Interface & recap

Aquí tienes una explicación clara y detallada del ejercicio, junto con una guía para afrontarlo en C++98.

Explicación del ejercicio

Este ejercicio tiene como objetivo reforzar el concepto de **interfaces** en C++98 utilizando **clases abstractas puras**. La idea es simular un sistema de objetos en el que existan diferentes tipos de "Materias" (hechizos o habilidades), personajes que puedan equiparlas y usarlas, y una fuente de Materias capaz de crearlas cuando se necesiten.

El ejercicio se divide en varias partes:

1. **Clase base abstracta AMateria**
    a. Representa una "Materia mágica".
    b. Tiene un **tipo** (ej., "ice" o "cure").
    c. Debe proporcionar una función **clone()** para crear copias.
    d. Debe tener un método **use(ICharacter& target)** que realice una acción específica dependiendo del tipo de Materia.
2. **Clases derivadas Ice y Cure**
    a. Son implementaciones concretas de AMateria.
    b. La función **clone()** debe devolver una nueva instancia del mismo tipo.
    c. La función **use()** debe imprimir el mensaje correspondiente.
3. **Interfaz ICharacter y clase concreta Character**
    a. **ICharacter** define cómo debe comportarse un personaje:
        i. Puede equipar y desequipar Materias.
        ii. Puede usar Materias.
    b. **Character** implementa esta interfaz:
        i. Tiene un **inventario** con 4 espacios para Materias.
        ii. Puede equipar una Materia en la primera ranura vacía.
        iii. Puede usar Materias, llamando a use() de AMateria.
        iv. No debe eliminar Materias al desequiparlas, solo dejar de referenciarlas.
        v. Implementa una **copia profunda** al duplicar un personaje, asegurando que se clonen sus Materias.
4. **Interfaz IMateriaSource y clase concreta MateriaSource**
    a. **IMateriaSource** define cómo una fuente de Materias debe comportarse:
        i. Puede aprender nuevas Materias.
        ii. Puede crear Materias a partir de su base de datos.
    b. **MateriaSource** almacena hasta 4 Materias "modelo".
    c. Cuando se solicite una Materia con createMateria(), devuelve una copia de una Materia aprendida.

Cómo afrontarlo en C++98

Dado que C++98 no tiene **interfaces** como en Java o C#, se usan **clases abstractas** (con al menos un método virtual puro). Aquí algunos consejos para estructurar la solución:

1. **Definir correctamente las clases base con métodos virtuales puros**
   a. Usa virtual ~Clase() para evitar fugas de memoria.
   b. Declara los métodos que deben ser sobrecargados como virtual y usa = 0 para los puros.
2. **Implementar la clase AMateria correctamente**
   a. Almacena un std::string type en protected.
   b. Implementa getType() para devolver el tipo.
   c. Implementa el constructor y un destructor virtual.
3. **Crear Ice y Cure correctamente**
   a. Define su clone() para devolver new Ice(*this) o new Cure(*this).
   b. Implementa use() con el std::cout correspondiente.
4. **Desarrollar ICharacter y Character**
   a. Almacenar el nombre y un array de 4 punteros a AMateria*.
   b. Implementar equip() para agregar una Materia en la primera ranura vacía.
   c. Implementar unequip() sin liberar memoria, solo eliminando la referencia.
   d. En use(), llamar al método use() de la Materia correspondiente.
   e. Manejar correctamente la **copia profunda** en el constructor de copia y operador =.
5. **Implementar IMateriaSource y MateriaSource**
   a. Guardar hasta 4 Materias en un array.
   b. learnMateria() almacena una copia de la Materia dada.
   c. createMateria() devuelve una nueva instancia si la Materia ha sido aprendida.
6. **Gestión de memoria**
   a. No olvidar delete en destructores.
   b. Evitar **memory leaks** asegurándose de borrar Materias correctamente.
   c. Controlar punteros al hacer unequip() para evitar accesos inválidos.

Con esta estructura bien organizada, puedes abordar el ejercicio de manera ordenada y eficiente en C++98.

El código

Este código implementa un sistema de "materias" inspirado en el sistema de magia de **Final Fantasy VII** en C++98, utilizando el concepto de interfaces mediante clases abstractas puras.

Voy a explicarte de manera clara y detallada qué hace cada parte del código y cómo funciona.

1. Concepto General del Código

Este programa permite:

- **Crear "materias" (objetos mágicos)** con clases derivadas (Ice, Cure) que heredan de una clase base AMateria.
- **Almacenar "materias"** en un personaje (Character) con un inventario de 4 espacios.
- **Usar las materias en otros personajes** para ejecutar efectos específicos.
- **Implementar un "proveedor de materias"** (MateriaSource), que aprende nuevas materias y las clona bajo demanda.

2. Explicación por Componentes

(a) Clase AMateria (Clase Base de las Materias)

AMateria.hpp

Es una clase abstracta pura (una interfaz en C++98), ya que contiene al menos una función virtual pura:

virtual AMateria* clone() const = 0;

AMateria.cpp

- Guarda el **tipo de materia** (_type).
- Implementa un **constructor** que inicializa _type.
- Tiene un **getter getType()** para obtener el tipo.
- No implementa clone() ni use() porque son funciones virtuales puras y deben ser definidas en clases derivadas.

(b) Clases Ice y Cure (Materias Concretas)

Ambas heredan de AMateria e implementan:

1. **clone()** → Devuelve una nueva instancia de sí misma (new Ice(*this) o new Cure(*this)).
2. **use(ICharacter&)** → Muestra un mensaje en consola indicando la acción realizada.

Ejemplo en Ice.cpp:

```
void Ice::use(ICharacter& target) {
   std::cout << "* shoots an ice bolt at " << target.getName() << " *" << std::endl;
}
```

- Si se usa una Ice, imprimirá: * shoots an ice bolt at bob *

(c) ICharacter (Interfaz de los Personajes)

```
class ICharacter {
public:
  virtual const std::string& getName() const = 0;
  virtual void equip(AMateria* m) = 0;
  virtual void unequip(int idx) = 0;
  virtual void use(int idx, ICharacter& target) = 0;
  virtual ~ICharacter() {}
};
```

- Define las funciones que un personaje debe tener.
- No implementa nada (solo es una guía para las clases derivadas).

(d) Clase Character (Personajes Jugables)

Inventario de 4 espacios para materias

```
AMateria* _inventory[4]; // Un personaje puede equipar hasta 4 materias.
int _inventorySize;
```

Equipar una materia (equip())

```
void Character::equip(AMateria* m) {
  if (_inventorySize < 4)
    _inventory[_inventorySize++] = m;
}
```

- **Si hay espacio en el inventario**, se añade la materia al primer slot disponible.
- **Si está lleno**, no hace nada.

Usar una materia (use())

```
void Character::use(int idx, ICharacter& target) {
  if (idx >= 0 && idx < _inventorySize)
    _inventory[idx]->use(target);
}
```

- **Si el índice es válido**, llama a use(target) de la materia.
- **Ejemplo de salida** si bob es el objetivo: * shoots an ice bolt at bob *
  * heals bob's wounds *

Eliminar materias (unequip())

```
void Character::unequip(int idx) {
  if (idx >= 0 && idx < _inventorySize) {
    idx++;
    for (int i = idx; i < _inventorySize; i++)
      _inventory[i - 1] = _inventory[i];
    _inventorySize--;
  }
}
```

- **No elimina la materia de la memoria**, solo la elimina del inventario.
- **Las materias "dejadas en el suelo" deben gestionarse manualmente.**

(e) IMateriaSource (Interfaz para Generadores de Materias)

```
class IMateriaSource {
public:
  virtual AMateria* createMateria(const std::string& type) = 0;
  virtual void learnMateria(AMateria*) = 0;
  virtual ~IMateriaSource() {}
};
```

- Define la funcionalidad de un "almacén de materias".

(f) MateriaSource (Generador de Materias)

Almacena hasta 4 materias "plantilla"

```
AMateria* _materias[4];
int _materiasSize;
```

- Guarda materias que pueden ser clonadas después.

Aprender nuevas materias (learnMateria())

```
void MateriaSource::learnMateria(AMateria* m) {
  if (_materiasSize < 4)
    _materias[_materiasSize++] = m;
}
```

- Guarda una materia para clonarla después.

Crear una materia (createMateria())

```
AMateria* MateriaSource::createMateria(const std::string& type) {
  for (int i = 0; i < 4 && _materias[i]; i++) {
    if (type == _materias[i]->getType())
      return _materias[i]->clone();
  }
  return NULL;
}
```

- **Busca una materia aprendida** con el mismo tipo y la clona.
- **Si no la encuentra, devuelve NULL.**

(g) main.cpp (Programa Principal)

```
int main() {
  IMateriaSource* src = new MateriaSource();
  src->learnMateria(new Ice());
  src->learnMateria(new Cure());

  ICharacter* me = new Character("me");

  AMateria* tmp;
  tmp = src->createMateria("ice");
  me->equip(tmp);
  tmp = src->createMateria("cure");
  me->equip(tmp);

  ICharacter* bob = new Character("bob");

  me->use(0, *bob);
  me->use(1, *bob);

  delete bob;
  delete me;
  delete src;

  return 0;
}
```

¿Qué sucede al ejecutar este código?

1. **Se crea un almacén de materias (MateriaSource) y se le enseñan Ice y Cure.**
2. **Se crea un personaje me y se le equipa con una Ice y una Cure.**
3. **Se crea un personaje bob.**

4. **me usa su Ice y Cure en bob, mostrando:** * shoots an ice bolt at bob *
   * heals bob's wounds *

5. **Los objetos creados se eliminan correctamente (evitando fugas de memoria).**

3. Resumen de cómo afrontarlo en C++98

1. **Define clases base abstractas** (AMateria, ICharacter, IMateriaSource) con métodos virtuales puros.
2. **Usa herencia** para implementar clases concretas (Ice, Cure, Character, MateriaSource).
3. **Maneja la memoria dinámicamente** con new y delete evitando fugas.
4. **Usa punteros para la polimorfia**, permitiendo trabajar con AMateria* en lugar de tipos concretos.
5. **Sigue buenas prácticas en el manejo de copias** (constructores de copia, operadores =).

Este código es un excelente ejercicio de programación orientada a objetos en C++98, combinando **polimorfismo, herencia y gestión de memoria dinámica.**