Polymorphism

Ejercicio 00: Polimorfismo

- **Directorio de entrega:** ex00/
- **Archivos a entregar:** Makefile, main.cpp, *.cpp, *.h o *.hpp
- **Funciones prohibidas:** Ninguna
- **Requisito:** Proporcionar las pruebas más completas posibles.

Cada clase debe tener constructores y destructores que muestren mensajes específicos (diferentes para cada clase).

Descripción del ejercicio

Se trata de implementar el concepto de **polimorfismo** mediante clases en C++. Para ello, se debe crear una jerarquía de clases que representen animales y que sean capaces de emitir un sonido particular.

1. **Crear una clase base Animal** con el siguiente atributo protegido:

   ```
   std::string type;
   ```

   a. Esta clase representará un animal genérico.
   b. Se puede dejar su atributo type vacío o con un valor predeterminado.
2. **Crear dos clases derivadas: Dog y Cat** que hereden de Animal.
   a. Cada una debe inicializar su atributo type con su nombre correspondiente:
      i. Dog: "Dog"
      ii. Cat: "Cat"
3. **Implementar una función miembro makeSound() en cada clase.**
   a. Dog imprimirá un sonido de perro, por ejemplo: "Woof!".
   b. Cat imprimirá un sonido de gato, por ejemplo: "Meow!".
   c. Animal puede imprimir un mensaje genérico o no hacer nada.
4. **Probar el código con el siguiente main(),** asegurándose de que las funciones se comportan como se espera:

   ```
   int main()
   {
      const Animal* meta = new Animal();
      const Animal* j = new Dog();
      const Animal* i = new Cat();
   ```

```
    std::cout << j->getType() << " " << std::endl;
    std::cout << i->getType() << " " << std::endl;
    i->makeSound(); // Debe imprimir el sonido del gato
    j->makeSound(); // Debe imprimir el sonido del perro
    meta->makeSound(); // Dependerá de la implementación en Animal

    delete meta;
    delete j;
    delete i;
    return 0;
}
```

5. **Implementar clases "incorrectas" (WrongAnimal y WrongCat).**
    a. WrongCat debe heredar de WrongAnimal.
    b. Si reemplazamos Animal y Cat por WrongAnimal y WrongCat en main(), WrongCat debería imprimir el sonido de WrongAnimal, lo que ilustra cómo el polimorfismo **no funciona** correctamente sin métodos virtuales.

Cómo afrontar el ejercicio en C++98

1. **Definir Animal correctamente:**
    a. Declarar un **constructor** y un **destructor virtual** (esto es clave para el polimorfismo).
    b. Implementar makeSound() de forma genérica.
    c. Crear un **getter** para obtener el type.
2. **Crear Dog y Cat:**
    a. **Heredar** correctamente de Animal.
    b. Sobrescribir makeSound() para emitir sonidos específicos.
    c. Configurar el type en el constructor.
3. **Usar punteros a Animal para probar polimorfismo**
    a. Se crean objetos Dog y Cat usando punteros a Animal.
    b. Se invoca makeSound() para verificar que se ejecuta la versión correcta.
4. **Implementar WrongAnimal y WrongCat sin virtual en makeSound()**
    a. Esto demostrará cómo, sin el uso de métodos virtuales, el tipo base WrongAnimal **oculta** la implementación en WrongCat.
5. **Escribir un Makefile para compilarlo correctamente**

Este ejercicio refuerza la importancia del **polimorfismo con métodos virtuales** y muestra qué sucede cuando estos no se utilizan.

Explicación del código

Este código implementa **polimorfismo en C++98** usando una jerarquía de clases (Animal, Dog, Cat) y una versión incorrecta (WrongAnimal, WrongCat) para demostrar cómo funciona la herencia con y sin métodos virtuales.

1️⃣ Explicación de Animal.hpp y Animal.cpp

Animal.hpp

- Es la clase base de la jerarquía.
- Tiene un atributo protegido _type para almacenar el tipo de animal.
- Sus métodos principales son:
  - **makeSound()** (virtual): Define un sonido genérico para animales.
  - **getType() y setType()**: Getter y setter para _type.
  - **Constructores y operador de asignación (operator=)**.

Animal.cpp

- Implementa la clase Animal.
- Contiene los constructores y el operador de asignación.
- **El destructor es virtual**, permitiendo la destrucción correcta de objetos derivados.

Ejemplo de ejecución

```
const Animal* a = new Animal();
a->makeSound();   // Output: Animal sound 🔊
delete a;         // Output: Animal destructor called 🔥
```

2️⃣ Explicación de Cat.hpp y Cat.cpp

Cat.hpp

- Cat hereda de Animal.
- Sobreescribe makeSound() para hacer "Meow meow 🔊".

- Usa using Animal::operator= para reutilizar la sobrecarga del operador de asignación de Animal.

Cat.cpp

- **El constructor Cat() inicializa _type con "Cat"**.
- **El destructor** muestra un mensaje cuando el objeto se destruye.
- **makeSound() está sobreescrito** para imprimir "Meow meow 🔊" en lugar de "Animal sound 🔊".

Ejemplo de ejecución

```
const Animal* c = new Cat();
c->makeSound();  // Output: Cat does: Meow meow 🔊
delete c;       // Output: Cat destructor called 🐱 -> Animal destructor called 🔥
```

3️⃣ Explicación de Dog.hpp y Dog.cpp

Dog.hpp

- Dog hereda de Animal.
- Sobreescribe makeSound() para hacer "Woof woof 🔊".
- Usa using Animal::operator= para heredar la sobrecarga del operador de asignación.

Dog.cpp

- **El constructor Dog() inicializa _type con "Dog"**.
- **El destructor** muestra un mensaje cuando el objeto se destruye.
- **makeSound() está sobreescrito** para imprimir "Woof woof 🔊" en lugar de "Animal sound 🔊".

Ejemplo de ejecución

```
const Animal* d = new Dog();
d->makeSound();  // Output: Dog does: Woof woof 🔊
delete d;       // Output: Dog destructor called 🦖 -> Animal destructor called 🔥
```

4️⃣ Explicación de main.cpp

Parte 1: Pruebas con Animal, Dog y Cat

```
const Animal* meta = new Animal();
const Animal* j = new Dog();
const Animal* i = new Cat();
```

- Se crean objetos de Animal, Dog y Cat pero referenciados como Animal*.

```
cout << j->getType() << endl;  // Output: Dog
cout << i->getType() << endl;  // Output: Cat
```

- Se verifica que los objetos Dog y Cat tienen sus tipos correctos.

```
i->makeSound();  // Output: Cat does: Meow meow 🔊
j->makeSound();  // Output: Dog does: Woof woof 🔊
meta->makeSound();  // Output: Animal sound 🔊
```

- Se llama a makeSound() en cada objeto para probar el **polimorfismo dinámico**.

```
delete meta;
delete j;
delete i;
```

- **Al eliminar j (Dog) y i (Cat), primero se ejecutan sus destructores y luego el de Animal**, demostrando la importancia del destructor virtual.

Parte 2: Pruebas con WrongAnimal y WrongCat

```
WrongAnimal *wrongAnimal = new WrongAnimal;
WrongAnimal *wrongCat = new WrongCat;
```

- **Nota:** WrongCat hereda de WrongAnimal **sin virtual en makeSound()**, lo que genera un comportamiento inesperado.

```
wrongAnimal->makeSound();  // Output: WrongAnimal sound 🐉
wrongCat->makeSound();     // Output: WrongAnimal sound 🐉 (¡incorrecto!)
```

- Aquí wrongCat->makeSound() no usa la versión sobreescrita en WrongCat, sino la de WrongAnimal, demostrando la ausencia de polimorfismo dinámico.

```
delete wrongAnimal;
delete wrongCat;
```

- Al eliminar wrongCat, solo se ejecuta su destructor base, lo que puede causar fugas de memoria si tuviera atributos propios.

5 Diferencia clave entre Animal y WrongAnimal

Característica	Animal	WrongAnimal
Usa métodos virtuales	✅ Sí	❌ No
Polimorfismo dinámico	✅ Funciona	❌ No funciona
Destructor virtual	✅ Sí	❌ No
makeSound() correcto	✅ Sí	❌ No

CONCLUSIÓN:

- **Los métodos virtuales en C++ permiten que el programa elija la función correcta en tiempo de ejecución.**
- **Si makeSound() no es virtual, C++ usa la versión de la clase base incluso si el objeto es de una clase derivada.**

6 Resumen final

1. **El código demuestra polimorfismo usando una clase base Animal y clases derivadas Dog y Cat.**
2. **Gracias al uso de virtual, makeSound() se ejecuta en la clase correcta en tiempo de ejecución.**
3. **La versión incorrecta (WrongAnimal y WrongCat) muestra qué sucede si makeSound() no es virtual: el polimorfismo falla.**

4. **El destructor virtual evita fugas de memoria al eliminar objetos derivados a través de punteros a la base.**

Este ejercicio es un buen ejemplo de cómo y por qué usar polimorfismo en C++98 con métodos y destructores virtuales.