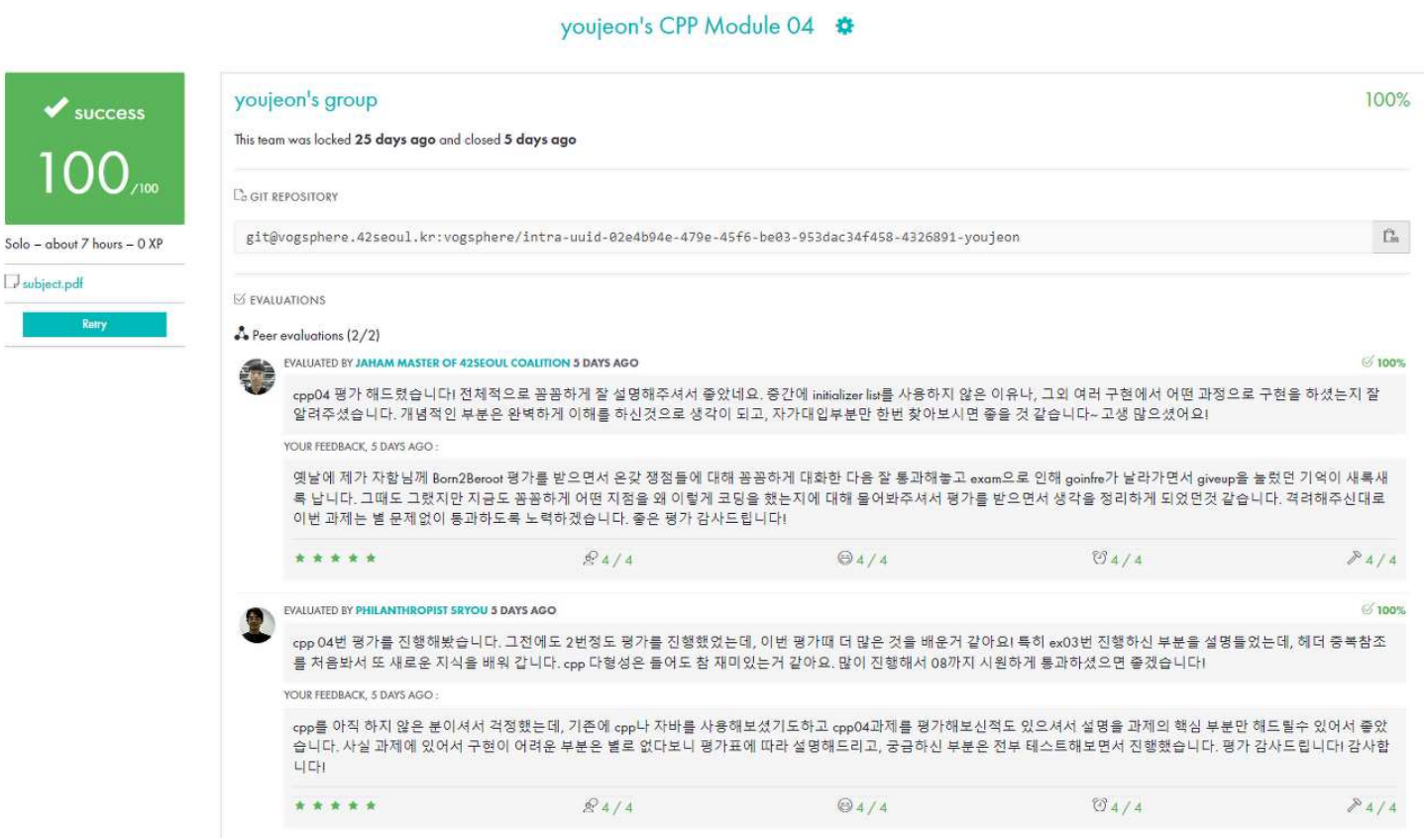
programación

[42Seoul] CPP Módulo 04 - Polimorfismo y clases abstractas

Aggrodonk 2022. 9. 10. 17:14

Introducción



La razón por la que no utilicé una lista de inicialización es para facilitar la comprensión del código a los evaluadores que no hicieron CPP, y me hicieron esta pregunta 3 veces.

Como se muestra en la primera página del tema, la tarea es implementar polimorfismo, clases abstractas e interfaces.

Como nota al margen, después del CPP 03, completé del 04 al 08 de una sola vez y luego recibí u na evaluación, pero después de recibirla, me di cuenta de que no era una muy buena idea. Duran te la evaluación, tuve un gran problema donde olvidé lo que estaba pensando cuando la hice y tu ve que preocuparme junto con el evaluador.

ex00

En cada ejercicio, debemos esforzarnos por ofrecer una prueba lo más completa posible.

Todos los constructores y destructores deben tener nombres diferentes según la clase.

Para empezar, implementamos solo una clase Animal. Es una clase que contiene solo una variabl e de tipo cadena como protector.

Cree una clase Perro y una clase Gato para que cada una tenga un tipo correspondiente a su nom bre. El perro se inicializa como 'Perro' y el Gato se inicializa como 'Gato'. Animal puede dejarse en blanco o configurarse como quieras.

Cada clase tiene makeSound() como función miembro. Puede hacer que genere adecuadamente de acuerdo con cada clase, y la clase Animal se implementa para que no se genere nada.

```
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(); //will output the cat sound!
        j->makeSound();
        meta->makeSound();
        ...
        return 0;
}
```

Implemente la prueba deseada adicionalmente en esta declaración principal. Para comprender c on precisión esta tarea, debe implementar WrongAnimal y WrongCat y crear una prueba (para qu e WrongCat emita el sonido de WrongAnimal).

```
#ifndef ANIMAL_HPP
# define ANIMAL_HPP

# include <iostream>
```

```
class Animal {
 protected:
        std::string type;
 public:
        Animal(void);
        Animal(const Animal& obj);
        Animal& operator=(const Animal& obj);
        virtual ~Animal(void);
        virtual void makeSound(void) const;
        std::string getType(void) const;
};

#endif


#ifndef WRONGANIMAL_HPP
# define WRONGANIMAL_HPP

# include <iostream>

class WrongAnimal {
 protected:
        std::string type;
 public:
        WrongAnimal(void);
        WrongAnimal(const WrongAnimal& obj);
        WrongAnimal& operator=(const WrongAnimal& obj);
        ~WrongAnimal(void);
        void makeSound(void) const;
        std::string getType(void) const;
};

#endif


int main(void)
{
        const Animal* meta = new Animal();
        const Animal* j = new Dog();
        const Animal* i = new Cat();
        const WrongAnimal* wrong = new WrongCat();

        std::cout << std::endl;
        std::cout << i->getType() << " " << std::endl;
        i->makeSound();
        std::cout << j->getType() << " " << std::endl;
        j->makeSound();
        std::cout << meta->getType() << " " << std::endl;
        meta->makeSound();
        std::cout << std::endl;

        std::cout << wrong->getType() << " " << std::endl;
        wrong->makeSound();

        std::cout << std::endl;
        delete meta;
        meta = NULL;
        delete j;
        j = NULL;
        delete i;
        i = NULL;
        delete wrong;
        wrong = NULL;
```

```
        return 0;
}
```

```
Cat
Cat say 'meow...'
Dog
Dog say 'Bark Bark!'
Animal
Animal say nothing, this message is something wrong

WrongCat
WrongAnimal say nothing, this message is something wrong
```

En C++, incluso un puntero a una clase principal puede contener la dirección de objeto de una cl ase secundaria. Esto se llama upcasting.

Sin embargo, dado que es un puntero a la clase principal, se produce un problema cuando se eje cuta la función original de la clase principal cuando se intenta ejecutar una función anulada (red efinida) en la clase secundaria.

Para resolver este problema, si adjunta la palabra clave virtual a la función de la clase principal p ara redefinirla en la clase secundaria, la función se convierte en una función virtual y la función d e la clase secundaria se puede ejecutar según lo previsto.

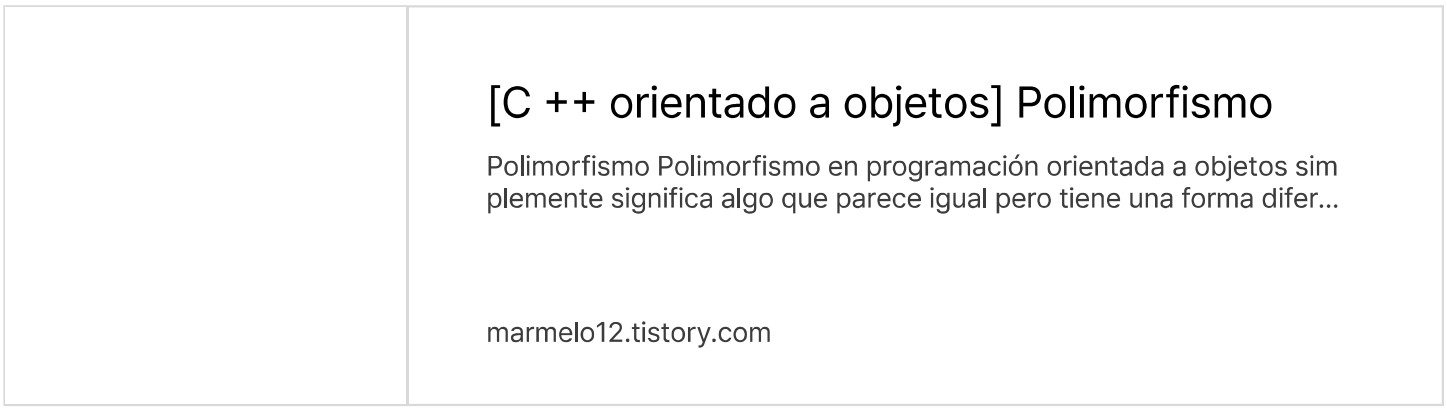
En esta tarea, debe usar la palabra clave virtual para hacer que la función makeSound y el destru ctor sean funciones virtuales para que la función Dog funcione según lo previsto cuando el punte ro del animal apunte a Dog. (El destructor no se menciona por separado en el tema, pero es nece sario para evitar pérdidas de memoria).

Como nota al margen, si una clase se declara como constante, está prohibido el uso de funciones sin const al final de las funciones miembro. Por lo tanto, para ejecutar correctamente la declaraci ón principal proporcionada, debe agregar const a makeSound().

En C++, solo se especifican las palabras clave y los métodos de operación de las funciones virtual es, y los principios operativos detallados se implementan según el compilador. En la mayoría de l os casos, se crea una tabla de funciones virtuales y, si una clase tiene una palabra clave virtual, la dirección apunta. a por la función se almacena por separado. Cuando se llama a una función, se ejecuta haciendo referencia a la dirección.

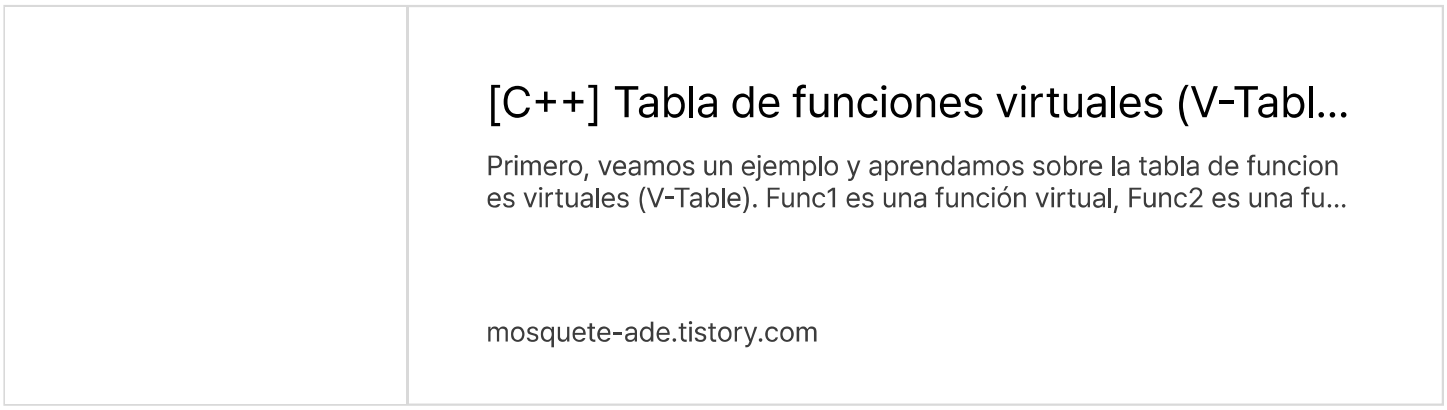
Naturalmente, no se debe abusar de la palabra clave virtual ya que consume más memoria y es más lenta, pero se debe usar cuando se espera anulación en clases secundarias.

https://marmelo12.tistory.com/m/285



http://www.tcpschool.com/cpp/cpp_polymorphism_virtual

https://musket-ade.tistory.com/entry/C-%EA%B0%80%EC%83%81%ED%95%A8%EC%88%98-%ED%85%8C%EC% 9D%B4%EB%B8%94-V-Mesa



https://stackoverflow.com/questions/56387509/this-argument-to-member-function-select-has-type-const-selectpa ram-but-fu

ex01

El constructor y destructor de cada clase deberían mostrar mensajes diferentes.

Crea una clase de cerebro. La clase tiene una variable miembro std::string [100] llamada ideas.

Dog y Cat tienen las clases correspondientes como variables miembro privadas.

Dog y Cat ejecutan new Brain() cuando se crean.

Perro y Gato ejecutan eliminación cuando son destruidos.

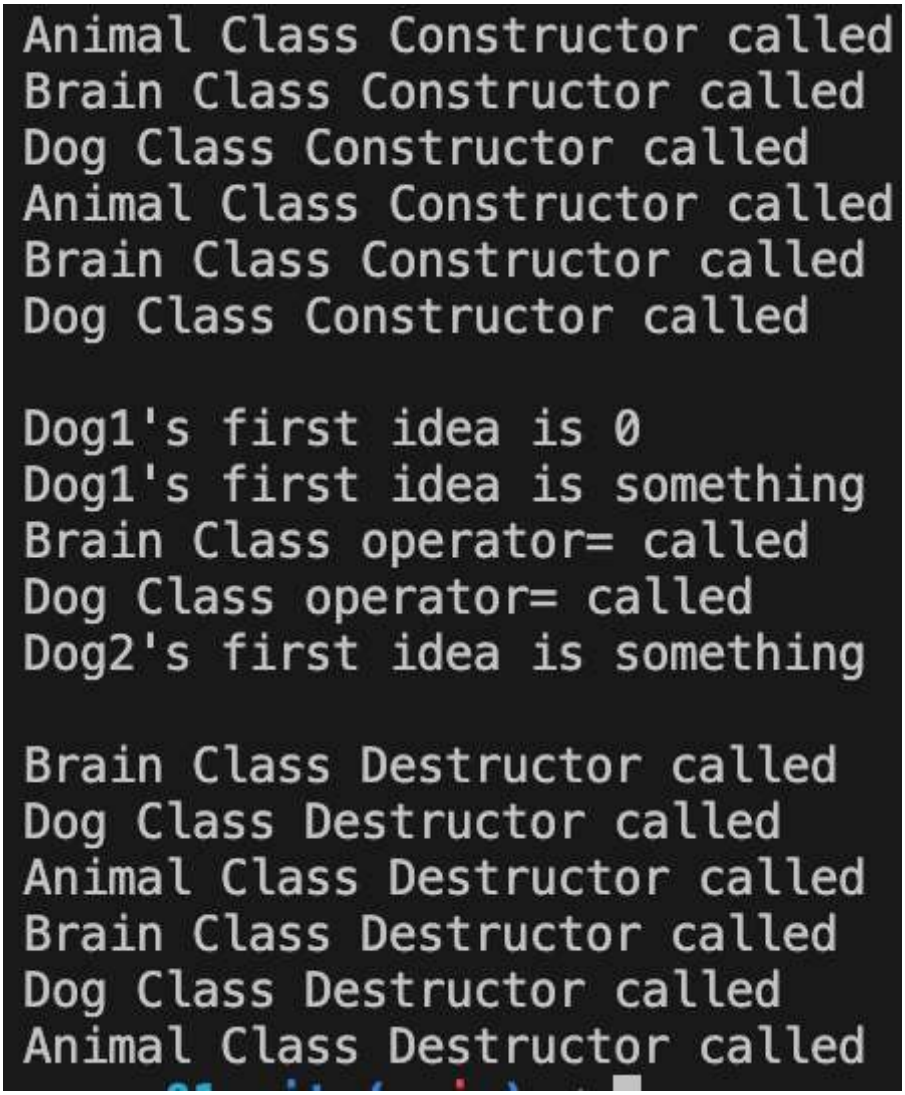
En la declaración principal, cree una matriz de animales, la mitad que contenga perros y la otra mitad que contenga gatos, y asegúrese de que los destructores se llamen en el orden correcto cu ando se cierre el programa.

Dog y Cat siempre realizan una copia profunda al copiar. Pruebe esto en la declaración principal.

Compruebe si hay pérdidas de memoria.

```
#ifndef BRAIN_HPP
# define BRAIN_HPP

# include <iostream>
# include <sstream>

class Brain {
 private:
```

```
        std::string ideas[100];
 public:
        Brain(void);
        Brain(const Brain& obj);
        Brain& operator=(const Brain& obj);
        ~Brain(void);
        std::string getIdeas(int n) const;
        void setIdeas(std::string idea, int n);
};

#endif


int main(void)
{
        std::string str;
        Animal *meta[10];

        for (size_t i = 0; i < 10; i++)
        {
                if (i % 2)
                {
                        meta[i] = new Dog();
                }
                else
                {
                        meta[i] = new Cat();
                }
        }
        std::cout << std::endl;

        for (size_t i = 0; i < 10; i++)
        {
                delete meta[i];
        }
        std::cout << std::endl;


        Dog *d = new Dog();
        Dog *d2 = new Dog();
        std::cout << std::endl;
        str = d->getBrain()->getIdeas(0);
        std::cout << "Dog1's first idea is "<< str << std::endl;

        d->getBrain()->setIdeas("something", 0);
        str = d->getBrain()->getIdeas(0);
        std::cout << "Dog1's first idea is "<< str << std::endl;

        *d2 = *d;
        str = d2->getBrain()->getIdeas(0);
        std::cout << "Dog2's first idea is "<< str << std::endl;

        std::cout << std::endl;
        delete d;
        d = NULL;
        delete d2;
        d2 = NULL;

        return 0;
}
```



No entendí por qué me pidieron que lo pusiera en un array y lo borrara. Pensé que entendería si miraba la tabla de evaluación, pero no había información relevante en la tabla de evaluación. Bu eno, supongo que es porque algo como esto no sucede simplemente durante uno o dos días.

Creo que es una tarea para ganar experiencia en el manejo de objetos de otras clases usando pu nteros dentro de una clase. Entonces, al manejar el objeto de la clase cerebral en un animal, pue des hacer que se cree al mismo tiempo que se crea y se destruya al mismo tiempo para que no h aya fugas.

En el tema, solo se enfatizó la copia profunda, pero en la tabla de evaluación, la copia profunda d ebe implementarse eliminando y renovando el Cerebro en Perro y Gato. Entonces, mientras solo miraba el tema y lo implementaba, pensé: 'El tamaño del cerebro es el mismo en 100 de todos m odos, así que solo necesito verificar que el contenido interno se haya cambiado con precisión a u na copia profunda, entonces, ¿por qué? ¿Hago una nueva asignación dinámica?' Lo implementé como pensaba y hubo una situación en la que no estaba sincronizado con la tabla de evaluación. Mencioné la situación a ambos evaluadores y, afortunadamente, ambos entendieron con frialda d y dijeron: 'Es cierto que es una copia profunda', así que pude seguir adelante...

La parte confusa aquí es *d2 = *d; Sin agregar el operador unario *, d2 = d; Al escribir de esta form a, el operador= de la clase a la que apuntaba el puntero no funcionaba y el operador= del punter o, es decir, la dirección a la que apuntaba, se cambiaba, provocando una explosión por doble lib eración en el borrado. Si lo piensas detenidamente, es una operación natural y un resultado natu ral, pero entonces ¿por qué no funciona? Deambulé así por un tiempo.

ex02

Para evitar los problemas causados por convertir a Animal en un objeto, conviértalo en una clase abstracta utilizando funciones virtuales puras.

```
class Animal {
 protected:
        std::string type;
 public:
        Animal(void);
        Animal(const Animal& obj);
        Animal& operator=(const Animal& obj);
        virtual ~Animal(void);
        virtual void makeSound(void) const = 0;
        std::string getType(void) const;
};
```

Literalmente, sólo necesitas hacer de la clase Animal una clase abstracta.

Si una función virtual creada en ex00 se adjunta a una función que "se espera" que sea redefinid a por una clase secundaria, una función virtual pura significa "una función que debe redefinirse". Además, una clase que contiene estas funciones virtuales puras se denomina clase abstracta.

La sintaxis es simplemente agregar = 0 al final de la función. Si lo crea de esta manera, no es nece sario definir la función por separado en esta clase. Si declara directamente una clase abstracta d e este tipo, se producirá un error de compilación, por lo que debe implementar y usar una clase s ecundaria.

http://www.tcpschool.com/cpp/cpp_polymorphism_abstract

tcpschool.com

ex03

La interfaz es un concepto que no existe en C++98 (y ciertamente no en C++20). Sin embargo, las clases abstractas puras generalmente se denominan interfaces. En este ejemplo, la tarea es impl ementar una interfaz utilizando estos módulos.

Cree una clase AMateria como se muestra a continuación:

```
class AMateria
{
        protected:
        [...]
        public:
        AMateria(std::string const & type);
        [...]
        std::string const & getType() const; //Returns the materia type
        virtual AMateria* clone() const = 0;
        virtual void use(ICharacter& target);
};
```

Cree dos tipos de Materias, o propiedades, que hereden la AMateria anterior. Hielo y cura. Las do s propiedades tienen un nombre llamado tipo y tienen el nombre de la propiedad en minúsculas (hielo y cura).

El clon de una función miembro de dos propiedades (que AMateria considera una función virtual pura) devuelve un nuevo objeto con las mismas propiedades.
use devuelve el siguiente resultado:

- Hielo: "* dispara un rayo de hielo a <nombre> *”
- Cura: "*cura las heridas de <nombre>*”

nombre es el nombre de la variable miembro que contiene el argumento ICharacter& target.

Cree una clase de interfaz ICharacter como se muestra a continuación:

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

```
        virtual void use(int idx, ICharacter& target) = 0;
};
```

La clase Personaje tiene cuatro inventarios vacíos y equipa Materia en los espacios 0 a 3. Si intent a agregar propiedades o usar/deshabilitar Materias faltantes cuando el inventario está lleno, no s e realizará ninguna acción (por supuesto, los errores están prohibidos).

La función de desequipar nunca elimina Materias (las materias que el personaje desequipa y no usa se manejarán automáticamente, pero no debería haber pérdidas de memoria).

La función de uso utiliza las Materias de la ranura [idx] y el objetivo se pasa como argumento par a el uso de AMateria.

Naturalmente, el inventario de un personaje debe poder contener cualquier Materia.

Una clase de carácter debe tener un constructor que tome un nombre como argumento y, por su puesto, las operaciones de asignación/copia deben ser copias profundas.

Durante la copia, se deben eliminar las propiedades que se tenían anteriormente y también se d ebe eliminar el destructor.

Implemente la clase de interfaz IMateriaSource como se muestra a continuación.

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

learnMateria copia las propiedades ingresadas como argumentos y las almacena en la memoria para su uso posterior.

De manera similar a la implementación de caracteres, MateriaSource solo puede tener 4. No nece sariamente tiene que ser único.

createMateria devuelve el MateriaSource aprendido previamente. Si no se conoce el MateriaSour ce correspondiente al valor de cadena ingresado como argumento, se devuelve 0.

En pocas palabras, debe poder aprender las plantillas de MateriaSource y luego crearlas cuando l as necesite. Luego, se puede crear una nueva Materia distinguiéndola solo por una cadena.

Las pruebas y resultados requeridos son los siguientes:

```
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;
}
```

```
$> clang++ -W -Wall -Werror *.cpp
$> ./a.out | cat -e
* shoots an ice bolt at bob *$
* heals bob's wounds *$
```

```
# include "ICharacter.hpp"

class AMateria;

class Character : public ICharacter
{
 private:
        AMateria* inventory[4];
        std::string name;
 public:
        Character(void);
        Character(std::string name);
        Character(const Character& obj);
        Character& operator=(const Character& obj);
        ~Character(void);
        std::string const & getName() const;
        void equip(AMateria* m);
        void unequip(int idx);
        void use(int idx, ICharacter& target);
        AMateria* getInventory(int idx) const;
};
```

우선 ICharacter / IMateriaSource는 서브젝트에서 준 그대로 사용하면 된다. 이 둘은 오히려 고치는 게 문제일 것 같다고 생각했는데, 다행히 평가표에서도 따로 고치지는 않았는지 체크하는 구문이 있었다. AMateria 도 서브젝트에서 요구하는대로 만들면 되며, ice / cure도 크게 어렵지 않으니 여기까진 따로 설명이 필요 없을 것 같다.

별생각 없이 여기까지 만들다 보면 character를 만들 때쯤 돼서 뭔가 이상함을 느끼게 된다. 바로 헤더의 상호 참조 문제이다. AMateria에서 use의 인자 target의 자료형으로 ICharacter를 사용하기 위해 해당 헤더를 넣어뒀는데, 다시 보니 ICharacter에서도 equip의 인자 자료형이 AMateria이기 때문에 서로를 참조하게 되어 컴파일이 되지 않는다.

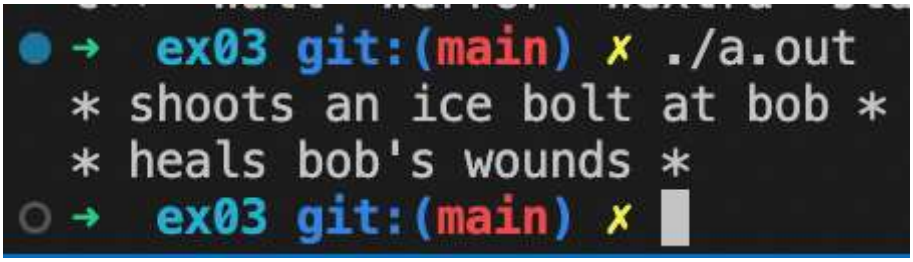
이를 해결하기 위해 클래스를 전방 선언해주어야 한다. 전방 선언이란 이러한 클래스가 있다고 미리 선언하되, 해당 내용에 대해서는 필요한 경우에만 소스 파일(.cpp 파일)에 따로 헤더를 선언해주는 것을 말한다.

위 코드를 예시로 들면 class AMateria;라고 미리 적어두면, 다른 클래스에서 이러한 클래스가 있다는 사실을 미리 알 수 있다. 그 이후에 소스 코드에서 AMateria를 사용한다면, 해당 소스 파일에서 AMateria의 헤더를 선언하여 사용하게 된다.

전방 선언의 단점은 해당 클래스의 객체를 만들 때, 위 코드의 AMateria *inventory처럼 객체는 포인터로만 사용할 수 있다. 이유는 당연하게도 여기서는 해당 이름을 가진 클래스가 있다는 사실만 알뿐, 해당 클래스의 크기를 알 수 없기 때문에, 크기를 지정해야 하는 객체를 만들 수가 없기 때문이다. 다만, 인자나 반환 값의 자료형은 클래스의 크기와 상관없기 때문에 사용할 수 있다. (다만 위 헤더에서는 AMateria 가 추상 클래스이기 때문에 equip의 인자에도 해당 클래스의 객체가 직접 올 수는 없고 포인터를 사용해야 한다.)

따라서 작성한 파일들이 서로를 순환 참조하지 않도록 클래스를 전방 선언하도록 변경해주면 된다.

```
int main()
{
        IMateriaSource* src = new MateriaSource();
        ICharacter* me = new Character("me");
        ICharacter* bob = new Character("bob");
        AMateria* tmp_ice = NULL;
        AMateria* tmp_cure = NULL;

        src->learnMateria(new Ice());
        src->learnMateria(new Cure());
        tmp_ice = src->createMateria("ice");
        me->equip(tmp_ice);
        tmp_cure = src->createMateria("cure");
        me->equip(tmp_cure);
        me->use(0, *bob);
        me->use(1, *bob);
        me->unequip(0);
        me->unequip(1);
        delete tmp_ice;
        tmp_ice = NULL;
        delete tmp_cure;
        tmp_cure = NULL;
        delete bob;
        delete me;
        delete src;
        return 0;
}
```



출력을 이것만 요구했으므로 다른 출력은 다 제거해준다.

과제에서 unequip이 delete를 동반하지 않도록 제약이 걸려있어서, 어떻게 처리해야 하나 한참 고민하다가 다른 사람들에게 물어봤는데, 그냥 메인에서 해당 메모리를 관리하는 식으로 메인 문을 만들었다고 해서 나도 그렇게 만들었다.

뭔가 코드를 보면 입맛도 텁텁하니 마음에 안 드는데 해당 제약조건을 우회한 채로 메모리 릭이 없도록 데이터를 관리할 수 있는 방법이 떠오르지 않았다. 더 좋은 해결책이 있으면 나중에라도 바꿀 예정이다.

https://blog.naver.com/hyungjungkim/60202456568

[C++] 클래스 상호 참조 해결 방법

C++ 프로그램을 하다보면 클래스를 정의하는 경우가 빈번히 발생하지요. 그리고 이렇게 정의된 클래스를 ...

blog.naver.com

https://coding-restaurant.tistory.com/504

[C++] 전방선언 (Forward Declaration)

C++에서 미리 함수를 정의하지 않으면 순차적으로 코드를 읽어들여 오류를 발생시킨다. (식별자 찾을 수 없음) 그러나 전방선언이 이뤄졌다면 컴파일러는 오...

coding-restaurant.tistory.com

https://ju3un.github.io/c++-forward-declaration/

C++ 전방 선언 (Forward Declaration)

전방 선언 (Forward Declaration) 식별자를 정의하기 전에 식별자의 존재를 컴파일러에게 미리 알리는 방식이다. 예를 들어보자. class A.h / class A.cpp c...

ju3un.github.io

https://dydtjr1128.github.io/effectivec++/2019/07/25/Effective-Cpp-item-11.html

C++ Operator=에서는 자기대입에 대한 처리가...

Do not take away the processing of self-admission in Operator= operator=라는 대입 연산자를 구현할 때, 객체가 자신...

dydtjr1128.github.io

'**프로그래밍**' 카테고리의 다른 글

[42서울] CPP Module 06 - 형변환 (1)	2022.09.11
[42서울] CPP Module 05 - 재사용성과 예외처리 (3)	2022.09.10
[42서울] CPP Module 03 - 클래스 상속 (0)	2022.08.18
[42서울] CPP Module 02 - 고정 소수점 클래스 만들기 (0)	2022.08.18
[42서울] CPP Module 01 - 클래스와 레퍼런스 (0)	2022.08.10

태그

#42서울, #cpp

'프로그래밍' Related Articles


[42서울] CPP Mod…


[42서울] CPP Mod…


[42서울] CPP Mod…


[42서울] CPP Mod…

나의 기술부채 청산일지

어그로동크 님의 블로그입니다.

구독하기

nombre

contraseña

Secret

Por favor ingrese sus valiosos comentarios.

댓글달기

최근 포스트

[42서울] ft_containers[4] - 맵…

[42서울] ft_containers[3] - 트…

[42서울] ft_containers[2] - 벡…

[42서울] ft_containers[1] - 스…

검색

Por favor esc

전체 방문자

[42서울] ft_containers[0] - 과…