

 dormir2.log

acceso

# módulo cpp 04

dormir2 · 3 de noviembre de 2021

[seguir](#)

0

cpp

42Seúl



▼ Vista de la lista

13 / 15



**v dormir2.log**

~20%

~52%

**ex00**

- std::tipo de cadena; (protegido)
- void hacerSonido();

Crea una clase de Perro que hereda Animal.

Crea una clase de Gato que hereda Animal.

- Hereda la clase Animal.
- void hacerSonido(); (Cada clase debe producir un sonido diferente).

Crea una clase WrongAnimal y WrongCat con las mismas condiciones.

- void hacerSonido();  
Se utiliza el método makeSound() de la clase WrongAnimal.

## principal

```
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();
    ...
}
```

## destructor virtual

Los destructores virtuales se utilizan a menudo como punteros al gestionar clases.

Como ejemplo en el proyecto, si crea la clase Animal y las clases Perro y Gato que heredan esa clase, luego ejecuta main, puede encontrar algún problema.

```
const Animal* j = new Dog();
delete j;
```

En el caso anterior, **sólo se ejecuta el destructor de la clase Animal**. En otras palabras, se produce **una fuga** porque no se ejecuta el destructor Dog . Para evitar esto, se utiliza algo llamado destructor

## v dormir2.log



```
virtual ~Animal();
```

Si agrega virtual delante del destructor como se muestra arriba, se ejecutará en el orden Destructor de perros -> Destructor de animales.

Comprobémoslo con una foto rápida.

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

Cuando no se utiliza el destructor virtual

```
The Animal is extinct.
The Animal is extinct.
The Animal is extinct.
```

Cuando se utiliza un destructor virtual

```
The Animal is extinct.
The cat is extinct.
The Animal is extinct.
The dog is extinct.
The Animal is extinct.
```

## función virtual

El valor de una función virtual se determina en tiempo de ejecución (enlace tardío). En otras palabras, **la función virtual no se ingresa mientras se ejecuta el compilador**.

En pocas palabras, `virtual void makeSound();` si está en la clase principal, crea un en el hijo y luego lo recibe del principal y lo hace, puede pensar que el compilador sigue el puntero y mira el , en lugar de mirar el . `void makeSound()`

```
Animal *a = new Cat(); a.makeSound()
```

**v dormir2.log**

Animal virtual Si el método de no tiene , el compilador verá primero  
Animal el de makeSound() .

Puedes considerar esto como la diferencia entre WrongAnimal y Animal en presencia o ausencia de funciones virtuales.

Cuando utilice makeSound(),  
ejecute makeSound() para Perro por Perro, Gato por Gato y Animal por Animal. Como referencia, en general, cuando se sobrecarga, debido a que se recibe como Animal, Animal::makeSound() se ejecuta para todos los Perros, Gatos y Animales.

Sin embargo, si se escriben juntos aquí, virtual void makeSound(); se ejecuta cada método sobrecargado.

Simplemente ejecute WrongAnimal sin agregar virtual y compare.

## animales.hpp

```
#ifndef ANIMAL_HPP
# define ANIMAL_HPP

#include <iostream>

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

#endif
```

## Animal Equivocado.hpp

```
#ifndef WRONGANIMAL_HPP
# define WRONGANIMAL_HPP

#include <iostream>

class WrongAnimal {
protected:
```

v dormir2.log



```
virtual ~WrongAnimal(),
WrongAnimal(WrongAnimal const &wronganimal);
void makeSound(void) const;
std::string getType(void) const;
WrongAnimal& operator=(WrongAnimal const &d);
};

#endif
```

## ex01

Importe el archivo anterior tal cual,

Crear una clase de cerebro

- Gato y Perro deben tener Brain\* de forma privada.
- Cat y Dog deben recibir un nuevo Brain\* usando Brain() en el constructor.
- Gato y Perro deben borrar Cerebro del destructor.
- Gato y Perro deberían poder recibir Cerebro y hacer copias profundas.
- El cerebro debe tener una matriz std::string de tamaño 100 llamada ideas.

## principal

```
int main()
{
    const Animal* j = new Dog();
    const Animal* i = new Cat();
    delete j;//should not create a leak
    delete i;

    Animal *a[4];

    for(int i=0; i<4; i++) {
        if (i < 2) {
            a[i] = new Cat();
        }
        else {
            a[i] = new Dog();
        }
    }
    for(int i=0; i<4; i++) {
        a[i]->makeSound();
    }

    for(int i=0; i<4; i++) {
        delete a[i];
    }

    Brain brain;
    Cat cat(brain);
```

## v dormir2.log



```
cat.setter( cat , 100),
dog.setter("dog" , 100);

std::cout << brain.getter() << std::endl;
std::cout << cat.getter() << std::endl;
std::cout << dog.getter() << std::endl;

return 0;
}
```

## Agregar gato

```
Cat::Cat(Brain const &brain) {
    *this->b = brain;
}
```

Si terminas así, terminarás con una copia superficial. Por lo tanto, es necesario agregar un operador en la clase cerebral.

## Cerebro::operador

```
Brain& Brain::operator=(Brain const &br) {
    std::cout << "Assignation operator called" << std::endl;
    Brain *a = new Brain();
    for(int i=0; i<100; i++) {
        a->ideas[i] = br.ideas[i];
    }
    return *a;
}
```

De esta manera, `*this->b = brain;` cuando lo hagas, podrás pasar al operador y hacer una copia profunda.

## ex02

Haga imposible crear una instancia de la clase Animal base.

## función virtual pura

Una función virtual pura significa que `virtual` la función especificada por `=0` no se define sumando . 인터페이스 Resulta más fácil pensar en ello en términos del concepto.

El modo de uso es sencillo.

## v dormir2.log



En este caso, la clase que tiene el método se convierte en una interfaz.

Por ejemplo, Animal si creas un método como ese en una clase,

```
int main()
{
    Animal *a = new Animal();
}
```

Se produce un error. En otras palabras, **la clase Animal creada como Interfaz no puede convertirse en un objeto.**

## ex03

Eh... no puedo detener esto...

Crea una clase AMateria.

Crea las clases Hielo y Cura. (Herencia de materia)

- AMateria clone(); // Copiar AMateria
  - void use(ICharacter&)
- Hielo -> " *dispara un rayo de hielo a NOMBRE* "
- Cura -> " *cura las heridas de NOMBRE* "
- NOMBRE -> IPersonaje에 있음

Cree una clase ICharacter (Interfaz).

```
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;
```

Crea una clase de personaje. (Herencia de caracteres)

- Tiene un máximo de 4 AMateria y se utiliza en orden de 0 a 3.
- A la hora de equiparte, no podrás hacerlo si tienes los 4.
- Al usar/desequitar, no puedes hacerlo si no tienes ningún arma.
- unequip no debe eliminarse.
- Al usarlo, debes enviar el objetivo a AMateria::use.

Crea una clase IMateriaSource. (Interfaz)

**v dormir2.log**

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

Crea una clase MateriaSource. (Herencia IMateriaSource)

- LearnMateria almacena Materia pasada como parámetro de la misma manera que Character.
- createMateria devuelve Materia propiedad del tipo pasado como parámetro.

# 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;
}
```


[seguir](#)
**dormir2**


próxima publicación  
módulo cpp 05


[Publicación anterior](#)

 dormir2.log

## Taller de Tokenización

Inscríbete en nuestro Taller C++ de Tokenización.

Tutellus

R

### 0 comentarios

Escribir un comentario

Escribir un comentario

### Publicaciones que podrían interesarte

#### C++ #07 Polimorfismo

07. Polimorfismo Técnica del polimorfismo El polimorfismo se refiere a la capacidad de múltiples objetos diferentes para procesar la misma función de diferentes maneras. Por ejemplo, armas como espadas, cañones y pistolas pueden realizar la misma función de "ataque" de forma diferente. image.png Entonces, sin tener que implementar realmente la función attack() en el objeto arma, es abstracto...

febrero 5, 2020 · 0 comentarios

 por underlier12

1

#### Resumen de las preguntas de la entrevista de C++

marzo 29, 2022 · 1 comentario

 por desarrollo principiante

10

**v dormir2.log**

## Taller de Tokenización

Inscríbete en nuestro Taller Gratuito de Tokenización.

광고 Tutellus



## Clases e interfaces de TypeScript

La programación orientada a objetos (OOP) es un método que puede reducir drásticamente la duplicación de código al desarrollar aplicaciones. La programación orientada a objetos divide grandes problemas en unidades llamadas clases, agrega relaciones entre clases y minimiza la duplicación de código .

19 de abril de 2020 · 1 comentario



por Daehyun Kim

5

## C++ #04 Herencia de clases

04. Herencia de clases La herencia es una de las principales características de la programación orientada a objetos, y permite estructurar jerárquicamente la estructura lógica de un programa. Una clase hija puede heredar y utilizar las propiedades de la clase padre. Por tanto, aumenta la reutilización del código fuente. imagen.png Herencia...

febrero 1, 2020 · 0 comentarios

## v dormir2.log



### C++ #03 Constructores y destructores

03. Constructor y Destructor Constructor En C++, puedes usar un constructor para crear un objeto e inicializar variables miembro al mismo tiempo. Un constructor es un método especial implementado con el mismo nombre que el nombre de la clase. Características del constructor El constructor no tiene valor de retorno El constructor se puede definir varias veces (parámetros variables) Constructor predeterminado Si no implementa un constructor separado en C++...

febrero 1, 2020 · 0 comentarios

w por underlier12

0

```
AnnotationChild extends Annotation
{
    int annotationParentMethod(int num1, int num2)
    {
        System.out.print("num1 + num2 : ");
        return num1 + num2;
    }
}
```

### ¿Qué significa la anotación @Override y por qué se utiliza?

Las anotaciones aparecieron a partir de JDK5 y se refieren al uso de @ en clases, métodos y variables. Anotación significa un comentario en el diccionario. Aunque su función es diferente a la de las anotaciones, se puede adjuntar como un comentario para darle un significado especial y permitir la inyección de...

16 de mayo de 2021 · 0 comentarios

por Doa Choi

6

### [c++] Puntero de función, puntero de función miembro

Puntero de función, puntero de función de un objeto.

27 de diciembre de 2021 · 0 comentarios

por Chae Myeong-seok

1

 dormir2.log

junio 2, 2021 · 1 comentario



por Codren

2

## [Java] Manejo de excepciones

Resumen del contenido de las clases de Java del 15 y 17 de octubre de 2019. Obtenga más información sobre el manejo de excepciones.

octubre 18, 2019 · 0 comentarios



por tomate

1

# 정적 팩토리 메서드 (Static Factory Method)

## Método de fábrica estática

¿Qué es un método de fábrica estático?

17 de abril de 2022 · 0 comentarios



por Jihwan Choi

9

**v dormir2.log**