I don’t want to set the world on fire

Aquí tienes una explicación detallada y sencilla del ejercicio propuesto, junto con una guía de cómo afrontarlo en C++98.

◇ Explicación del ejercicio

Objetivo:

Se trata de implementar un sistema de clases en C++98 para modelar animales, específicamente **perros (Dog) y gatos (Cat)**, con un componente adicional de memoria llamado **Brain** (cerebro). El objetivo principal es gestionar correctamente la memoria, asegurándose de que **no haya fugas (memory leaks)** y de que las copias de objetos sean **copias profundas (deep copies)** en lugar de superficiales (shallow copies).

Puntos clave del ejercicio:

1. **Crear una clase Brain** que contenga un **array de 100 std::string** llamado ideas.
2. **Agregar un puntero a Brain** en las clases Dog y Cat como un atributo privado.
3. **Gestionar la memoria dinámicamente**:
    a. Cuando se crea un Dog o Cat, debe asignar memoria para un nuevo Brain (new Brain()).
    b. Cuando se destruye un Dog o Cat, debe liberar la memoria asignada (delete Brain).
4. **Evitar fugas de memoria** asegurando que todas las instancias creadas dinámicamente se destruyen correctamente.
5. **Implementar copias profundas**:
    a. Si copiamos un Dog o Cat, su Brain debe ser copiado completamente en una nueva instancia, no solo apuntar a la memoria del original.
6. **Ejecutar pruebas adecuadas**:
    a. Se debe crear un **array de Animal** que contenga mitad Dog y mitad Cat.
    b. Se debe recorrer este array y eliminar los objetos correctamente (eliminándolos como Animal para verificar que los destructores se ejecuten correctamente en la jerarquía).
    c. Se debe comprobar que las copias de Dog y Cat son **copias profundas**.

◇ ¿Cómo afrontarlo en C++98?

1️⃣ Definir la clase Brain

- Contendrá un array de 100 std::string llamado ideas.
- Implementar un constructor y un destructor que muestren mensajes para verificar su ejecución.

2️⃣ Crear la clase base Animal

- Debe ser una **clase abstracta** con un método virtual makeSound() para que cada animal tenga su propio sonido.

3️⃣ Implementar las clases Dog y Cat

- Deben **heredar de Animal**.
- Tendrán un puntero privado a un Brain.
- En el constructor deben asignar memoria (new Brain()).
- En el destructor deben liberar la memoria (delete Brain).
- Implementar correctamente el **constructor de copia y el operador de asignación** para hacer copias profundas.

4️⃣ Implementar el programa principal (main)

- Crear dinámicamente dos instancias (Dog y Cat), eliminarlas y verificar que no hay fugas de memoria.
- Crear un array de Animal* que contenga mitad Dog y mitad Cat.
- Recorrer el array y eliminar todos los objetos correctamente.
- Probar la copia profunda.

5️⃣ Comprobaciones finales

- **Mensajes en constructores y destructores**: Deben imprimirse en el orden correcto.
- **Prueba de fugas de memoria**: Puede usarse valgrind (en Linux/Mac) para comprobar si hay leaks.
- **Verificación de copias profundas**: Cambiar el contenido de ideas en un objeto copiado y verificar que el original no cambia.

◇ El código:

Explicación clara y detallada del código

Este código implementa una jerarquía de clases en C++98 para modelar un sistema de **animales**, específicamente **perros (Dog) y gatos (Cat)**, con un sistema de memoria a través de la clase **Brain (Cerebro)**. Se enfoca en la correcta **gestión de memoria dinámica**, **copias profundas** y **polimorfismo**.

◇ Desglose de las clases y su funcionamiento

📌 Clase Animal (Base)

📁 *Animal.hpp* y *Animal.cpp*

🔹 **Características principales**:

- Clase base abstracta con un método puro getBrain(), lo que significa que no se puede instanciar directamente.
- Contiene un atributo std::string type para almacenar el tipo de animal.
- Implementa:
  - **Constructor por defecto**
  - **Constructor con parámetro** para establecer el tipo.
  - **Constructor de copia**.
  - **Destructor virtual** (muy importante para asegurar la eliminación correcta en clases derivadas).
  - **Operador de asignación**.
  - **Método makeSound()**, que imprime un mensaje genérico.
  - **Método getType() y setType()** para obtener y establecer el tipo de animal.

🔹 **Explicación de la memoria**:

- **El destructor es virtual**, lo que permite que cuando eliminemos un Animal*, se llame correctamente al destructor de la subclase correspondiente (Dog o Cat).

📌 Clase Brain (Cerebro)

📁 *Brain.hpp* y *Brain.cpp*

🔹 **Características principales**:

- Contiene un **array de 100 std::string** (ideas) donde cada instancia de Brain puede almacenar pensamientos.
- Un **atributo count** que controla cuántas ideas han sido almacenadas.
- Implementa:
  - **Constructor por defecto** que inicializa un cerebro vacío.
  - **Constructor de copia** que copia todas las ideas (evitando una copia superficial).
  - **Destructor** para liberar correctamente la memoria.
  - **Operador de asignación =** que copia los datos profundamente.
  - **Métodos addIdea() y getIdea()** para agregar y recuperar ideas.

🔹 **Explicación de la memoria**:

- La memoria es **administrada internamente**, pero cuando un Brain es copiado, todas sus ideas se duplican correctamente para evitar que dos objetos compartan la misma memoria (deep copy).

📌 Clase Cat (Gato)

📁 *Cat.hpp* y *Cat.cpp*

🔹 **Características principales**:

- Hereda de Animal y establece el tipo "Cat".
- Contiene un **puntero a Brain**, que se maneja dinámicamente (new Brain(); en el constructor y delete this->brain; en el destructor).
- Implementa:
  - **Constructor por defecto**: Crea un Brain nuevo.
  - **Constructor de copia**: Crea un Brain nuevo y copia el contenido del original (deep copy).
  - **Destructor**: Libera la memoria del Brain.
  - **Operador de asignación =**:
    - Usa dynamic_cast para asegurarse de que la asignación es entre objetos Cat.
    - Copia el tipo de animal y los datos del Brain profundamente.
  - **Método makeSound()** que imprime "Meow 😺".
  - **Método getBrain()** que devuelve el puntero a Brain.

🔹 **Explicación de la memoria**:

- Cada Cat tiene su propio Brain, asegurando que cuando se copian o asignan objetos, se realiza una **copia profunda**.

📌 Clase Dog (Perro)

📁 *Dog.hpp* y *Dog.cpp*

🔹 **Características principales**:

- Hereda de Animal y establece el tipo "Dog".
- Contiene un **puntero a Brain**, que se maneja dinámicamente (new Brain(); en el constructor y delete this->brain; en el destructor).
- Implementa:
  - **Constructor por defecto**: Crea un Brain nuevo.
  - **Constructor de copia**: Crea un Brain nuevo y copia el contenido del original (deep copy).
  - **Destructor**: Libera la memoria del Brain.
  - **Operador de asignación =**:
    - Usa dynamic_cast para asegurarse de que la asignación es entre objetos Dog.

        - Copia el tipo de animal y los datos del Brain profundamente.
    - **Método makeSound()** que imprime "Woof 🐶".
    - **Método getBrain()** que devuelve el puntero a Brain.

🔹 **Explicación de la memoria**:

- Igual que en Cat, cada Dog tiene su propio Brain, garantizando **copias profundas**.

📌 Función main()

📁 *main.cpp*

🔹 **Objetivo principal**:

- **Crear y gestionar memoria para un grupo de animales (Dog y Cat)**.
- **Verificar que los destructores se ejecutan correctamente** sin fugas de memoria.

🔹 **Explicación del código**:

1. Se declara un **array de punteros a Animal** (animals[COUNT]), donde COUNT = 4.
2. Se llena el array con **2 Dog y 2 Cat**, alternándolos.
3. Se imprimen mensajes de creación para verificar que los constructores se ejecutan correctamente.
4. **Se eliminan los objetos recorriendo el array**, verificando que los destructores se ejecutan correctamente.
5. Gracias al **polimorfismo y destructores virtuales**, se eliminan correctamente los Brain y Animal sin fugas de memoria.

◇ Resumen y puntos clave

✅ **Uso correcto de herencia y polimorfismo**:

- Animal es una clase base abstracta con makeSound() y getBrain() como métodos virtuales.
- Cat y Dog heredan de Animal e implementan su propio comportamiento.

✅ **Gestión adecuada de la memoria dinámica**:

- Cada Dog y Cat tiene un Brain que se gestiona dinámicamente.
- Se crean con new Brain(); y se eliminan con delete brain; en los destructores.
- El operador de asignación y el constructor de copia manejan **copias profundas**, evitando problemas con punteros compartidos.

✅ **Ejemplo de uso en main()**:

- Se crean 4 animales (Dog y Cat intercalados).

- Se recorren y eliminan correctamente, verificando que los destructores se ejecutan en el orden esperado.

✅ **Evita fugas de memoria**:

- Uso correcto de **destructores virtuales** y delete en cada instancia creada con new.
- Se pueden verificar fugas con herramientas como valgrind.

◇ Conclusión

Este código es un excelente ejemplo de:

- **Programación orientada a objetos (POO) en C++98**.
- **Uso de memoria dinámica y copias profundas**.
- **Aplicación de polimorfismo y clases abstractas**.
- **Buenas prácticas en la gestión de recursos**.

Al ejecutarlo, se verá claramente cómo los **constructores y destructores** trabajan correctamente para evitar fugas de memoria, asegurando que los objetos Dog y Cat sean completamente eliminados cuando ya no se necesiten. 🚀