

Herencia múltiple en C++

(Multiple inheritance in C++)

TPV2
Samir Genaim

Herencia Múltiple

- ◆ ¿Qué es la herencia múltiple?
- ◆ ¿Cómo se usa en C++?
- ◆ ¿Qué problemas introduce y cómo se resuelven?
- ◆ ¿Cuándo se usa la herencia múltiple?
 - herencia de interfaces
 - herencia de implementación
- ◆ ¿Cómo se usa en otros lenguajes de programación?

¿Qué es la herencia múltiple?

El concepto básico es bastante simple:

Se puede crear un nuevo tipo **A** (una clase) heredando de más una clase base **B₁,...,B_n**

A tiene “varias caras”:

- **A** es **B₁**
- **A** es **B₂**
- ...
- **A** es **B_n**

Ejemplo de Herencia Múltiple: I

```
class Keyboard {  
    char key;  
public:  
    Keyboard() : key(0) {}  
    void press(char c) { key = c; }  
    char getPressedKey() { return key; }  
    ...  
};
```

```
class Mouse {  
    Button button;  
public:  
    Mouse() : button(NONE) {}  
    void click(Button b, int x, int y) { ... }  
    Button getClickedButton() { return button; }  
    ...  
};
```

```
class KMCombo : public Keyboard, public Mouse {  
public:  
    // add more methods  
};
```

La clase KMCombo hereda la **API** y la **implementación** de Keyboard y Mouse

Ejemplo de Herencia Múltiple: II

```
class Animal {  
public:  
    void virtual eat(/*...*/) = 0;  
    ...  
};
```

```
class Drawing {  
public:  
    void virtual draw(/*...*/) = 0;  
    ...  
};
```

```
class Dog : public Animal, public Drawing {  
public:  
    void eat(/*...*/) { ... }  
    void draw(/*...*/) { ... }  
    ...  
};
```

La clase Dog hereda la **API** de las clases Animal y Drawing

Ejemplo de Herencia Múltiple: III

```
class InFile {  
public:  
    void read() {  
        cout << "reading ..." << endl;  
    }  
};
```

```
class OutFile {  
public:  
    void write() {  
        cout << "writing ..." << endl;  
    }  
};
```

```
class IOFile : public InFile, public OutFile {  
    ...  
};
```

```
int main() {  
    IOFile f;  
  
    f.read();  
    f.write();  
}
```

Herencia Múltiple: ambigüedad

```
class InFile {  
public:  
    void open() { ... }  
    void read() {  
        cout << "reading ..." << endl;  
    }  
};
```

```
class OutFile {  
public:  
    void open() { ... }  
    void write() {  
        cout << "writing ..." << endl;  
    }  
};
```

```
class IOFile : public InFile, public OutFile {  
    ...  
};
```

```
int main() {  
    IOFile f;  
  
    f.open();  
    f.read();  
    f.write();  
}
```

La llamada es ambigua, tenemos que
cualificar:
f.InFile::open o f.OutFile::open

El Problema del Diamante

```
class File {  
public:  
    string name_;  
    void open(...){ ... }  
};
```

```
class InFile : public File {  
public:  
    void read() {  
        cout << "reading ..." << endl;  
    }  
};
```

```
class OutFile : public File {  
public:  
    void write() {  
        cout << "writing ..." << endl;  
    }  
};
```

```
class IOFile : public InFile, public OutFile {  
}; ...
```

Ambigüedad; `IOFile` tiene 2 copias de `open` y dos copias de `name_`:

`f.InFile::open(); f.OutFile::open(); f.InFile::name_; f.OutFile::name_`

Herencia Virtual

Usando la **herencia virtual** para heredar de una clase, garantizamos tener sólo una copia de dicha clase en cualquier clase que la hereda (de manera virtual), directamente o indirectamente.

```
class File {  
public:  
    string name_;  
    void open(...){ ... }  
};
```

IOFile f

f.OutFile::open, f.InFile::open
y f.open refieren a open de
la clase File. Igual para el
atributo name_ ...

```
class InFile : virtual public File {  
public:  
}; ...
```

```
class OutFile : virtual public File {  
public:  
}; ...
```

```
class IOFile : public InFile, public OutFile {  
...  
};
```

Herencia Virtual: inicialización

La clase que está más abajo en la jerarquía tiene que llamar a la constructora de la clase base (se ignoran las otras llamadas a dicha constructora en la jerarquía)

```
class File {  
    string name_;  
public:  
    File(string name) : name_(name) { ... }  
    void open() { ... }  
};
```

```
class InFile : virtual public File {  
public:  
    InFile(string name) : File(name) {}  
    void read() {  
        cout << "reading ..." << endl;  
    }  
};
```

```
class OutFile : virtual public File {  
public:  
    OutFile(string name) : File(name) {}  
    void write() {  
        cout << "writing ..." << endl;  
    }  
};
```

```
class IOFile : public InFile, public OutFile {  
public:  
    IOFile(string name) : File(name), InFile(name), OutFile(name) {}  
};
```

! Es muy complicado i

Si la herencia múltiple es tan complicada, ¿merece la pena usarla?

Interface Segregation Principle

Interface segregation principle – ISP

(Principio de separación de interfaces)

Los clientes de un programa sólo deberían conocer de aquellos métodos que realmente usan, y no aquellos que no necesitan usar

Ejemplo de ISP

Necesitamos construir un modelo (clase) para modular a una persona que se llama Andy.

Andy sabe hacer muchas cosas, en su trabajo como ingeniero, con sus padres, con sus amigos, etc.

Si construimos sólo una clase para modular Andy, todo el mundo tiene que saber de todo lo que puede hacer Andy para poder comunicar con él correctamente ... y todo el mundo puede comunicarse con Andy de cualquier manera ...

```
class Andy {  
    public:  
        // 400 APIs  
};
```

Ejemplo de ISP

Una solución mejor sería definir la distintas funcionalidad de Andy usando clases distintas, con APIs mucho más pequeños, y heredar todas esas clases para definir a Andy ...

```
class Engineer {  
public:  
    // 15 APIs  
};
```

```
class Son {  
public:  
    // 10 APIs  
};
```

```
class Friend {  
public:  
    // 20 APIs  
};
```

...

```
class Andy : public Engineer, public Son, ... {  
public:  
    // 400 APIs  
};
```

Ejemplo de ISP

```
void colleague(Engineer& a) {  
    // talk to Andy as an Engineer  
}  
  
void parents(Son& a) {  
    // talk to Andy as Son  
}  
  
...  
int main() {  
    Andy x;  
  
    colleague(x);  
    parents(x);  
  
    ...  
}
```



Cada uno sabe sólo lo que necesita de Andy. Es mucho más fácil para ellos entender cómo comunicarse con Andy y ademas no pueden usar medios de comunicación que no deben usar

Otro Ejemplo de ISP: I

- ◆ Queremos escribir un juego con actores que disparan
- ◆ Necesitamos permitir a un actor disparar, es decir añadir balas al juego
- ◆ Al mismo tiempo el Game Engine necesita dibujar todas las balas activas durante el juego

Otro Ejemplo de ISP: II

Se puede implementar las 2 funcionalidades en una sola clase que hereda de GameObject

- mantiene una lista de balas
- su update()/render() actualiza y dibuja las balas
- ademas, tiene un método shoot que permite disparar

Pasamos una instancia de esa clase a los actores que pueden disparar para poder llamar a shoot() y al Game Engine para llamar a render() y update()

Otro Ejemplo de ISP: III

```
class GameObject {  
public:  
    GameObject();  
    virtual ~GameObject();  
    virtual void render() = 0;  
    virtual void update() = 0;  
};
```

El problema con este diseño es que los actores tienen acceso a métodos al que no tienen que tener acceso, como render() y update().



```
class BulletsManager : public GameObject {  
public:  
    BulletsManager(...) { ... }  
    virtual void render() { ... }  
    virtual void update() { ... }  
    virtual void shoot() { ... }  
private:  
    vector<Bullet*> bullets_;  
};
```

Otro Ejemplo de ISP: III

Otra posibilidad es

- Definir una interfaz BulletsManager que incluye sólo el método shoot()
- Implementar un StarWarsBulletsManager como antes pero heredando de GameObject y de BulletsManager

Una instancia de StarWarsBulletsManager se pasa como

- BulletsManager a los actores que disparan
- GameObject al Game Engine

Cada uno ve sólo lo que necesita y no puede usar lo que no debe usar...

Otro Ejemplo de ISP

```
class GameObject {  
public:  
    GameObject();  
    virtual ~GameObject();  
    virtual void render() = 0;  
    virtual void update() = 0;  
};
```

```
class BulletsManager {  
public:  
    BulletsManager();  
    virtual ~BulletsManager();  
    virtual void shoot() = 0;  
};
```

```
class StarWarsBulletsManager : public GameObject, public BulletsManager {  
public:  
    BulletsManager(...) { ... }  
    virtual void render() { ... }  
    virtual void update() { ... }  
    virtual void shoot() { ... }  
private:  
    vector<Bullet*> bullets_;  
};
```

Otro Uso de Herencia Múltiple

- ◆ Suponemos que tenemos un Game Engine que queremos usar para desarrollar un videojuego ...
- ◆ ... pero no nos gusta la parte para comprobar colisiones entre objetos de juego y nos gustaría usar otra librería muy eficiente para comprobar colisiones entre objetos
- ◆ El problema es que los 2 sistemas usan tipos distintos, el Game Engine usa el tipo **GameObject** y la librería de colisiones usa el tipo **PhysicalObject**.
- ◆ No es inmediato usar los 2 sistemas con los mismos objetos de juego porque uno tiene que verlos como **GameObject(s)** y el otra como **PhysicalObject(s)**

Interoperabilidad

- ◆ Modificar uno de los sistemas no es una opción, es lo peor que podemos hacer. Incluso puede ser que el código fuente no esté disponible ...
- ◆ Hay muchas soluciones que nos permiten trabajar con los 2 sistemas a la vez de manera transparente, una de ellas es mediante la herencia multiple.

Interoperabilidad: CollisionManager

```
class CollisionManger {  
public:  
    CollisionManger();  
    virtual ~CollisionManger();  
    bool checkCollision(PhysicalObject* a, PhysicalObject* b);  
    ...  
};
```

```
class PhysicalObject {  
public:  
    PhysicalObject();  
    virtual ~PhysicalObject();  
    virtual double getObjectWidth() = 0;  
    virtual double getObjectHeight() = 0;  
    virtual double getObjectXPos() = 0;  
    virtual double getObjectYPos() = 0;  
};
```

Interoperabilidad: GameEngine

```
class GameObject {  
public:  
    GameObject(double width, double height, double x, double y);  
    virtual ~GameObject();  
    virtual void render() = 0;  
    virtual void update() = 0;  
    ...  
protected:  
    double width_;  
    double height_;  
    double x_;  
    double y_;  
};
```

```
class GameEngine {  
public:  
    GameEngine();  
    virtual ~GameEngine();  
    void addGameObject(GameObject* o);  
    void update();  
    void render();  
    ...  
};
```

Interoperabilidad: MyGameObject

```
class MyGameObject : public GameObject, public PhysicalObject {  
public:  
    MyGameObject(double width, double height, double x, double y) :  
        GameObject(width, height, x, y) { ... }  
    virtual ~MyGameObject() { ... }  
    virtual double getObjectWidth() { return width_; }  
    virtual double getObjectHeight() { return height_; }  
    virtual double getObjectXPos() { return x_; }  
    virtual double getObjectYPos() { return y_; }  
    virtual void render() = 0;  
    virtual void update() = 0;  
};
```

MyGameObject tiene 2 “caras”, una que entiende el Game Engine y otra que entiende el Game Manager

Interoperabilidad: Hero & Enemy

```
class MyGameObject : public GameObject, public PhysicalObject {  
public:  
    ...  
};
```



```
class Hero: public MyGameObject {  
public:  
    Hero(...) : MyGameObject(...) { ... }  
    virtual ~Hero() { ... }  
    virtual void render() { ... }  
    virtual void update() { ... }  
};
```

```
class Enemy : public MyGameObject {  
public:  
    Enemy(...) : MyGameObject(...) { ... }  
    virtual ~Enemy() { ... }  
    virtual void render() { ... }  
    virtual void update() { ... }  
};
```

Interoperabilidad: game

```
void game() {  
    GameEngine e;  
    CollisionManger c;  
  
    Hero a(5, 5, 10, 10);  
    Enemy b(5, 5, 12, 12);  
  
    e.addGameObject(&a);  
    e.addGameObject(&b);  
  
    e.render();  
  
    cout << "collision? " << (c.checkCollision(&a, &b) ? "Yes" : "No") << endl;  
  
    e.update();  
    e.render();  
  
    cout << "collision? " << (c.checkCollision(&a, &b) ? "Yes" : "No") << endl;  
}
```

Hero y Enemy tienen 2 “caras”, una que entiende el Game Engine y otra que entiende el Game Manager

Herencia: Interfaz/Implementación

- ◆ Lo que complica la herencia múltiple es la herencia de implementación, no la de interfaz ...
- ◆ Si nos limitamos a herencia multiple de clases abstractas puras (incluyen sólo métodos abstractos puros), o herencia de implementación única, no introducimos ninguna ambigüedad, además podemos usarla para el ISP, interoperabilidad, etc.
- ◆ **Conclusión:** si es posible, evitar la herencia múltiple de implementación, usar la herencia múltiple sólo con clases abstractas puras.
- ◆ En Java y C# se permite sólo herencia simple y herencia multiple de interfaces (que son equivalentes a clases abstractas puras de C++)