Clase Abstracta

Explicación del Ejercicio 02: Clase Abstracta

El objetivo de este ejercicio es modificar la clase Animal para que no pueda ser instanciada directamente. Esto se debe a que, en la realidad, un "animal" como concepto general no emite un sonido específico, sino que son sus subclases (como Dog o Cat) las que deberían definir su propio comportamiento.

Para lograr esto, se debe convertir la clase Animal en una **clase abstracta**. En C++, una clase abstracta es aquella que tiene **al menos un método virtual puro**, lo que significa que al menos una de sus funciones debe declararse sin implementación en la clase base. Esto se hace utilizando la sintaxis = 0.

El programa debe seguir funcionando igual que antes, lo que implica que las clases derivadas (Dog, Cat, etc.) deben seguir implementando los métodos de manera adecuada.

Cómo Enfrentar el Ejercicio en C++98

1. **Convertir Animal en una clase abstracta**
    a. Declarar al menos una función virtual pura, como makeSound().
    b. Esto impedirá que se puedan crear objetos de Animal directamente.
2. **Mantener la funcionalidad de las clases derivadas**
    a. Las clases Dog y Cat deben seguir heredando de Animal.
    b. Deben **implementar obligatoriamente** el método makeSound() ya que su clase base ahora lo exige.
3. **Opcional: Cambiar el nombre de la clase**
    a. Se sugiere renombrar Animal como AAnimal para reflejar que es abstracta.

El código

Este código implementa un sistema de clases en C++98 para representar una jerarquía de **animales**, con clases específicas para **gatos (Cat) y perros (Dog)**, y una clase Brain que modela un cerebro con ideas.

El **objetivo** principal es gestionar correctamente la memoria dinámica al manipular objetos de estas clases mediante **polimorfismo y clases abstractas**, asegurando una **correcta copia profunda** de los atributos dinámicos.

1 ¿Qué hace este código?

1. **Define una clase base abstracta Animal**, que:
   a. No puede instanciarse directamente (por tener métodos virtuales puros).
   b. Posee un atributo _type para diferenciar animales.
   c. Obliga a implementar makeSound(), getBrain() y sobrecarga del operador = en clases derivadas.
2. **Crea clases Dog y Cat**, que:
   a. Heredan de Animal.
   b. Poseen un atributo Brain* _brain que se gestiona dinámicamente.
   c. Implementan makeSound(), getBrain() y manejan copias profundas en el operador =.
3. **Crea la clase Brain**, que:
   a. Posee un array _ideas[100] donde almacena ideas.
   b. Tiene métodos para añadir y obtener ideas.
   c. Implementa un constructor, destructor y sobrecarga del operador = para manejar correctamente la copia de cerebros.
4. **En main()**:
   a. Se crea un arreglo de Animal* con COUNT = 4, alternando Dog y Cat.
   b. Se asignan ideas al Brain de animals[3].
   c. Se copia animals[3] en animals[2] con copia profunda.
   d. Se agregan nuevas ideas para comprobar que cada cerebro es independiente.
   e. Se eliminan los animales asegurando la liberación de memoria.

2 Explicación Paso a Paso

Clase Animal (Abstracta)

Esta clase **define la estructura común** para todos los animales.

```
class Animal {
public:
  virtual Animal& operator=(const Animal& src) = 0; // Método virtual puro = clase abstracta
  virtual ~Animal();  // Destructor virtual
  const std::string& getType() const;  // Devuelve el tipo del animal
  void setType(const std::string& _type);  // Permite cambiar el tipo
  virtual void makeSound() const = 0;  // Método virtual puro (debe implementarse en subclases)
  virtual Brain* getBrain() const = 0;  // Cada animal tiene un cerebro
protected:
  std::string _type;  // Tipo de animal
```

```
};
```

- makeSound() = 0: Obliga a que cada subclase tenga su propio sonido.
- getBrain() = 0: Obliga a implementar cómo acceder al cerebro.
- operator=() es puro, asegurando que todas las clases derivadas implementen **copias profundas**.
- ~Animal() es **virtual** para garantizar que los destructores de las subclases sean llamados correctamente.

Clase Brain (Modelo del cerebro de un animal)

```
class Brain {
private:
  unsigned int size;
  std::string _ideas[100];  // Almacena hasta 100 ideas
public:
  Brain();  // Constructor
  Brain(const Brain& src);  // Constructor de copia
  Brain& operator=(const Brain& src);  // Copia profunda en el operador =
  ~Brain();  // Destructor
  void addIdea(std::string idea);  // Agrega una idea
  const std::string& getIdea(unsigned int i) const;  // Obtiene una idea
};
```

- addIdea(): Añade una idea si hay espacio.
- getIdea(): Devuelve la idea en un índice específico.
- **Importante:** operator= copia todas las ideas correctamente.

Clases Cat y Dog (Heredan de Animal)

Cada una:

- Posee un **puntero dinámico a Brain**.
- Implementa makeSound(), getBrain() y **una sobrecarga del operador =** con **copia profunda**.
- Libera el Brain en el destructor.

Ejemplo para Cat:

```
class Cat : public Animal {
private:
  Brain* _brain;
public:
  Cat();
  Cat(const Cat& src);
  Cat& operator=(const Cat& src);  // Copia profunda
  Animal& operator=(const Animal& src);  // Copia profunda desde otro `Animal`
  ~Cat();
  void makeSound() const;
  Brain* getBrain() const;
};
```

Implementación en main.cpp

Creación de un grupo de animales

```
Animal* animals[COUNT];  // Arreglo de punteros a `Animal`
for (int i = 0; i < COUNT; i++) {
  if (i % 2 == 0)
    animals[i] = new Dog();
  else
    animals[i] = new Cat();
}
```

Alterna entre Dog y Cat.

Asignar ideas a un animal

```
brain = animals[3]->getBrain();
brain->addIdea("I hate the humans");
brain->addIdea("I hate the cats");
brain->addIdea("I hate the dogs");
brain->addIdea("I hate the animals in general");
```

Modifica el Brain del animal en animals[3].

Copia profunda de animals[3] en animals[2]

```
*(animals[2]) = *(animals[3]);  // Copia el tipo y el cerebro
*animals[2]->getBrain() = *animals[3]->getBrain();  // Copia las ideas
```

Garantiza que animals[2] tenga **una copia independiente** del cerebro de animals[3].

Asignar ideas distintas a animals[2] y animals[3]

```
animals[2]->getBrain()->addIdea("The last of the n. 2");
animals[3]->getBrain()->addIdea("The last of the n. 3");
```

Verifica que cada cerebro es único.

Liberación de memoria

```
for (int i = 0; i < COUNT; i++) {
   delete animals[i];
}
```

Cada delete llama al **destructor virtual de Animal**, asegurando que los destructores de Cat y Dog se ejecuten, eliminando Brain correctamente.

3 Resumen Final

✅ **Usa una clase abstracta (Animal)** para garantizar que solo se creen instancias de Dog o Cat.

✅ **Cada animal tiene un Brain dinámico**, lo que requiere **gestión de memoria manual**.

✅ **Se implementan copias profundas** en el operador =, asegurando que cada Brain sea independiente.

✅ **El programa crea, copia y destruye animales dinámicamente**, evitando fugas de memoria.

Este código es un excelente **ejemplo de polimorfismo, clases abstractas, memoria dinámica y copias profundas en C++98**. 🚀