tipos de datos sin repetir el código.

Flexibilidad y generalización: Se puede usar el mismo código para distintos tipos sin perder tipo seguro.

Ejemplo template función:

template <typename T>

T sumar(T a, T b) {

return a + b;

}

int main() {

std::cout << sumar(5, 3) << std::endl; // Suma de enteros

std::cout << sumar(5.5, 3.2) << std::endl; // Suma de flotantes

}

Ejemplo template clase:

template <typename T>

class Caja {

T valor;

public:

Caja(T v) : valor(v) {}

T obtenerValor() { return valor; }

};

int main() {

Caja<int> cajaInt(10);

Caja<double> cajaDouble(3.14);

std::cout << cajaInt.obtenerValor() << std::endl; // 10

std::cout << cajaDouble.obtenerValor() << std::endl; // 3.14

}

1. **Functor (objeto función) en C++**

Un functor es un objeto de una clase que sobrecarga el operador (), lo que le permite ser invocado como si fuera una función.

Relación con un predicado:

Un predicado es un tipo de functor que retorna un valor booleano (verdadero o falso) y se utiliza para evaluar condiciones en algoritmos.

Ejemplo de un functor:

#include <iostream>

using namespace std;

class Multiplicador {

public:

int operator()(int a, int b) {

return a \* b; // Multiplica dos números

}

};

int main() {

Multiplicador multiplicar;

cout << multiplicar(4, 5) << endl; // Llama al functor como función (4 \* 5)

}

Ejemplo de predicado en un algoritmo estándar:

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

class EsPar {

public:

bool operator()(int x) {

return x % 2 == 0; // Predicado: verifica si un número es par

}

};

int main() {

vector<int> numeros = {1, 2, 3, 4, 5, 6};

EsPar esPar;

auto it = find\_if(numeros.begin(), numeros.end(), esPar); // Encuentra el primer número par

if (it != numeros.end())

cout << "Primer par: " << \*it << endl;

}